Noname manuscript No.
(will be inserted by the editor)

A Survey on Transactional Stream Processing

Shuhao Zhang · **Juan Soto** · **Volker Markl**

Received: date / Accepted: date

Acknowledgements This work is supported by the National Research Foundation, Singapore and 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.

Abstract Transactional stream processing (TSP) has been increasingly gaining traction. TSP aims to provide a single unified model that offers both transaction- and stream-oriented guarantees. Over the past decade, considerable efforts have resulted in the development of alternative TSP systems, which enables us to explore the commonalities and differences across these solutions. However, a widely accepted standard approach to the integration of transactional functionality with stream processing is still lacking. Existing TSP systems typically focus on a limited number of application features with non-trivial design trade-offs. This survey initially examines diverse transaction models over streams and TSP specific transactional properties, followed by a discussion on the consequences of certain design decisions on system implementations. Subsequently, we highlight a set of representative scenarios, where TSP is employed, as well as discuss some open problems. The aim of this survey is twofold. First, to provide insight into disparate TSP requirements and techniques. Second, to engage the design and development of novel TSP systems.

Keywords Transactions · Stateful Stream Processing · standardization · Survey

✉ Shuhao Zhang
Singapore University of Technology and Design, Singapore
E-mail: shuhao_zhang@sutd.edu.sg

Juan Soto
Technische Universität Berlin, Germany
E-mail: juan.soto@tu-berlin.de

Volker Markl
Technische Universität Berlin, Germany
E-mail: volker.markl@tu-berlin.de

1 Introduction

Transactional stream processing (TSP) can be broadly defined as processing streaming data with transactional correctness guarantees [112]. These guarantees not only include properties that are intrinsic to stream processing (e.g., time order, exactly-once semantics), but also the `ACID` properties in traditional OLTP-oriented databases. In this regard, TSP brings transaction semantics to stream processing, and vice versa, resulting in a unified system that combines the best of both worlds: transaction processing and stream processing.

Consider an example use case involving self-driving vehicle monitoring [77], where vehicles continuously generate status data via their sensors. By employing a stream processing engine (SPE) many services would become available, such as enabling a warning or providing a list of nearby gas stations when the amount of remaining fuel in the vehicle is below a certain threshold. In this example, while processing the flood of streaming sensor data from vehicles, it is crucial to maintain consistent and up-to-date states of gas stations and roads in the system. This is challenging to achieve because the processing of input data from different vehicles can modify *shared application states*, such as the status of a common gas station.

By employing modern SPEs, existing workarounds, such as using external databases to store the shared application states can lead to significant extra programming efforts [125], poor system performance [78] and even

incorrect results [37, 53]. This problem is exacerbated if more complex shared mutable state storage and retrieval queries, such as range look-ups are further required. In contrast, TSP systematically manages concurrent accesses to shared application states. To ensure correctness, state accesses are performed transactionally with transactional properties being guaranteed.

However, the requirements and specifications of TSP have not yet been classified in a comprehensive way and lack common definitions, leading to a great deal of variation in the supported functionalities and the achieved performances. In particular, different research groups have developed their own definitions and corresponding implementations of *transactional stream processing systems (TSPSs)* over the past decade [14, 31, 44, 53, 59, 89, 134]. Without a thorough understanding of how the various proposals complement one another, it is difficult for researchers to improve upon the state-of-the-art. Furthermore, it also brings significant challenges for users to adopt a particular design or system for their application. As a result, a common semantic model across disparate systems has yet to emerge. This is largely due to the diverse number of applications proposed in the literature and the fact that each system focuses on its own particular application features, which impose implicit assumptions and objectives. To help researchers, particularly from different backgrounds, to gain a better understanding of this area, we systematically summarize prior works involving TSPSs.

Outline of the Survey. This survey aims to provide an overview of TSP as well as our vision for TSP in the foreseeable future. The organization of the survey is as follows: i) *Background:* We introduce the basics of stream processing, offer a brief history of stream processing, and an overview on TSP in Section 2; ii) *Transaction Models over Streams:* We discuss different transaction models to bridge the transaction processing paradigms and stream processing in Section 3; iii) *Properties of Transactional Stream Processing:* We review various transactional properties that define what is considered a correct execution in TSPSs in Section 4; iv) *Execution Mechanisms:* We discuss relevant execution mechanisms in TSPSs, including execution and fault tolerance artifacts of TSPSs in Section 5. v) *Aspects of Transactional Stream Processing Engines:* We summarize three key system design aspects: APIs, system architectures, and state representations proposed in prior works to support TSPSs in Section 6; and vi) *Advanced Issues of Transactional Stream Processing:* We compiled a set of representative use cases that cover a wide range of application features in Section 7. In Section 8, we discuss several related research directions that partially overlap with TSP, but are actually orthogonal. In Section 9, promising new research directions are underscored. To the best of our knowledge, this is the first attempt at understanding how different TSPSs are designed for specific application scenarios and whether they can be applied to other use cases. Our review of related use cases involving TSPSs also leads to a discussion on what challenges remain to be solved in future systems.

2 Background

In this section, we will first provide an overview of stream processing, followed by an overview of transactional stream processing.

2.1 Stream Processing Overview

We first introduce the basics of stream processing and state management of stream processing and then offer a brief history of stream processing.

2.1.1 Basics of Stream Processing

Stonebraker [107] refers to *stream processing* as "a class of software systems that deals with processing streams of high-volume messages with very low latency." Associated with stream processing are numerous key terms, including the notion of *state*, *event*, *timestamp*, *stream query*, and *stream operator*. We will be drawing on several sources to define these terms and concepts, including Abadi et al. [13].

The notion of *state* arises in numerous contexts, including stream processing [13] systems. We define *state* [116] to be "the intermediate value of a specific computation that will be used in subsequent operations during the processing of a data flow."[1] The state is a key concept in stream processing, since comparing the "present" with the "past" is common in many applications (e.g., when computing moving averages).

A (data) *stream* [107] is a "sequence of data items that collectively describe one or more underlying signals," such as a network traffic stream, which describes the type and volume of data transmitted among nodes in a network. An *event* e is a 3-tuple $e =< t,k,v >$, where t, k and v are the timestamp, key, and payload, respectively. A *timestamp* specifies the time when an event took place. When data items arrive out of sequence (i.e., not in chronological order), a stream is referred to as being *out-of-order*. The *key* refers to an attribute or set of attributes associated with a certain state in stream processing. The *payload* refers to data values to be stored as states in stream processing, functions to modify states, or values to participate in certain computations during stream processing.

[1] This definition differs from its common use in traditional database systems, where a state is a set of relational tables at a specific point in time.

For handling infinite streams, bounded subsets of streams are referred to as *windows* [119]. There are many types of windows, including tumbling windows, sliding windows, and session windows. Tumbling and sliding windows discretize a stream into windows of fixed length l. Additionally, sliding windows define a slide step l_s that declares how often new windows start. Thus, records are assigned to multiple, concurrent, potentially overlapping sliding windows if $l_s < l$. In contrast, session windows end if no record is received for a time l_g (session gap) after a period of activity. A *stream query* is comprised of a collection of *operators* that continuously process events [133]. Operators are organized into DAGs (directed acyclic graphs), where each vertex corresponds to an operator and each edge represents an event flowing downstream from the producer operator to the consumer operator. To sustain a high input stream ingress rate, each operator in a stream query may be spread across multiple *executors* (e.g., Java threads). Each of which handles multiple input events concurrently via stream partitioning [133].

2.1.2 State Management of Stream Processing

The need for explicit *state management* originates from the need to keep and automate the maintenance of a persistent state for event-driven applications in a reliable manner. Typically, a state is categorized into one of two types: 1) *read-only state*: where applications look up read-only data to get the information required to process input events. 2) *read-write state*: where some states are maintained and updated as stream events continue to be processed. The management of the read-write state is particularly challenging, especially when the processing of different input events relies on reading/writing to the same state.

Due to diverse system requirements, such as managing states beyond main memory, elastic scaling, and migrating states among shared-nothing architectures, SPEs were designed to be fully aware of states, to relieve the burden on developers. SPEs with built-in state management support are known as *stateful SPEs* [36]. For more information about state management, see the following survey [116]. The usage of state has evolved over time. For example, early stream processing engines (SPEs) adopted a bounded memory model for predefined relational stream operations (e.g., window aggregation) to keep their intermediate computing results. Consequently, since the state size cannot exceed the pre-allocated space, the maintained state quickly becomes an approximation (e.g., sketch or sample), trading off precision or correctness with memory consumption.

Other common uses of state, include counters aggregated over windows of records and buffered data for a join [17]. Besides storing simple intermediate results, there are some novel state representations required during stream processing, for example for graph data structures [136] and transactional records [78], which is the focus of this survey.

2.1.3 A Brief History of Stream Processing

Increased automation in healthcare, transportation [83, 92], finance, and the IoT [117], among other market sectors has led to ever faster data generation rates. For example, Alibaba in November 2019, reported the need to handle 2 billion requests/second on Singles Day [1]. Unfortunately, traditional database systems, which store and index data before processing it, are ill-suited to handle large volumes of data ingested in real time. In contrast, SPEs allow users to build applications that are able to achieve high performance for very large data volumes. They offer scalability and provide fault tolerance. Over the past decade, many SPEs [12, 13, 36, 57] have been proposed from academia and industry.

Early SPEs (1992-2010) were often extensions of existing relational database systems. Driven by the emergence of sensor-based applications, the initial design goal of early SPEs was simply to process continuous queries with low latency data ingested into the system. Among the early SPEs were Tapestry [114], TelegraphCQ [39], Aurora [13], and STREAM [20, 57], which run on a single machine and processes ordered event streams. Most early SPEs were designed to handle window queries (e.g., sliding window aggregation) over data streams, which is today considered to be classical. Later, advanced features, such as fault tolerance [12], adaptive query processing [96] and more complex query expressions [26] (e.g., complex event processing) were proposed along with some new SPEs, such as Borealis [12], System-S [52].

The design of modern SPEs (2010-) was largely influenced by the *MapReduce* paradigm and the trend towards cloud computing. Notably, modern SPEs have two key features. They are both scalable over a cluster of commodity machines and highly fault-tolerant. Among the modern SPEs are S4 [82], Storm [118], Heron [69], Flink [36], Spark Streaming [131], Samza [84], and the recent Kafka Streams [3]. The emergence of massively parallel processors with multi-terabyte storage capacity and network bandwidths exceeding several gigabytes/second has led to a burst of activity over the past five years, with the objective to enable hardware-conscious stream processing [58, 67, 80, 104, 115, 132, 133, 137]. However, APIs and the support for query semantics were inherited from earlier SPEs. Despite the fact that they support the same types of applications with stateless or simple stateful queries, modern SPEs offer much higher performance, greater energy efficiency, and improved scalability via the

use of modern hardware, such as multi-/many-core CPUs, GPUs, and FPGAs [135].

More recently, hybrid system architectures that integrate streaming with other forms of data processing (e.g., OLTP, OLAP, batch, interactive) have emerged [14, 18, 38, 78, 94, 134]. Storage management and correctness guarantees have become ever more critical [112]. However, mainstream SPEs, such as Spark Streaming [131] and Flink [36], offer limited support for storage management. Unfortunately, this is undesirable in many cases [31, 98, 125], including computer simulation, task-driven model training for machine learning, and graph aggregation, since these require the availability of *shared mutable state*, where multiple threads spawned from the same stream application can be referenced and updated. Such demands motivate increasing attention on *transactional stream processing*, which accommodates transaction and stream processing into a unified model. Although this approach is unconventional and challenging to realize, it has great potential for supporting non-key-parallel modes of operation, model optimization algorithms, and more complex data representations that are hardly divisible [35].

2.2 Transactional Stream Processing: An Overview

The idea of mixing relational queries with continuous stream processing had been proposed ever since the first generation SPE was developed [54]. Such a mixture also naturally permits (the aforementioned) shared mutable state management during stream processing. However, early SPEs simply disallow maintaining relational tables [23]. In contrast, others allow (shared) relational tables to be queried during continuous query processing, but implicitly assume that relations remain unchanged, at least throughout the lifetime of the query [13, 75].

Stream applications are typically expressed with a sequence of operators (e.g., a map function) and each operator may need to maintain states during processing for future reference [34]. Due to the lack of transactional semantics, modern SPEs restrict each execution entity (e.g., each operator/thread), to maintain a disjoint subset of states (or partitioned states). Thereby, disallowing or avoiding state sharing in the system. For example, Flink [36] achieves high performance via the disjoint partitioning of application states, often through key-based partitioning [65], so that each operator maintains a disjoint subset of the application states. However, the forced partitioning of states could lower performance across many uses cases, due to the tedious task for developers to establish a workaround, such as keeping state in a shared external storage system [10]. We will return to this matter in Section 7.

Integrating both streaming and transactional facilities into mainstream SPEs is a nontrivial task. For example,

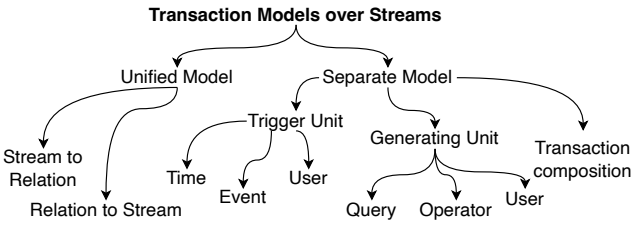


Fig. 1: Transaction models of TSP.

to schedule continuous updates to a relational table during stream processing, while the table is concurrently being visited by ad-hoc transactional queries is challenging. Even defining correctness in such a case can be quite tricky due to a fundamental difference in the design goals of transactional databases and SPEs. In particular, transactional databases have a clear notation for transactions and well-defined transactional properties, whereas SPEs typically sacrifice the support for well-defined relational semantics and commonly do not guarantee transactional consistency. In the face of such a non-trivial task, many prior works [9, 15, 20, 31, 47, 57, 77] incorporate either transactional semantics into SPEs or continuous stream processing capabilities into relational database systems. For the purposes of this survey, the term – transactional SPEs (TSPSs) – refers to this novel class of systems.

What makes TSPSs unique is that they allow and sometimes encourage a system to maintain shared mutable states, which can be accessed by multiple entities (e.g., concurrent stream queries, ad-hoc queries [78, 134]). In particular, concurrent access (i.e., read or write) to mutable shared states must satisfy predefined constraints to ensure some form of transactional properties, which may be further customized. In prior works [78, 134], alternative definitions and implementations of TSPSs have been proposed, which we present comprehensively in detail in subsequent sections.

3 Transaction Models over Streams

A standard that is widely accepted on transaction models over data streams has yet to emerge [111]. For example, in batch processing, particularly, in transactional databases, new (periodically incoming) queries are executed over finite and unordered data sets. In contrast, in stream processing, continuous queries are executed over conceptually infinite and time-ordered data streams. Consequently, it is difficult to formally compare querying under these two paradigms in terms of their execution orderings, schemas, and consistency requirements, which can potentially lead to *dilemma query results* (see Remark 2 in Section 4.2.2). Nonetheless, many related proposals have been raised in the last decade. In particular, researchers have proposed to either combine

the two paradigms and treat them as one, or employ both paradigms, but select each one individually. Next, we discuss these two competing approaches.

3.1 Unified Execution

To unify the two processing paradigms, we can transform each one into the other. That is, treat relational operations as streaming input events or treat streaming data as sources with regular read/write operations. Both approaches have been explored in the research literature. However, a formal theoretical and empirical comparison between them is still an open problem.

3.1.1 Mapping Relational Data to Data Streams

Relational data sources can be transformed into data streams by associating each relational operation with a stream event. Shaikh et al. [101] propose an incremental continuous query processing approach with isolation guarantees that transforms relational updates into stream tuples. However, since relations may have been updated during join processing, an inconsistency in the results may arise. To mitigate this, the authors employ a mapping table as a transform. To track modifications, the SPE-owned database must be associated with a trigger. Once a trigger is fired, the modification request is sent to the SPE as an input event (e). Upon processing e, the SPE will incrementally inform all of the participating operators about the modification. Since e is timestamped and each event is assumed to be executed in order, such a mapping ensures the desired isolation properties: when processing each input event, all of the operators will see a consistent database. Via experimental evaluation, the authors demonstrate the efficiency of their proposed approach to solve the aforementioned inconsistency problem. However, their approach has not been extended for generic stream processing use cases.

3.1.2 Mapping Data Streams to Relational Data

Data streams can be transformed into relational data. For example, in the case of Botan et al. [31], all data sources (e.g., relational, non-relational, streaming) are treated uniformly: they are all sets of data items on which read and write operations are executed, in accordance with the page model [126]. There are four relational operations: *insert,* which adds new tuples, *delete,* which removes tuples, *value update,* which replaces existing tuples attribute values, and *read,* which obtains the values of tuples.

To handle stream data, a special relation is employed, where input events are modelled as write operations to the relation represented by the input stream. Botan et al. [31] proposed to represent a continuous query as a (possibly infinite) sequence of one-time queries that are fired as a result of the data sources being modified (e.g., the arrival of new events, update of existing inputs) or by periodic execution (e.g., every second). Thus, one-time execution of a continuous query can be translated into reading operations on all of its input data sources plus (possibly) a number of write operations, corresponding to the results that it may generate. As another example, in the case of Meehan et al.'s [78] S-Store system, stored procedures are used to represent stream operators and transform stream processing into transaction processing (i.e., each stream operator is modelled as a transaction). In this way, S-Store only needs a single execution engine – the transaction processing engine (i.e., H-Store [108]).

Yet another example is the works proposed by Oyamada et al. [87, 88]. In their case, they propose to model a database as a source of information related to data streams and use a database to archive data streams. In this way, database transactions are triggered by the arrival of data streams. Since the arrival rate of data streams can be high, it is necessary to invoke the triggered transactions efficiently, to ensure the performance is high. To this end, they propose several transaction invocation schemes [87]. To generalize their idea to include other types of data sources (e.g., machine learning model), they subsequently introduce the concept of continuous query (CQ)-derived transactions [88], which derive read-only transactions from continuous queries. These read-only transactions can be mixed with other update requests without introducing inconsistencies in the shared application states.

3.2 Separate Execution

Under the separate execution model, instead of unifying the two paradigms, the processing of both streams and transactions is handled separately in the same system. A common approach to realising separate execution is to treat transaction processing as *transactional state accesses*, which are triggered during stream processing. However, it still needs to ensure execution correctness when the two processing paradigms interfere with one another (e.g., when transaction processing results in the modification of application states that are being referenced during stream processing). As in conventional database transactions, to ensure correctness, transactional semantics are employed and transactional state accesses are modelled as *state transactions*, but these contain additional attributes, such as trigger events and trigger timestamps.

Intuitively, since stream data are unbounded and the processing of queries is endless, the conventional notion of a transaction boundary is difficult to define in stream processing. However, determining the boundary of stream transactions is rather flexible. For example, an SPE offers

varying ways to trigger a transaction (i.e., *triggering unit*), such as per input event or per batch of events with a common timestamp. There are also alternative ways to generate a transaction (i.e., *generating unit*), such as per operator and per query. In addition, transactions can spawn other transactions. Next, we discuss the varying boundary setting approaches: triggering units, generating units, and transaction spawning.

3.2.1 Triggering Units

Traditionally, database transactions occur when user programs are explicitly triggered by user requests. In contrast, transactions in TSPSs occur when they are triggered by incoming streaming events. Triggering units define the granularity of transaction boundaries and these units can be time-based, batch event-based, single event-based, and user-defined.

Time-based Transactions. Although the STREAM [20, 57] project does not explicitly refer to *transactions*, it does assume that batches of tuples with a common timestamp are executed atomically to ensure progress correctness. That is, STREAMs processing model is *time-driven*, given that time advances from $t-1$ to t when all the events with timestamp $t-1$ have been processed. This behaviour follows that of a transactional model, where all events with a common timestamp belong to the same transaction, irrespective of whether they arrive on the same stream.

With a focus on the concurrent aggregation of sliding windows, Golab and Ozsu [53] proposed to model both updates to windows and queries on windows as transactions consisting of atomic sub-window reads and writes. In their model, a window of length of time nt is stored as a circular array of n sub-windows, each spanning a length of time t. Each window may have different values for n and t. Every t time unit, the oldest sub-window is replaced with a buffer containing incoming tuples that have arrived in the last t time units. Each long-running query Q also specifies its desired re-execution frequency. The frequency must be a multiple of t, i.e., Q will be scheduled for re-execution every m window updates, where $1 \leq m < n$. Subsequently, a snapshot query or a particular re-execution of one or more long-running queries is defined as a read-only transaction, whereas the update of each sub-window is defined as a write-only transaction.

Oyamada et al. [89] pursue a path similar to that of Golab and Ozsu [53] but in a more general setting. They assume continuous queries may trigger read-only transactions that reference a shared data resource (e.g., a relational table) and write-only transactions triggered by resource updates (e.g., ad-hoc queries that update tables). Read-only transactions consist of successive resource reference operations whose results are involved in the same downstream operation (e.g., aggregate). Thus, each window corresponds to one read-only transaction. Write-only transactions simply consist of an ad-hoc write operation to a shared resource.

Conway [47] studied the concept of transactions in data stream processing and also proposed associating transaction boundaries in a database with window boundaries in a data stream, where a window is the basic unit of data flow in an SPE. For example, windows are the unit of isolation for continuous queries, the unit of durability for archived streams, and the output streams of continuous queries themselves.

Time-based transaction triggering is also employed in commercial products. For example, Coral8 [6] specifies that the minimum recovery unit in the event of a failure is the time slice. Although time slices may be processed concurrently, each time slice is executed separately and the order between them is preserved. Time slices represent a row or a sequence of rows that have a common timestamp and arrive in the same stream or window. Similarly, in an early version of StreamBase [7], query invocation happens every second as the state transactions, even when the content of a window remains the same.

Batch Event-based Transactions. The key data structure of the DataCell [73, 74] engine is the *basket*. Its role is to store a portion of a stream in a temporary main-memory table. Queries are then evaluated against each filled basket, which contains a batch of events. Subsequently, they introduced *Basket ACID*, where concurrent access to the contents of each basket is regulated via a locking scheme or the scheduler. Similarly, Meehan et al. [78] logically view tuples with the same batch-ID as occurring concurrently as a group, which should be processed as an atomic unit. Sailesh et al. [68] define the unit of work (or a transaction) as the processing of one or more chunks of data called "runs", where a "run" is a finite sub-part of a stream (or a group of data) simultaneously collected by an SPE (in this case DataCell [50]). Implicitly, DataCell assumes tuples that arrive simultaneously are grouped into the same batch, which triggers a transaction to be processed.

Chen et al. [44, 45] propose the cycle-based transaction model to support CQCP (continuous querying with continuous persisting) with cycle-based isolation and visibility. They convert infinite stream data into a sequence of "chunks" and execute queries over each chunk sequentially. In this form, a stream query is committed one cycle at a time, where a cycle can be a batch of input events.

S-Store [78] implements stream transactions as stored procedures and represents stream dataflow graphs as lists of stored procedures. When a batch of input tuples arrives, the first stored procedure is triggered, which may trigger subsequent stored procedures. Punctuation or control tuples can also be used to set a transaction boundary [56].

Single Event-based Transactions. The Aurora and Borealis [12, 13] stream processing engines provide operators that perform selections, insertions, deletions, and updates on Berkeley DB tables for each new input stream tuple. Brito et al. [33] utilize software transactional memory to correctly parallelize order-sensitive stream operations. Sturzrehm et al. [110] extend their original approach to a distributed environment. They model the processing of each input (called a task) as one transaction with a pre-assigned timestamp as its commit order. Specifically, as an event enters the system, it will be assigned a logical timestamp, which is unique and continuous (i.e., there are no gaps in time). To ensure this assumption is met, whenever an event is discarded (e.g., a filter drops an irrelevant event) a *null* event is inserted to carry the timestamp through the system. Wang et al. [125] defines a stream transaction as a sequence of ACEP (active complex event processing) system state changes triggered by a single input event. Ray et al. [95] propose to employ stream transactions to ensure concurrent shared maintenance and the re-use of sub-patterns across queries. Zhang et al. [134] define the set of state accesses triggered by the processing of a single input event at one executor as one *state transaction*. The timestamp of a state transaction is the same as its triggering event.

User-defined Transactions. Botan et al. [31] define a transaction in rather flexible terms. By modelling streaming primitives as reading/writing operations on a transactional database, they can reuse the transaction processing capabilities from existing transactional databases (i.e., H-Store in this case) very easily. However, performing transactions on data streams raises the issue of setting transactional boundaries, which may not obvious. Consequently, they propose to allow users to set their own transaction boundary [31]. Nevertheless, all of their examples assume that a transaction is triggered by one input event and includes the subsequent one-time continuous query execution. Another example is discussed by Chen et al. [43], where TSP may be applied to handle conventional OLTP workloads or support ad-hoc transactional queries during stream processing. In such a case, a transaction in TSP is directly posed by client applications in the same manner as conventional database systems.

3.2.2 Generating Unit

Unlike triggering units which determine "when" a transaction is created, generating units determine "who" generates a transaction. In particular, transactions are generated via user clients directly or via continuous queries on a per-query or per-operator basis. There are varying types of generating units: query-based, operator-based, and user-defined. We examine each of these types in turn.

Query-based Transactions. Via a query-based transaction generation scheme, early SPEs enabled the interactive processing of relational data and stream data. In particular, operations involved in the one-time execution of the entire query were grouped into a single transaction. For example, STREAM [20, 57] models the execution of an entire query on a batch of tuples with a common timestamp as a single transaction. In the case of Golab and Ozsu [53], a snapshot query or a particular re-execution of one or more long-running queries are read-only transactions that reference sliding windows.

Operator-based Transactions. Another design approach is to allow each operator of a query to generate transactions. For example, S-Store [78] models a transaction as a single invocation of a stored procedure, and each stored procedure represents one operator in a streaming dataflow graph. Similarly, Zhang et al. [134] adopt a transactional dataflow model and prescribe a transaction generating unit in a much more fine-grained manner, where each operator in a streaming dataflow graph can generate a transaction that can read or write to the shared states. Operator-based generating units are far more flexible than query-based generating units. However, they can lead to a *dilemma*, which will be discussed in Section 4.

User-defined Transactions. Some applications require the handling of ad-hoc transactional queries during stream processing. These user-driven ad-hoc queries may read or update states that are shared with continuous queries. Consequently, in order to resolve any conflicts, access must be managed transactionally. Another example arises when client applications (i.e., users) create transactions that are handled by TSPSs, which are employed to handle conventional OLTP workloads [43].

In the case of Affetti et al. [14, 15], they extend the dataflow model of an SPE by introducing the concept of a transactional subgraph (i.e., a *t-graph*), which identifies a portion of the graph of computation where the state of enclosed operators is accessed and updated with transactional semantics. They also enable users to specify *where* consistency needs to be enforced, and *which* consistency constraints are required. In particular, each tuple that enters the *t-graph* initiates a read-write transaction. All of the operators within the *t-graph* are processed as a single transaction with ACID guarantees. The state of operators within the *t-graph* is also externally queryable through read-only transactions. A drawback of the user-defined approach is the difficulty in performing optimizations, given that the type of transaction is unknown to the system in advance. Moreover, users are responsible to ensure that the system is free of any dilemmas.

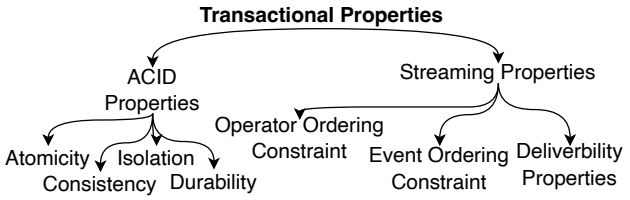


Fig. 2: Transactional Properties in TSP.

3.2.3 Transaction Spawning

Transactions can spawn, i.e., trigger and generate other transactions. This is particularly useful in service-oriented architectures, where the basic premise is to treat all functionalities as services and compose and execute them, according to the user or application-specific requirements [123]. Treating each service execution as a transaction and requiring atomic execution of those transactions have been found to be very helpful in this process. The execution of service compositions yields composite transactions [122], where a transaction is an execution of a service [121]. Subsequently, a transaction spawning consists of a nonempty set of services. Some of which have executions that are continuous and others may spawn new transactions.

3.3 Remarks Concerning Transaction Models over Streams

TSP requires the convergence of both stream and transactional processing. However, there are many different ways for this convergence to occur. This results in transaction models that are highly flexible. To date, there is no single transaction model that is widely accepted for TSP, which only increases the difficulty in comparing the performance of the varying TSPSs.

4 Transactional Properties in TSP

In the research literature, transactional properties greatly vary. We examine each of these properties in turn.

4.1 ACID Properties

Traditionally, database transactions are encapsulated sequences of (read, write)-operations that must be ACID compliant [27]. TSPSs also offers the same ACID guarantees via the OLTP model. For example, in S-Store [78] all transactions are predefined and represented as stored procedures. In addition, in S-Store, stream transactions are transformed into database transactions, which are then handled in H-Store [108] – a relational database system.

However, several studies [14, 31, 125, 134] support the idea that the concurrent execution of multiple state transactions should satisfy a variation of ACID properties for two key reasons. First, in relational databases ACID properties are strict requirements. However, these properties can be relaxed in other contexts, depending on the required semantics. Second, in some streaming applications, the traditional ACID properties do not hold. For example, Wang et al. [125] propose a mapping of the classical ACID properties to stream-ACID properties (s-ACID) for active complex event processing.

Unfortunately, due to the lack of a common standard, prior works on TSP typically focus on only a narrow set of application workloads. In addition, the notion of a transaction in the context of stream processing varies, and often provides an informal guarantee of correctness that is tightly coupled with the operational semantics given by a specific system's implementation. Next, we discuss the custom ACID properties in TSPSs: atomicity, consistency, isolation, and durability.

4.1.1 Atomicity

In relational databases, atomicity is a transaction property that guarantees that a database is either fully updated or not updated at all [124]. An atomic transaction is an indivisible and irreducible series of database operations that either all occur or none occur. This idea of atomicity in relational databases carries over to TSPSs. For example, the unit of atomicity in Gürgen et al. [59] is a set of "continuous queries that read updatable resources". In the case of Streaming Ledger [9] either all of the row modifications are performed (i.e., the transaction succeeds), or none are performed (i.e., the transaction fails).

In contrast to the conventional definition of atomicity, Wang et al. [125] propose that all operations triggered by a single input event should be atomic in their entirety. In S-Store [78] atomic batches of input events are executed in isolation. Hence, atomicity is concerned with the batch of input events rather than the execution of a single transaction. TSpoon [15] ensures that all the effects within a transactional subgraph (triggered by an input event) are either stored entirely or not at all.

4.1.2 Consistency

Consistency in database systems refers to the requirement that any given database transaction must not leave the database in an invalid state, according to defined rules, including constraints, cascades, and triggers [124]. For example, a user's age should not become negative. Similarly, consistency matters in TSPSs as well. However, few works discuss consistency in TSP due to its similarity

to conventional database systems. In the case of Wang et al. [125], the execution of stream transactions must offer a guarantee that the ACEP system will start in a correct state and when it ends leave the ACEP system state correct. TSpoon [15] enables developers to specify integrity constraints on the value of individual keys in their proposed transactional subgraph region (i.e., the t-graph). They ensure that the state in a t-graph is always consistent – before and after a successfully committed transaction makes modifications to the states kept in the t-graph. Similarly, Streaming Ledger [9] ensures that tables are always consistent.

4.1.3 Isolation

Isolation regulates the interaction between concurrently executed transactions that read and write common keys. Stricter isolation levels constrain the interaction between transactions and offer higher guarantees, but at the cost of lowering the degree of concurrency and decreasing performance. Conversely, more relaxed isolation levels impose fewer constraints, enable a higher degree of concurrency and increase performance. Akin to databases, TSPSs also offers a number of isolation levels, which control the degree of locking that occurs when selecting data from shared states [15].

Serializable Isolation. *Serializable isolation* is the highest isolation level that ensures transactions are executed as if they were executed sequentially. In Wang et al. [125], isolation holds when a single input event that must appear to be executed as if no other input events are being processed concurrently triggers a change to the ACEP system state. Although not explicitly mentioned, serializable isolation is guaranteed due to the usage of locking-based concurrency control protocols, which will be discussed further in Section 5.3. FlowDB [14] also supports serializable isolation. It ensures that the results of two transactions take place as if they were executed in some sequential order, without interleaving updates to state operations. Similarly, in Streaming Ledger [9], each transaction executes as if it were the only transaction being performed on a table.

Snapshot Isolation. A number of TSPSs were built on the notion of snapshot isolation [12, 31, 44, 46, 56], a relaxed isolation property common in conventional transaction processing. Snapshot isolation guarantees that all reads performed in the same transaction have access to a consistent snapshot of the database, and the transaction itself will only commit if no updates conflict with any concurrent updates made since the snapshot. In practice, each transaction can access the last committed values in the database that existed at the time the transaction began.

Eventual Consistency. Eventual consistency [25] is a consistency model used in distributed computing to achieve high availability. It informally guarantees that if no new updates are made to a given data item, eventually all accesses to that item will return the last updated value. SPEs usually adopt some form of eventual consistency since states exposed to the outside world are expressed as output streams, and instant consistency of the global system state is hidden from users. However, it is unclear how the eventual consistency model can be applied to TSPSs, given that shared mutable states (or their snapshot) need to be immediately visible and queryable both internally and externally to the system.

4.1.4 Durability

In database systems, durability guarantees that committed transactions will survive permanently even in the presence of a system failure. In TSP, in order to enable fault tolerance, there are three key durability properties, which we discuss next.

Durable Input Streams. *Archived streams* are streams that have been written to durable mediums and can subsequently be accessed by queries. For example, they are used by TSPSs to perform correlations between historical and current data [40]. As new stream tuples arrive, TSPSs update the results of continuous queries, and then periodically archive the tuples for later access.

Durable States. Most TSPSs also require that intermediate changes made during stream processing or application states be durable (persistent). This is generally known as "fault-tolerance" in SPEs. It is one of the core features that distributed SPEs should provide.

Durable Output Results. Lastly, the output results in TSPSs must also be durable (permanent). Wang et al. [125] state that (stream) durability is the property that the output of pattern queries must be "permanently valid". That is, at any given point in time, all of the output events from the ACEP system up until that point satisfy the query semantics. Similarly, Botan et al. [31] consider durability in the context of the input events of committed transactions that will never be reprocessed or duplicated.

Achieving durability for input streams, states, and output results simultaneously in a system is often unnecessary as fault tolerance can be achieved with any one of the three durability properties. However, the run-time overhead and the recovery delays will vary accordingly. To the best of our knowledge, there is still no in-depth study on the design of efficient fault tolerance mechanisms for TSPSs. Fault tolerance mechanisms will be discussed in Section 5.4.

4.2 Streaming Properties

Besides the ACID properties, TSPSs also guarantees streaming properties, including an operator ordering constraint, an event ordering constraint, and deliverability properties. Next, we discuss each of these in turn.

4.2.1 Operator Ordering Constraint

As previously discussed in Section 3.2.1, TSPSs can represent an application as a directed acyclic graph (DAG). For each input event, the processing of each operator (i.e., a vertex) can trigger a stream transaction. Naturally, this results in a topological ordering of the transactions. In S-Store [78], the scheduling of triggered transactions must satisfy one of the topological orderings. Although there may be many correct schedules corresponding to multiple topological orderings of the DAG, S-Store currently allows only one of them. In contrast, several other TSPSs naturally satisfy an operator ordering constraint due to their dataflow-based execution model [14, 134].

4.2.2 Event Ordering Constraint

Besides the operator ordering constraint, TSPSs must also ensure that the resulting transaction schedule order is aligned with the timestamps corresponding to the sequence of streaming events [31, 78]. For example, a read operation triggered by an event must never be able to access an updated state that is modified by a write operation triggered by a "future" event. This is critical as streaming events are chronologically ordered. Golab et al. [53] were the first to point out that serializability is insufficient for the correct concurrent processing of sliding windows. A window is split into multiple sub-windows to allow parallel processing. A conflict occurs when two interleaved transactions operate on the same sub-window with at least one write operation. Alternative conflict-serializable schedules can lead to different stream processing results. The key reason for this is that conventional serializability does not have a notion of time. However, stateful stream processing is ordering sensitive – application states referenced during stream processing change over time.

Golab et al. [53] 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 when all sub-windows of the window are updated. The second is called *latest-window-serializable*, which allows a read, only on the latest version of the window, i.e., after the window has been completely updated. Instead of always enforcing ordering constraints, FlowDB [14] enables developers to optionally ensure that the effects of transactions are the same as if they were executed sequentially, in the same order in which they started.

4.2.3 Deliverability Property

In addition to durability, TSPSs strive to provide deliverability guarantees for all streams, akin to SPEs. However, TSPSs requires a deliverability guarantee on both the local state of an operator and the strict time ordering of an input stream for the corresponding operation. This is because the results are dependent on the local state of an operator as well as the time ordering of the input streams. Hence, this model is more restrictive than the *exactly-once* guarantee found in many SPEs.

TSPSs needs to replay failed tuples in the exact timestamp sequence of their triggering input events and avoid the processing of duplicate messages. Thus, TSPSs ensures that a new worker will start exactly from the point when a failed worker stopped. Thereby giving the impression that a failure never occurred. One way to achieve this is to checkpoint each message before processing and replay them in the event of failure. Due to its significant overhead, most TSPSs do not provide such a strict deliverability guarantee. Therefore, additional investigation is warranted, to identify a more efficient mechanism.

4.3 Relaxed Transactional Properties

Transaction models with relaxed properties, e,g., *sagas* [51] and ConTract [130] have been proposed. The *sagas* model [51] allows a transaction to be split into several smaller sub-transactions. Thus, isolation is relaxed in the original transaction and delegated to the individual sub-transactions. The atomicity of the original transaction is preserved by ensuring all sub-transactions are executed successfully or none at all. A similar idea of splitting transactions has been adopted in TSPSs such as TStream [134]. The ConTract model [130] was proposed as a mechanism for grouping transactions into a multitransaction activity. Transactions are made up of multiple steps, with explicit dependency relationships specified between the steps. The system ensures that such dependencies hold when the steps execute. In addition to the relaxed isolation, ConTracts provide relaxed atomicity so that a ConTract may be interrupted and re-instantiated. In the case of a failure, the state of the ConTract must be restored and its execution may continue. Due to its advantage for fault tolerance, the contract model may be adopted in TSPSs, but we are unaware of any related works.

Vidyasankar et al. [121] argue that strict ACID and streaming properties are inappropriate for TSP in the emerging IoT environment, since triggers and triggered transactions may be executed in distinct autonomous sites

"far away" from one another. Consequently, they distinguish between triggers and triggered transactions by atomic unit. They also introduced the term *weak atomicity* for composite service executions. These relaxed atomicity and isolation properties can be useful for the IoT. For example, the healthcare domain may require stronger atomicity, whereas comfortable homes and offices may tolerate weaker atomicity. Later, Vidyasankar et al. [122] further relaxed the properties to include partial orderings and completions.

4.4 Remarks concerning Transactional Properties in TSP

The transactional properties in TSP are far more flexible and diverse than the conventional transaction processing paradigm, where an agreed standard has been established. We argue that it is not wise to devise the same standard properties for TSP, given that there is great diversity in stream application requirements. That said, it is still desirable to devise a standard that summarizes both the common and representative properties of a future system or application that can later referenced. We hope this survey will inspire the development of such a standard for TSP. In the following, we discuss two important remarks concerning transactional properties in TSPSs.

Remark 1 (Failure of Concurrency Control Protocols) Conventional concurrency control (CC) protocols, which are widely used in OLTP database systems, fail to guarantee the transactional properties of TSPSs. To illustrate why, we use a conventional timestamp-ordering concurrency control (T/O CC) [28] as an example. Similar discussions can be also found in a prior work [125]. Let $txn_1 = write(k1, v1)$ and $txn_2 = read(k1)$ be two distinct transactions. For simplicity, let us assume that there are only these two transactions in the system. If $txn_2.ts > txn_1.ts$, then both txn will be successfully committed. However, their serial order would be $txn_2 \rightarrow txn_1$, which violates the event order constraint (i.e., txn_2 will wrongly read the original value of $k1$). On the other hand, if $txn_2.ts < txn_1.ts$, then txn_2 will be successfully committed. However, txn_1 will be aborted as the writes will come too late. Aborting a transaction that represents an undo of an externally visible output or action may not be acceptable in TSP applications. Similarly, in other conventional CC protocols, either the results are in the wrong serial order or one of transactions has to be aborted, which eventually will result in the wrong serial order upon a restart. In other words, conventional CC protocols are not yet ready for such *event-driven* transaction execution.

Remark 2 (Timestamp Assignment Dilemma) Naturally, one could assume that a transaction triggered by a corresponding input event can instantaneously take effect once the event occurs [125]. In this setting, the timestamp of the transaction

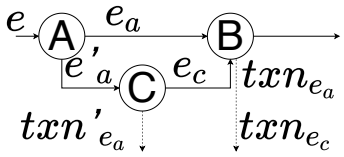


Fig. 3: Example of timestamp assignment dilemma.

is then set to be the same as the timestamp of its corresponding triggering event. When all transactions are generated by external events, the transaction schedule is aligned with an external event sequence, to satisfy the event ordering constraint. However, a dilemma arises when transactions can also be generated by internal events (i.e., outputs generated by operators) with timestamps assigned according to input events, as illustrated in Figure 3. In the DAG each operator can produce output streams and generate stream transactions. Suppose operator A receives input event e and generates two events e_a and e'_a, that are passed to operators B and C, respectively. Further, suppose operator C processes e_a, generates e_c as an output, and then passes e_c to operator B. Now, let us assume operator B processes two events e_a and e_c and generates two transactions txn_{e_a} and txn_{e_c}, respectively. In addition, operator C processes event e'_a and generates transaction txn'_{e_a}. On the one hand, txn_{e_a} must be committed jointly with txn_{e_c}, for otherwise the event ordering constraint would be violated. On the other hand, txn_{e_c} cannot be generated before txn'_{e_a} committed, for otherwise this would violate the operator ordering constraint, which means txn_{e_a} is unable to be committed. The system then runs into a deadlock situation, where txn_{e_a} and txn'_{e_a} are infinitely waiting for each other to be committed. There are two approaches to prevent this dilemma, 1) an additional ordering constraint can be enforced between operators B and C, or 2) different timestamps could be assigned to the generated events: e_a, e'_a, and e_c. However, to date, we are not aware of any general efficient solution to address such a dilemma.

5 Execution Mechanisms in TSPSs

The execution mechanisms in today's TSPSs serve to ensure that transactional properties are met. In particular, *atomicity* is achieved via logging (Section 5.1), *consistency* is attained via integrity constraints (Section 5.2), *isolation* (Section 5.3) is enabled via concurrency control, and *durability* (Section 5.4) is accomplished via fault tolerance. Moreover, the streaming-specific properties (e.g., operator ordering constraint, event ordering constraint, deliverability) are also met by each of the aforementioned mechanisms. For example, fault tolerance not only preserves durability but also provides deliverability and satisfies the two

ordering constraints. Next, we discuss each of the execution mechanisms: logging, integrity constraints, concurrency control, and fault tolerance.

5.1 Logging

TSP applications need to ensure atomicity when updating shared states. Barriers to achieving atomicity, include system failures or when user-defined transactions abort during processing. Most prior works on TSP either do not mention their logging mechanism [14, 134] or they rely on the logging mechanism provided by their storage systems (e.g., traditional database systems [78]).

5.2 Integrity Constraints

Typically, applications prescribe the integrity constraints that are to be imposed, such as a foreign key constraint on a relation, or a set of business rules. For example, Meehan et al. [77] discuss the stream ETL scenario, where TSPSs need to preserve foreign key constraints among tables.

In relational database management systems, integrity constraints arise during the design of the relational schema and are usually specified in SQL. Via this approach, the specification and enforcement of integrity constraints are tightly bound to the relational model. Some TSPSs are built on extensions to traditional database systems (e.g., S-Store [78]), which naturally offer support for integrity constraints. However, since many TSPSs (e.g., TStream [134]) are not or not only based on relational algebra or SQL, they simply do not offer support for integrity constraints. Thus, to ensure data integrity the burden falls on the application developer.

5.3 Concurrency Control

Concurrency control (CC) is essential, to ensure correctness in systems where interactions between concurrent accesses and updates are prevalent. In particular, CC is related to the *isolation* property. In the context of TSP, various CC mechanisms have been proposed to ensure a correct schedule for concurrent state transactions. These can be classified into five approach types: *single-version lock-based*, *multi-version lock-based*, *static partition-based*, *dynamic partition-based*, and *optimistic*. Next, we discuss each of these approaches.

5.3.1 Single-version Lock-based Approach

In STREAM [57], *synopses* allow different operators to share common states. To guarantee operators view the correct version of a state, the system needs to track the progress of each *stub* and present the appropriate view (i.e., a subset of tuples) to each of the *stubs*. This is achieved via its local-timestamp-based execution model with a global schedule that coordinates the successive execution of the individual operators via time slot assignments. Batches of tuples with the same timestamp are executed atomically to ensure progress correctness, and a simple lock-based transactional processing mechanism is implicitly involved.

An earlier study by Wang et al. [125] 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 [27] algorithm, Wang et al. [125] *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. In this process, once lock insertion is complete, the system will increase the counter, and then allow the next transaction to proceed irrespective of whether the transaction was fully processed. To fulfil the event ordering constraints, read or write locks are strictly invoked in their triggering event order. However, the *locking* mechanism has to synchronize the execution for every single input event, which may negatively impact system performance.

Oyamada et al. [87] contribute 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 and the execution results are naturally generated following the event arrival sequence. ATSI removes the blocking behaviour of STSI by asynchronously spawning new threads that wait for the transaction to complete. OPATSI extends ATSI via a priority queue to further guarantee the order of the results.

FlowDB/TSpoon [14, 15] implement a single-version lock-based mechanism and work at a more fine-grained granularity of the single "key, value" pair stored by a state operator. When an ordering constraint is required, they introduce an additional sequencer component before each stateful operator in a stream query graph. This consists of a single instance and reorders transactions by ID.

5.3.2 Multi-Version Lock-based Approach

Wang et al. [125] 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

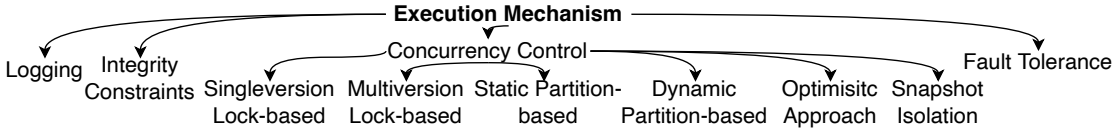


Fig. 4: Execution and fault tolerance artifacts of transactional stream processing.

must be performed monotonically in event order, but read operations are allowed to execute as long as they are able to 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 two-fold. First, MVCC aborts and then restarts a transaction when an outdated write occurs. In contrast, LWM ensures that writes are permitted strictly in their timestamp sequence, thereby 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.

5.3.3 State Partition-based Approach

S-Store [78] splits the streaming applications internal states into multiple disjoint partitions. The computation on each sub-partition is performed by a single thread. To guarantee state consistency, S-Store uses partition-level locks to synchronize access. However, the state partition-based approach 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.

5.3.4 Transaction Partition-based Approach

TStream [134] is a recently proposed TSP system that adopts transaction decomposition to improve stream transaction processing performance on modern multicore processors. Despite the relaxed isolation properties, TStream ensures serializability, as all conflict operations (being decomposed from the original transactions) are executed sequentially as determined by the event sequence. TStream provides a novel no-lock two-phase execution approach to guarantee the correctness of transactional state accesses. Furthermore, two modules are implemented in TStream, thereby eliminating cross-process communication overhead. However, TStream only runs on a single node.

5.3.5 Optimistic Approach

Targeting *window serializable* properties, Golab et al. [53] propose a scheduler that executes window movements optimistically and uses serialization graph testing (SGT) to abort any *read-only* transactions that cause a read-write conflict. The schedule is conflict-serializable if and only if the precedence graph is acyclic. They further propose to re-order the read operations within transactions to reduce the number of aborted transactions and thereby yield a better schedule. FlowDB/TSpoon [14, 15] also introduce an optimistic timestamp-based protocol. These systems do not lock resources, but rather use timestamps to ensure that transactions always read/update versions that are consistent with the desired isolation level. When this is not possible, transactions are aborted and rescheduled for execution.

5.3.6 Snapshot Isolation Approach

A number of TSPSs employ snapshot isolation [12, 31, 44, 46, 56] with the goal to split a stream into a sequence of bounded chunks and apply the semantics of a database transaction to each chunk, i.e., putting the operation on a data chunk in a transaction boundary to yield a state snapshot. In this way, processing a sequence of data chunks generates a sequence of state snapshots. Götze and Sattler [56] present a snapshot isolation approach for TSP. Each state has multiple versions of values 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.

5.4 Fault Tolerance

Failure events in SPEs can cause an application to block or produce erroneous results. Streaming applications typically run for indefinite periods, which increases the chance of service disruptions due to unexpected system failures. Hence, a large number of fault tolerance approaches have been proposed for SPEs. These approaches often have their own particular performance priorities (e.g., runtime overhead, recovery efficiency). For a general discussion on fault tolerance mechanisms in SPEs, readers should reference an earlier survey [116].

Although modern SPEs usually offer fault-tolerance mechanisms, they do not always satisfy the fault tolerance requirements of TSP. In the event of a failure, TSPSs require all of their states to be recovered (including the input/output streams, operator states, and shared mutable states), such that any committed transactions remain stable. In addition, any uncommitted transactions are not permitted to affect this

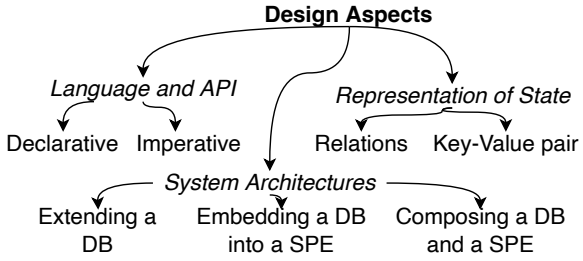


Fig. 5: Design Aspects of TSPSs.

state. A transaction that has begun, but has not yet been committed should be undone and reinvoked with the proper input parameters, once the system is stable again. This requires an upstream backup and an undo/redo mechanism similar to an ACID-compliant database. Furthermore, to satisfy streaming properties, the recovered states should be equivalent to the one that was under construction when there was no failure. To achieve this, an order-aware recovery mechanism is required [105]. However, the widely adopted recovery operation in modern SPEs, especially, the parallel recovery operation, might lead to different transactional states as there are no guarantees on the event processing sequence during recovery.

5.5 Remarks Concerning Execution Mechanisms in TSP

To create *a hybrid system*, where stream queries can refer to both 1) static data stored in a database and 2) stream analysis results (whether intermediate or final), an execution mechanism is needed to allow stream processing to periodically "commit" results and make them visible. However, designing an efficient execution mechanism for TSP is difficult as it must satisfy not only the ACID properties but also the additional three streaming properties. Existing works often implicitly overlook certain properties in their execution mechanism design, to improve performance. This should be made more transparent so that users can be more aware of which properties are not guaranteed in adopting a TSPSs. Unfortunately, as there is currently no standard definition of TSP, it is difficult for the community to reach a consensus.

6 Design Aspects of TSP Systems

In this section, we discuss three major design aspects of TSP systems: *languages and APIs*, *system architectures*, and *representations of state*.

6.1 Languages and APIs

Stream processing languages facilitate the development of stream processing applications. They simplify common coding tasks and make code more readable and maintainable. Additionally, their compilers are able to catch programming errors and optimize code transformations. However, unlike relational databases, where SQL is the standard programming language with numerous SQL-based APIs, it is not yet clear what the APIs for TSPSs should be, largely due to the absence of a standard transaction model. Next, we turn our attention to existing languages and APIs for TSPSs as well as some of the representative APIs designed for SPEs that may be applied to TSPSs.

6.1.1 Declarative Languages

Almost every attempt to create a standard programming language for streams has been an extension of SQL [6,20,50, 57,81]. For example, the well-known STREAM system [20, 57, 81] supports a declarative query language called CQL (Continuous Query Language), which is designed to handle both relational data and data streams. In CQL, the traditional *from* clause in SQL is defined for both relations and streams. In addition, in CQL, streams and time-varying relations are accepted as inputs and treated uniformly as relations. Moreover, CQL incorporates the notion of time: converting streams into relations via sliding window operators and converting relations into streams via three operators,i.e., Istream, Dstream, and Rstream. Istream and Dstream convert a relation to stream whenever there is an insertion or deletion, respectively. The Rstream operator maintains the entire current state of its input relation and outputs all of the tuples as insertions at each time step.

Like STREAM, Coral8's [6] continuous computation language (CCL), which has a SQL-like syntax, supports both data streams and event streams. In addition, CCL offers support for continuous queries as well as primitive/composite events with temporal and windowing capabilities over streams with access to persistent data sources. Due to Franklin et al. [50], TruSQL is yet another relational stream query language. In TruSQL, stream processing is fully integrated into SQL, including persistence. Moreover, in TruSQL, the notion of a stream (i.e., an ordered unbounded relation) has been added to the standard relational model.

The aforementioned relational stream query languages are well-defined, semantically precise, and well-suited for TSP, especially when handling a single query at a time. However, they are unable to correctly express the interaction among queries, which can lead to ambiguous execution results. The original paper [21] expresses the linear road

benchmark as a single query with multiple interdependent subqueries, which cannot easily be parallelized.

Potentially, there are two ways to address this issue. One is to explicitly express the proper interaction among queries (via stored procedures), which would require the language to be a mixture of declarative and imperative styles. The other is to express the interaction constraints as separate rules (like triggers). To better understand the trade-offs between the two approaches, further investigation is warranted.

6.1.2 Imperative Languages

Although a SQL-like declarative language provides a succinct and simple solution to many streaming problems, the operational flow of an imperative language [17] is more apt to express state abstractions and complex application logic.

In Aurora [13], query plans are constructed via a graphical interface by arranging boxes (representing query operators) and joining them with directed arcs to specify a uniform dataflow. Among the operators in Aurora, three are noteworthy: *map*, *resample*, and *drop*. Arising as a second-order function in functional programming languages, and popularized in the MapReduce programming model (for scalable data processing), the map operator applies a user-defined function to each input item. The resample operator interpolates the values of missing items within a window, whereas the drop operator randomly drops items whenever the input rate is too high. Like Aurora, STREAM allows the direct input of query plans expressed as a dataflow.

Influenced by MapReduce-like APIs, the majority of recent open-source streaming systems, such as Flink, embed functional/fluent APIs into general-purpose programming languages, to hard code Aurora-like dataflows. This design is also inherent in current TSPSs. For example, FlowDBMS [14, 15] extends the Flink data stream API and follows an imperative language design to support TSP. In FlowDBMS users must explicitly declare shared mutable states and use two primitives, *openTransaction* and *closeTransaction* to define the boundaries of transactional subgraphs. TStream [134] is yet another example of a TSPSs with imperative APIs that provide a list of user-implemented and system-provided APIs within each operator – following the Aurora approach. User-implemented APIs draw on application-specific requirements, whereas system-provided APIs draw on system libraries. When employing a user-implemented API to express stream transactions, users can invoke system-provided APIs to gain access to shared states.

6.2 System Architectures

From the aforementioned application use cases, we observe that TSP applications typically require SPEs to be used in combination with data storage and analysis frameworks, such as a database management system (DBMS) [8] to build software architectures that combine data storage, retrieval, and mining [14]. To meet this goal efficiently in TSPSs, there are three approaches to form a TSP system: 1) extending a DBMS, 2) embedding a DBMS in an SPE, and 3) composing a DBMS and an SPE. Next, we dive into each of these approaches.

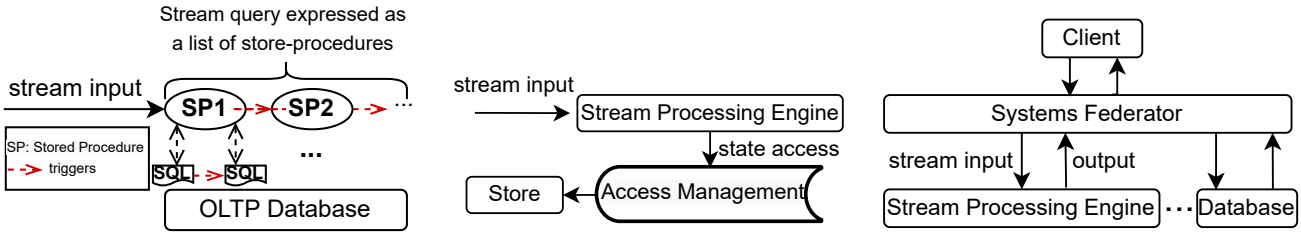
6.2.1 Extending a DBMS

To provide support for TSP, one option is to extend an existing database system, as depicted in Figure 6a. Next, we describe several systems that employ this approach, including *DataCell*, *TruCQ*, *MaxDBMS*, and *S-Store*.

As early as 2009, Liarou and Kersten [74] criticized SPEs for lacking the power of fully-fledged database systems. To address this problem, they proposed DataCell, an SPE built on top of a modern database kernel that immediately stores stream tuples into specially designed tables called *baskets*. Based on DataCell, continuous queries are evaluated as if they were conventional one-time queries. Similarly, the Truviso Continuous Analytics system [50], later known as TruCQ (a commercial product), also follows the DBMS extension approach. In this case, the open source PostgreSQL DBMS [109] was extended to enable the continuous analysis of streaming data and tackle the problem of low latency query evaluation over massive data volumes. By integrating traditional relational query processing and streaming, TruCQ employs a stream-relational database architecture, which runs SQL queries continuously and incrementally over incoming data. Furthermore, TruCQs query processing significantly outperforms traditional store-first-query-later database technologies, given that query evaluation has already been initiated upon the arrival of the first tuples. Moreover, TruCQ allows the evaluation of one-time queries, continuous queries, and combinations of both.

In order to integrate heterogeneous SPEs, DBMSMSs, and storage devices, Botan et al. [30] developed MaxStream, an SPE that extends MaxDBMS [4], a traditional DBMSMS. As the last example, S-Store (a recent TSPSs) due to Meehan et al. [78] leverages the existing transaction processing capabilities in H-Store [63] – a distributed OLTP-DBMS. In S-Store, a stream query is a list of stored procedures (each of which can be viewed as a stream operator), streams and windows are represented as time-varying tables, and triggers enable push-based processing over streams and windows.

In S-Store, triggers are associated with either a stream table or a window table. When new tuples are appended to a table, downstream processing is automatically activated. A limitation of this approach is that it cannot fully support



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

Fig. 6: Alternative system architectures for TSPSs.

native or hybrid stream processing applications efficiently, particularly those that do not or only partially require ACID properties (e.g., when only a subset of operators require transactional support). Furthermore, stream applications are usually expressed as a DAG of operators for which there is at least one topological ordering among operators. However, the current triggering-based design solely permits a single topological sort [78], which limits optimization opportunities.

There are two fundamental issues involved in this architectural design. First, the existing store-first-query-later approach although suitable for large databases is ill-suited for stream processing applications. Thus, most SPEs employ a query-on-the-fly approach [106] instead. Second, supporting stream-native operations (e.g., windowing operations) in databases is nontrivial. As a result, implementing stream processing on top of existing relational databases limits the performance. Drawing on the linear road benchmark, Chen et al. [43] note that an SPE can achieve an order of magnitude higher throughput in comparison to Botan et al. [30] who propose extending a DBMS to an SPE.

6.2.2 Embedding a DBMS in an SPE

To provide support for stream processing, another option is to embed a database system in an SPE. As illustrated in Figure 6b, a relational DBMS (depicted as a component that stores shared-mutable states) is embedded in an SPE. This enables an SPE to support ordinary stream applications while also providing the transactional facilities. Due to TSP requirements, such as streaming properties that are not provided in conventional database systems [37], an additional *access management* component must be provided, to link a storage component to an SPE.

Aurora [13] and its successor Borealis [12] both use a popular embedded key-value store, Berkeley DBMS [85] for relational data storage. In these systems, incoming data stream tuples can trigger queries in Berkeley DBMS. Drawing on a model that integrates data management capabilities and stream processing, Affetti et al. [14, 15] propose embedding a key-value store to manage shared mutable states in Flink. Similarly, TStream [134] adopts a modular design that includes two modules. One is a stream module that is based on BriskStream [104], a highly optimized general-purpose SPE with an architecture similar to Storm. The other is a state module based on the Cavalia [129] database, that manages state access via system-provided APIs.

6.2.3 Composing a DBMS and an SPE

To provide support for stream processing, a third option is to compose a DBMS and an SPE, as depicted in Figure 6c. Although feasible, this approach is comparably more complex than the other two approaches. In particular, it imposes an additional burden on developers: the difficult task of writing application-specific code to ensure integration correctness. As a consequence, this approach hinders the design, implementation, maintenance, and evolution of the solution.

In 2009, Botan et al. [29] proposed to decouple storage management from stream processing. As a consequence, they developed the Storage Manager for Streams (SMS), which provides *flexibility* to replace/upgrade the system component, *adaptability* to specific requirements, and *optimizability* for tunable parameters (e.g., deciding whether or not to share state among concurrent stream operators). In 2012, Botan et al. [31] introduced additional concurrency control mechanisms, to refine their initial idea and offer support for transactional updates.

6.3 Representations of State

State can be represented as either *relations* or *key-value pairs*. Below we discuss each representation in turn.

6.3.1 Relations

There are stream processing systems that represent states as relations. Representative examples include STREAM, S-Store, and TStream.

In STREAM [57], state is represented as a time-varying relation that maps a time domain to a finite, but the unbounded bag of tuples adhering to the relational schema. In order to treat relational and streaming data uniformly, there are two operations: *To_Table* to convert streaming data to relational data, and *To_Stream* to convert relational data to streaming data. Several implicit assumptions are made about time-varying relations, such as all stream elements and related updates are timestamped according to a global clock. Unfortunately, this is often unrealistic, particularly for distributed stream processing.

Similarly, S-Store [78] does not implement its own state management component. Instead, shared states represented as relations are stored in H-Store [108] and S-Store relies on H-Store to ensure the transactional properties of shared states. Lastly, TStream [134] relies on the Cavalia relational database [128] to support the storage of shared states. Such an approach has the advantage of reusing well-developed techniques in relational databases, such as persistence and recovery mechanisms. However, unless we assume the state is constant, we must introduce the notion of time to the relation given that stream processing often relies on both time and the sequence of inputs to make progress. This is something that is not naturally captured in relational models.

6.3.2 Key-Value Pairs

There are also systems that represent states as key-value pairs. Representative examples include MillWheel [17], Flink with RocksDBMS, AIM (Analytics in Motion) [32], and FlowDBMS/TSpoon [14, 15].

In MillWheel [17], state is represented as an opaque byte string that is managed on a per-key basis. In order to store or transmit these byte strings, users need to implement serialization and deserialization methods. To ensure data integrity is completely transparent to the end user, the persistent state is backed by a replicated and highly available data store, such as Bigtable [42] or Spanner [48].

To support shared queryable state, Flink [36] relies on an LSM-based key-value store engine called RocksDBMS [5]. Likewise, Götze and Sattler [56] adopt a key-value store for transactional state representation. In particular, they also adopt multi-version concurrency control, where each state (i.e., key) has multiple *commit timestamps*, *delete timestamps*, or *values*.

In the case of AIM [32], state is represented in a distributed in-memory key value store, whose nodes store system state as horizontally-partitioned data in a *ColumnMap* layout. The *Analytics Matrix* system state provides a materialized view on a large number of aggregates for each individual customer (subscriber). When an event arrives in an SPE, the corresponding record in the *Analytics Matrix* is updated atomically.

Lastly, in FlowDBMS/TSpoon [14,15], a key-value store is also employed and state is 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.

6.4 Remarks of Aspects of TSP Systems

Determining the right type of system design for TSPSs depends on a wide range of factors, including both software and hardware perspectives. For example, determining a suitable state representation will heavily depend on a state access pattern (e.g., with range selection, without range selection) and the hardware platform (e.g., in-memory, on disk). A stream processing task typically runs forever unless it is explicitly terminated by a user. Since both input workloads and hardware resources may change over time during execution, we envision an adaptive or progressive optimization approach is more suitable. Rather than improving on a single TSPSs design, multiple design options can be provided and selected on demand accordingly. Deciding which design to pick may be based on either a cost model-based approach or a learning model-based approach.

7 Advanced Issues for TSP Systems

Beyond relational database systems, the transaction paradigm serves to ensure the correctness of executions in specialized execution engines for varying application domains such as transactional memory [33], graph databases [49], distributed key-value stores [11] as well as stream processing [47]. In fact, many works [14,31,78,125] have emphasized the need for transactional state management in streaming applications. In particular, these applications typically employ streaming facilities to persist *state* or offer (near real-time) views of shared tables, and simultaneously employ transactional facilities to ensure a consistent representation of the *state* or a summary of the shared tables. Although some works do not employ transactional semantics, they can potentially benefit from TSP systems and these will also be discussed.

As illustrated in Figure 7, TSP is employed in four scenarios: *stream processing optimization*, *concurrent stateful processing*, *stream & database management system integration*, and *recoverable stream processing*. Note that some of these scenarios have multiple functionalities and therefore will be discussed from another perspective. Next, we discuss these four scenarios across varying real-world use cases.

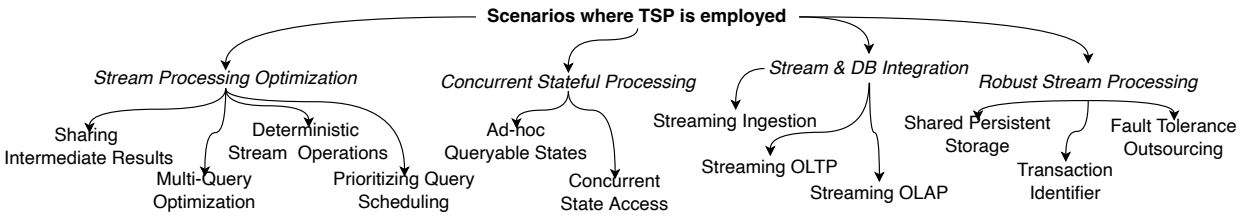


Fig. 7: Transactional stream processing for diverse needs.

7.1 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*.

Sharing Intermediate Results. To illustrate the sharing of intermediate results, we will examine two examples: STREAM and IBWJ (a parallel index-based window join algorithm). In the STREAM system [57], since multiple states called *synopses* may be *nearly* identical within a single query plan, they are kept in a single store to reduce storage redundancy. Since stores are shared, operators access their own states exclusively via an interface called a *stub*. However, as operators are scheduled independently, they will likely require slightly different views of the data. Hence, STREAM employs a simple timestamp-based execution mechanism to preserve correctness.

Due to Shahvarani and Jacobsen, IBWJ [100] utilizes a shared index data structure to accelerate sliding window joins, reduce redundant memory access during tuple matching, and improve performance. For all tuples in a window, an index structure is constructed. In order to match tuples, each time a new tuple arrives from one stream, IBWJ searches the index structure of the other stream. Intuitively, the arrival of a new tuple will trigger an update to an index structure. Since the index structure is shared among threads, this will raise additional concurrency control problems. Instead of adopting transaction semantics, Shahvarani et al. [100] propose a low-cost and effective concurrency control mechanism to meet the demands of high-rate update queries. The concurrency problem raised from sharing a synopsis and an index during a window join can naturally be handled by a TSP system instead of the previously proposed ad-hoc solution [57, 100]. This is especially important when properties, such as durability is required. For example, one can treat a shared index structure as a shared mutable state and model each read or write request to the structure as a transaction. However, to date, we are unaware of anyone having employed this approach in practice.

Multi-Query Optimization. Due to Ray et al. [95], the *SPASS* (Scalable Pattern Sharing on Event Streams) framework provides an optimizer that leverages time-based event correlations among queries and effectively shares processing among them. Initially, the optimizer finds a shared pattern plan in polynomial-time that covers all sequence patterns, while still ensuring an optimality bound. The runtime then exploits the shared continuous sliding view technology to execute the identified shared pattern plan. Since the sliding view may be concurrently modified by multiple pattern queries, a sequence transaction model on shared views is introduced to define the correctness of concurrent shared pattern execution. The most notable feature of SPASS is that it does not modify existing states. Instead, it can select, insert and delete shared states (i.e., sliding views). Unfortunately, some implementation details are not specified in the original paper (e.g., the data layout of the sliding views, the key to be used when searching for shared sliding views).

Deterministic Stream Operations. Handling an out-of-order stream is a common performance bottleneck, as there is a fundamental conflict between data parallelism and order-sensitive processing. Data parallelism seeks to improve operator throughput by allowing more than one thread to operate on different events concurrently. However, these events may be handled out-of-order. Most works attempt to solve the conflict by employing locks. In contrast, some works utilize non-lock (e.g., sorting) algorithms [135]. One of the interesting non-lock approaches involves using software transactional memory (STM) for stream processing as proposed by Brito et al. [33]. In their approach, processing a batch of input data at order-sensitive operators is modeled as a transaction, and commit timestamps are pre-assigned to transactions, effectively imposing order. Consequently, events that are received out-of-order and/or conflict with one another are processed in parallel optimistically, but are not output until all *preceding* events have been completed. In this way, they ensure that operators remain in a consistent state despite parallelization.

Prioritizing Query Scheduling. To handle potentially infinite data streams, continuous queries are typically formulated with a window constraint to limit the number of tuples that must be processed at any point in time. In most

implementations, the execution of a sliding window query and a window update is conducted serially (e.g., a query is triggered by the arrival of a tuple and during execution the query will update the corresponding window). Such an implementation implicitly assumes that a window cannot be advanced, while it is being accessed by a query.

Golab et al. [53] argue that the concurrent processing of queries (reads) and window-updates (writes) is required, to allow prioritized query scheduling to improve the freshness of answers. To achieve that, they model window updates and queries as transactions consisting of atomic sub-window reads and writes. However, it is obvious that such a computing strategy can lead to a read-write conflict as multiple threads may concurrently access the same state. Golab et al. [53] prove that the traditional notion of conflict serializability is insufficient in this context and define stronger isolation levels that restrict the allowed serialization orders following event ordering. Similarly, Golab et al. [55] propose mechanisms to index time-evolving data in secondary storage. When there are updates and reads from concurrent threads, it can also lead to conflicts. The authors mention that they plan to address the issue of update consistency due to changes made in-place or the replacement of an entire sub-index with a copy on which updates have been made.

7.2 Concurrent Stateful Processing

In this scenario, application workloads are comprised of both ad-hoc and continuous queries and these may access and modify common application states. For each use case below, we will discuss a few representative applications. For further examples of related applications readers are encouraged to examine Tatbul et al. [113], Wang et al. [125], Meehan et al. [78], and Zhang et al. [134].

Ad-hoc Queryable States. Ad-hoc queries (also called *snapshot queries*) are analogous to traditional database queries: They can be submitted to an SPE at any time, executed once, and answer queries (to provide insight into the current state of the system). Moreover, ad-hoc queries may be used to obtain further details in response to changes in the result of continuous queries.

Botan et al. [31] describe a streaming application, where real-time sensors generate temperature measurements (on a Celsius scale) to ensure (temperature-sensitive) devices are operating within their design specifications (stored in a database table). In their example, whenever a temperature reading falls out of the operating range, it will trigger an alert. Typically, an SPE would model this problem using a continuous query: For each incoming temperature reading, the table of specifications is probed and an alarm is raised whenever a violation is detected. That said, suppose that after the arrival of an event, we want to express temperatures on a Fahrenheit scale. This would require an update to the specifications table. However, if the updates do not occur prior to the arrival of new sensor measurements, false alarms may be generated.

To avoid the aforementioned problem, an SPE should be able to order table updates and stream temperature readings. However, this cannot be achieved in conventional SPEs. The fundamental challenge in supporting such workloads stems from the differences between the two query processing worlds: traditionally stored data sources focus on (read/write) operations, while stream processing operates on events. Although an ordering is defined among events (e.g., based on timestamps, based on arrival order) and operations (e.g., defined by a transactional model), there is no well-defined order across events and operations, which makes it impossible to compare them directly. Moreover, in batch processing queries are traditionally handled one at a time, whereas in stream processing long-running, continuous queries are commonplace. This problem is exacerbated when more complex analytical queries, such as scan and range lookups are required. In a similar setting, Shaikh et al. [101] consider the inconsistency problem that arises when relational data sources referenced by stream-relation joins are updated during stream processing.

Concurrent State Access. There are instances when a stream query's operators can share their states. To illustrate concurrent state access in stream processing we will consider the Ververica Streaming Ledger [9] (*SL*) application. Input streams are continuously processed by four operators: parser, deposit, transfer, and sink. During processing, these operators may need to share access to the application states consisting of account and asset data.

If the *SL* is implemented using a conventional SPE like Flink, the natural choice is to partition the application states into disjoint subsets based on accID and assID corresponding to the account table and asset table, respectively. However, it may be that a parameter in a transaction may be dependent on the value of certain states. For example, during the processing of a transfer request, an update to one account may depend on a value in another account, which can incidentally be modified concurrently. As a result, data dependencies among transactions in *SL* cannot easily be handled using conventional SPEs. A comparable stream application involving bank transactions is described by Affetti et al. [14, 15].

7.3 Stream and Database Management System Integration

There is a growing need for the integration of SPEs with database management systems [111]. For example, the arrival of continuous data streams from external sources into big data management systems are either processed incrementally or used to populate a persisted dataset

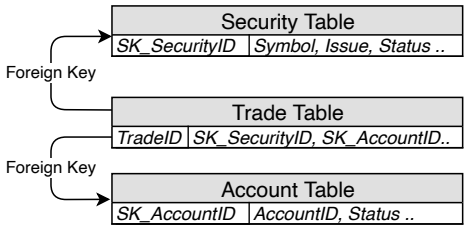
and associated indexes. Such a stream data ingestion (*Streaming Ingestion*) scenario can be well supported by TSPSs. Furthermore, TSPSs also support alternative ways of implementing OLTP queries (*Streaming OLTP*) as well as mixed stream and analytic queries (*Streaming OLAP*). Next, we will dive into each of these scenarios.

Streaming Ingestion. Data ingestion is an essential part of companies and organizations that collect and analyze large volumes of data. Traditionally, *batch ingestion* (say in ETL systems) processes large batches of data overnight. To keep pace with massive and fast-moving data, systems must be able to ingest, process, and persist data continually. In contrast, *streaming ingestion* processes smaller microbatches throughout the day to enable quicker access to incremental results. Following this idea, Meehan et al. [77] adapt TPC-DI [93] – a standard benchmark for data ingestion, to examine the effectiveness of streaming ingestion. However, TPC-DI [93] was originally designed as a benchmark for traditional ETL systems, the act of breaking large batches into smaller ones introduces new data dependencies [77].

Figure 8 illustrates a simplified version of a streaming TPC-DI workload with three tables of shared states: Security, Trade, and Account. Note that the Trade table contains foreign keys on both the Security and Account tables as shown in Figure 8 (a). When new rows are defined within the Trade table, a reference must be made to the other two tables to assign the SK_SecurityID and SK_AccountID keys, which means that the corresponding rows must already exist in their respective tables. Such data dependencies can naturally be expressed in current SPEs as a streaming dataflow graph as illustrated in Figure 8 (b), which contains four major operators: parser, update security, update account, and update trade. The input events, e_1, e_2 (a microbatch of files) are processed sequentially by these major operators fulfilling the operator ordering constraint. During processing, these three tables may be read or updated concurrently by all operators and their parallel instances.

Streaming OLTP. Conventional OLTP workloads can be handled using streaming queries. To support this idea, Chen and Migliavacca [43] proposed *StreamDB*, which adopts a TSP scheme. In *StreamDB*, there are only three types of operators in a streaming query, including the 1) *Source* operator, which receives transactions from client applications, timestamps them, and issues transactions to downstream data operators; 2) *Data* operator, which maintains a portion of a database (either as a vertical or a horizontal partition), carries transaction execution and produces intermediate results for other data operators, or passes the final results to *Sink*; 3) *Sink* operator, which receives transaction responses.

Figure 9 illustrates an example implementation of TPC-C in *StreamDB*. By splitting a database among multiple



(a) Three Tables with Shared States

(b) Streaming Dataflow Graph

Fig. 8: Streaming TPC-DI (Streaming Ingestion).

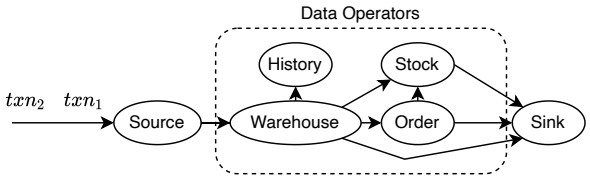


Fig. 9: Streaming TPC-C (*Streaming OLTP*).

data operators, *StreamDB* reduces (and can sometimes completely remove) lock contention during transaction processing. However, how to optimally come up with a good stream dataflow graph for an arbitrary OLTP workload remains an open question.

Streaming OLAP. Many organizations require real-time analysis of their data streams to make instantaneous decisions and a number of related applications have been described in the literature [32, 56, 102]. Huawei-AIM workload [32] (*AIM*) is a specifically designed telecommunications workload as a result of a collaboration between Huawei and the ETHZ Systems Group. AIM has a three-tier architecture consisting of storage, an SPE, and real-time analytics (RTA) nodes. RTA nodes push analytical queries down to the storage nodes, merge the partial results, and finally deliver the results to the client. ESP nodes process the incoming event stream and update the corresponding records by sending *Get* and *Put* requests to the storage nodes. The consistent state (or snapshot) is not allowed to be older than a certain bound, which is a service level objective (SLO) of the AIM workload. This requirement is very different from other applications, which usually require serializable isolation level.

7.4 Robust Stream Processing

Besides such requirements as *scalability* and *low latency*, many critical streaming applications require SPEs to recover quickly in the event of a failure [106]. It is for this reason

that a lot of activity has been directed at achieving fault tolerance in SPEs. Next, we discuss earlier attempts to leverage transactional-like concepts, so as to achieve high availability and fault tolerance in stream processing.

Shared Persistent Storage. MillWheel [17] runs stateful computations using an event-driven API and persistently stores input and output of operations for each input tuple before it influences other elements or operational states. In MillWheel all state updates are handled remotely using a storage system like BigTable [41]. This remote store has to independently handle fault-tolerance (e.g., by replicating data). To avoid inconsistencies in persisted state, MillWheel wraps all per-key updates in a single atomic operation. Unlike MillWheel, Samza [84] takes an alternative approach, it combines local on-disk storage, a more efficient changelog, and caching mechanisms.

Transaction Identifier. The second approach, represented by Tridents "transactional topology" [76], treats the whole stream processing topology as a single operation. It uses small batches of tuples and assigns them unique transaction identifiers (*TXID*). TXID are logged in external storage along with the state of the operator and used to figure out the state of a failed tuple. A batch requires re-submission in the event of an unmatched TXID, in order to recover from failure. However, to perform correctly, it requires a strict transaction processing ordering, which limits system throughput. Furthermore, it ignores the state of buffered inputs and thus suffers from the loss of intermediate results in the event of failures.

Fault Tolerance Outsourcing. Ishikawa et al. [62] aims to develop new data stream processing methodologies, such as incorporating fault tolerance in an OLTP engine. When backing up data streams, they propose to write the data into an in-memory database system instead of a file system. In this case, the persistence of the data is supported by the function of the database system. Such a mechanism is very similar in spirit to H-Store's state partitioning transaction processing mechanism. While storing state in an external file system outsources the responsibility of fault-tolerance, this approach is inefficient. It can also overwhelm the remote store (e.g., in the presence of spikes), which can negatively impact other applications using the shared store.

7.5 Remarks concerning Advanced Issues for TSP Systems

From the above analysis, it is evident that TSP based application use cases have diverse requirements, which differ from traditional database workloads or ordinary stream processing workloads. For example, some applications do not require insert and delete operations at all, while some others do not update shared mutable states. In addition, TSP has the potential to better support conventional database workloads (i.e., OLTP and OLAP) as well. However, there are many open research questions yet to be solved. Another interesting take away 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. Systems can take advantage of this to simplify their design and improve system performance.

8 Orthogonal Work

Next, we turn our attention to ongoing research on problems that are partially related to TSP (i.e, arising only in some applications that draw on TSP as well as other areas). This includes research on non-transactional state sharing and non-transactional streaming properties.

Non-Transactional State Sharing. In the context of stream processing, state sharing is often prohibited. However, a number of recent works [100, 138] have pointed out the necessity of supporting state sharing during stream processing. Due to the potential for conflicts when reading and writing to the same state, these works also need to address concurrency. For example, Cheng et al. [138] recently introduced a *timestamped state sharing* technique as a novel system-wide state abstraction. In contrast to TSP, the key difference in these works is that they neither consider `ACID`, nor the concept of a transaction. Instead, they assume there are only single key accesses, i.e., each read and write contains only one key and the proposed system only provides key-value store style APIs (i.e., Put/Get).

Non-Transactional Streaming Properties. In contrast to transactional processing, TSP needs to guarantee additional streaming properties (previously referenced in Section 4.2): *operator ordering* and *event ordering*. These streaming properties are also important in non-transactional stream processing applications. A stream operator is order-sensitive, if it requires input events to be processed in a predefined order (e.g., chronological order). Handling out-of-order input data in an order-sensitive operator often turns out to be a performance bottleneck, as there is a fundamental conflict between *data parallelism*, which seeks to improve the throughput of an operator by letting more than one thread operate on different events concurrently (possibly out-of-order) and *order-sensitive processing*. Various methods and techniques have been proposed to guarantee an ordering during stream processing, including general techniques, such as a buffer-based technique [24], a punctuation-based technique [72], speculative techniques [97], a synchronization specification based technique [19], and specially-designed techniques applied in a certain type of operator, such as window aggregation [119]. In comparison to TSP, the key difference of these works is that TSP offers more fine-grained control of state accesses that are ordered than ordered event processing can offer. Specifically, input

events may be processed in an arbitrary order in TSPSs, but their issued transactions must be equivalent to a conflict-free schedule that follows an input event sequence.

9 Research Outlook

To support modern streaming applications, traditional SPEs complement data management tools for processing, storing, and querying [98]. Thereby reducing the complexity of managing separate subsystems, relieving developers from having to integrate these solutions, as well as minimizing the risk of introducing functional errors and performance problems. Adopting TSP would not only provide great benefits, but also mitigate the aforementioned challenges. Next, we offer a perspective on future research directions.

9.1 Standardization and Benchmarking of TSP

Many applications would benefit from TSP, including Healthcare [125], the IoT [31], and E-commerce [14]). However, to date, there is no standard benchmark for TSPSs [112], which must include comprehensive performance metrics, diversity of workload features, and meaningful application scenarios. In Table 1, we list 13 applications that may serve as a starting point for the construction of a standard benchmark. In particular, these applications encompass many important features, such as:

Access Scope of Shared State: Some applications require varying degrees of access to shared states with wide ranging scope, spanning from intra-operator to inter-systems. In the event an application has mixed scopes (i.e., diverse access rights to shared states), then it would be classified according the broadest access rights. In the case of stream operators, these may need to maintain multiple states. For example, the shared states can be the index structure of an input stream or some other user-defined data structure. Concurrency control issues arise when more than one entity (i.e., threads) shares state (i.e., intra-operator) and may concurrently modify the same state during execution. States may be also shared among multiple operators or among multiple queries. 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.

Transaction Scope: In some streaming applications a *transaction* is the (entire) process employed when handling a batch of input events, while others define a transaction in a fine-grained manner, such as the processing of an input event at a stream operator. There are also cases where a transaction is defined flexibly within each operation.

Data Dependency: In most use cases, the parameters (e.g., read/write sets) of a stream transaction are given explicitly or can be extracted directly from an input event. However, in some use cases, when parameters depend on the value of a certain state, a *data dependency* exists. A state transaction with a data dependency is more difficult to handle as their dependent state may be modified by other transactions that are running concurrently.

Data Manipulation Statements (S/U/I/D): Data manipulation statements employed in applications constrain both system design and potentially optimizations. Some applications require an *insert* (I) or a *delete* (D) statement to be supported. When the system needs to further maintain a foreign key constraint, such as in the case of streaming ETL [77], storing shared states as relations could be a reasonable choice of the system design. However, a foreign key constraint is rarely required in most TSP scenarios, and many need only the *select* (S) and *update* (U) statement to be supported for manipulating shared states during stream processing. In such a case, storing shared states as vanilla key-value pairs is sufficient and simplifies the design of TSPSs. In general, specific optimizations shall be adopted by the TSPSs, according to the application needs.

Properties of Transactional Stream Processing: The properties, include ACID properties as well as three constraints (an operation ordering constraint, an event ordering constraint, and delivery guarantees).

- *ACID Properties: Atomicity (A)* requires each transaction to execute either all its operations or none at all. It is a common property in varying applications in TSPSs, as well as traditional database systems. *Consistency (C)* traditionally refers to maintaining integrity constraints in database systems. In TSP, applications can also similarly enforce constraints on shared mutable states. *Isolation (I)* guarantees correctness even when multiple transactions are interleaved. Most TSP applications require serializable isolation, while some others allow for snapshot isolation. *Durability (D)* requires that modifications performed by a transaction are permanent. This property is critical in order to recover from a system crash. Many TSP applications do not explicitly require durability, however, they resort to an SPE's built-in fault tolerance mechanism.
- *Operation Ordering Constraint:* This constraint requires the execution of a transaction to follow the topological ordering of stream operators expressed in the DAG. As events flow through the DAG, this requirement is naturally preserved in SPEs. However, in order to support TSP in database systems, this constraint would impose yet another requirement.
- *Event Ordering Constraint:* This constraint requires transactions with conflicting schedules to be ordered by their triggering event timestamp. Note that this is unlike *external consistency* and *strict serializability* as

Table 1: Summary of Relevant Applications

App Name	Access Scope of Shared State	Transaction Scope	Data Dependency	Data Manipulation Statements	Properties of Transactional Stream Processing			
					ACID	Operator Ordering Constraint	Event Ordering Constraint	Deliverability Properties
Deterministic Stream Operation [33]	Intra-operator	Per operator execution	No	S/U	AI	No	Yes	Exactly-once
Shared Index Window Join [100]	Intra-operator	Per operator execution	No	S/U	AI	No	Yes	Exactly-once
Toll Processing [134]	Intra-query	Per operator execution	No	S/U	ACID	Yes	Yes	Exactly-once
Streaming Ledger [9]	Intra-query	Per operator execution	Yes	S/U	ACID	Yes	Yes	Exactly-once
Leaderboard Maintenance [78]	Intra-query	Per operator execution	Yes	S/U/I/D	ACID	Yes	Yes	At-least-once
Active Complex Event Processing [125]	Inter-query	Per query execution	No	S/U	AID	No	Yes	Exactly-once
Multi-Pattern-Query Optimization [95]	Inter-query	Per query execution	No	S/I/D	AI	No	Yes	Exactly-once
Concurrent Sliding Window [53]	Inter-query	Per operator/query execution	No	S/I/D	AI	No	Yes	Exactly-once
Monitoring with Updatable Specifications [31]	Inter-query	Per query execution	No	S/U	ACID	No	Yes	Exactly-once
Waveforms Alert Monitoring [113]	Inter-query	Per operator/query execution	No	S/U/I/D	AID	Yes	Yes	Exactly-once
Streaming Ingestion [77]	Inter-system	Per operator execution	No	S/U/I/D	ACID	Yes	Yes	Exactly-once
Analytics In Motion [32]	Inter-system	Per query execution	No	S/U	Snapshot isolation	No	Yes	Exactly-once
Streaming OLTP [43]	Per transaction	user-defined	No	S/U/I/D	ACID	Yes	Yes	At-least-once

the required schedule is determined explicitly by input event rather than the transaction execution order.

- *Delivery Guarantee:* Applications may require different guarantees concerning how each input event is to be processed. An *exactly-once* guarantee would ensure that each input is processed once and only once. An *at-least-once* guarantee would ensure that each input is always processed, but maybe more than once. An *at-most-once* guarantee would ensure that each input is either processed or dropped, but never revised. Few applications demand an *at-most-once* guarantee.

9.2 Novel Applications

Due to the rise of the IoT, data are increasingly generated in real-time and these need to be processed as soon as possible. Traditionally, big data applications were designed to tackle large static datasets. However, today's applications are far more demanding. Consequently, we envision the development of novel streaming applications that would benefit from TSP solutions. Moreover, the spectrum of applications that SPEs serve is further widening. Currently, some research is investigating novel systems that meet these requirements, for example, NebulaStream [132]. Next, we turn our attention to varying application areas, including online machine learning/stream mining, mixed batch/stream transactional workloads, streaming materialized views, and operational stream processing.

Online Machine Learning/Stream Mining. Online learning and mining from streams of data will soon become mandatory for data scientists. Traditional machine learning (ML) algorithms assume that all data is available in advance. In addition, the assumption is that these algorithms will iterate over time, refine model parameters, and reach a global optimization objective on the entire dataset [61]. To support streaming data, online ML algorithms define a local objective function for each item with respect to the current model parameters, and search for locally optimal parameters. It has been shown that the local approach can converge to a global optimal point, subject to certain conditions [60, 139].

Driven by those demands, attempts have been made in the literature to support continuous queries (CQs) to reference non-streaming resources, such as relational data in database systems and ML models [89]. Model-based streaming systems, like anomaly detectors, depend on predictions that are generated from weeks worth of data, and their models must be updated on-the-fly as new data arrives. Scaling these systems by orders of magnitude should not cause a commensurate increase in the operational costs of building and maintaining the system [17]. However, due to the lacking of transactional support in conventional SPEs, users have to apply cumbersome workarounds in the implementation of emerging streaming mining algorithms, such as streaming event detection [98]. It thus remains an interesting future work to study how those online ML and stream mining algorithms can be supported efficiently in TSPSs, which bring features, such as elastic scaling, fault tolerance guarantees, and shared state consistency to users, even at the virtual space [86].

Mixed Batch/Stream Transactional Workloads. An increasing number of enterprise applications, particularly

those in financial trading and the IoT, produce mixed workloads by simultaneously supporting continuous stream processing, online transaction processing (OLTP), and online analytical processing (OLAP). For example, DeltaLake [22] lets streaming jobs write small objects into a table at low latency, then for performance reasons transactionally coalescing them into larger objects at a later point time. Fast "tailing" reads of new data added to a table are also supported, so that jobs can treat a Delta table as a message bus. Tatbul [111] enumerates the challenges in streaming data integration, such as the lack of a common semantic model across different systems, optimization challenges, and transactional issues. To date, these challenges are still present and this is largely due to: (1) the diverse applications proposed in the literature, and (2) each system focuses on a narrow list of features.

Meehan et al. [77] discussed self-driving vehicles as an example to motivate the need for streaming ingestion. They argue that the traditional data integration process is insufficient, given that the value of sensor data decreases drastically over time. As a consequence, the ability to make decisions based on that data is only useful, if the analysis is done in near real-time. There is also a necessity to maintain the order of time series, however, only if it can be processed in a timely manner (i.e., not waiting hours for a large batch to become available). Additionally, time series can become very large quickly, particularly, if sensor sample rates are high. Storing this data can become extremely expensive, and it is likely that the entirety of the time series need not be stored, to compute the relevant analytics.

Streaming Materialized Views. Materialized views (MVs) are essentially stored continuous queries that are re-executed as their base data are modified. MVs were developed precisely to address the inherent inefficiencies in store-first-query-later database technologies for query-heavy, non-ad hoc workloads observed in many analytics applications [50]. It conceptually shares many high-level similarities with stateful stream processing that continuously updates states. Traditional MV algorithms are not optimized for high-velocity data stream processing. They still require storing the entire dataset and do not fully exploit the time-oriented semantics of the data and queries in modern analytics workloads. For example, stream updates are often append-only or peak-only, while MVs are designed to treat random updates as first-class citizens. There is a rising need for *streaming materialized views* (SMVs), which require the system to be able to handle high-velocity inputs, continuously update states with nearly random access patterns, and share the updated states among concurrent running entities (e.g., continuous queries). For example, a recent work by Verheijde et al. [120] proposed S-Query, which focuses on exposing the live uncommitted state and past consistent state with serializable isolation of a shared-nothing distributed streaming system to external applications. Different from TSPSs, S-Query does not provide strict ACID guarantees. Thus, the clear distinction of use cases drives also a clear separation of concerns.

Winter et al. [127] recently proposed a so-called *continuous view* scheme that is highly correlated with the design goal of SMVs. It is implemented in a database system called Umbra and compared with Flink [36]. However, Flink was not designed to handle such workloads. Furthermore, the performance gap between an in-memory database system and a dedicated SPE that handles streaming applications has been reported in prior work [66]. It remains to be seen whether the proposed approach implemented in a database system can outperform a state-of-the-art TSPSs, such as S-Store [78], TStream [134] and TSpoon [15], which are better able to handle emerging SMVs-like workloads.

Operational Stream Processing. Katsifodimos et al. [64] discussed a use case involving employing an SPE as a backend for stateful event-driven applications, such as microservices. An example of such a use case called *stateful function-as-a-service* was also recently demonstrated [16]. A set of requirements were discussed including *ACID* transactions, global state consolidation, and the need for debugging and auditing. Those requirements are correlated surprisingly to those required by TSP, including transactional shared state management during stream processing. However, whether the TSPSs, such as S-Store [78] can fully satisfy the requirements of *operational stream processing* still remains an open question. For example, operational stream processing has a particular focus on supporting stateless computing paradigm. In the context of TSP questions like how to support debugging [70] and isolation [103] during the execution of stateful stream processing as microservices are also interesting to explore.

9.3 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 TSPSs and put emerging hardware platforms to good use [135]. Next, we take a closer look at multi-/many-core architectures, non-volatile storage, and trusted computing platforms.

Multi-/Many-core Architectures. In order to support shared mutable states, TSPSs have a potential system bottleneck, namely, concurrent state accesses. TStream [134] is an example of a recent attempt to utilize multicore CPUs successfully and improve concurrent shared state access performance. It exploits parallelism opportunities via two novel techniques: dual-mode scheduling and a dynamic transaction restructuring mechanism. However, state-of-the-art TSPSs (e.g.,

TStream [134]) are still not scalable when there are lots of input dependencies (i.e., the state access of one event depends on the state access results of another event) among the processing of multiple events. More broadly beyond relational analytics, more investigation is required to further enhance existing TSPSs when dealing with more complicated types of workloads (e.g., machine learning, graph aggregation). We also need to revisit the current design of TSPSs for the emerging multi-/many-core architectures with high-bandwidth memory, which provides a potential solution to the common system bottleneck seen in TSPSs. Further scaling TSPSs on multi-node settings with the same or relaxed correctness guarantees is nontrivial and requires extensive future exploration [112].

Non-Volatile Storage. Non-Volatile Memory (NVM) has emerged as a promising technology that brings many new opportunities and challenges. NVM technology promises to combine the byte-addressability and low latency of DRAM with the persistence and density of block-based storage media. However, NVM suffers from a limited cell endurance and read-write asymmetry regarding latency. Fernando et al. [91] have recently explored efficient approaches to support analytical workloads on NVM, where an NVM-aware storage layout for tables is presented based on a multidimensional clustering approach and a block-like structure to utilize the entire memory stack. As argued by the authors, the storage structure designed on NVM may serve as the foundation for TSPSs [99] in the future.

Non-Volatile Memory Express (NVMe)-based solid-state devices (SSDs) are expected to deliver unprecedented performance in terms of latency and peak bandwidth. For example, the recently announced PCIe 4.0-based NVMe SSDs [2] are already capable of achieving a peak bandwidth of 4GB/s. Lee et al. [71] have recently investigated the performance limitations of current SPEs on managing application states on SSDs and have shown that query-aware optimization can significantly improve the performance of stateful stream processing on SSDs. Their pioneering work is also highly valuable for TSPSs due to its strict `ACID` and streaming property requirements. However, further investigation is required.

Trusted Computing Platforms. The demand for low-latency and the local processing of sensitive data in the IoT requires that data be processed as much as possible near the source and calls for data stream processing on the edge. However, edge devices are often vulnerable to attacks because of their limited power and computing capacity. As a result, edge processing exposes sensitive data to severe security threats. Given the IoT's exploding volume and edge devices' weak nature satisfying stream processing requirements should incorporate a scalable and energy-efficient security mechanism. Thus, a plausible solution [90] are trusted computing platforms (TCPs), which protect data and codes by loading them into an isolated and encrypted memory area. However, to bring TSP on TCPs is nontrivial and requires further investigation. In particular, TCPs provide limited physical memory, and therefore, how to perform transactional stateful stream processing within limited memory remains an open question [79]. In addition, we need to scale the system to multiple TCPs in a distributed environment, which is sometimes a must, since each computing node has its own computational limits.

10 Conclusion

Driven by the realization that static data is merely a partial snapshot of a data stream, the data technology industry is focusing increasingly on data streams (also known as data-in-motion). The current predominant data management solution, i.e., database systems fall short of the throughput and latency requirements demanded by modern data processing applications. Specialised stream processing engines (SPEs) were subsequently developed, such as Storm, Flink and Spark Streaming. Despite their architectural diversity, none of these systems addresses one of the primary features originally provided by database systems, i.e., support for transactions. TSPSs resolve such issues by employing transactional semantics and providing the opportunity to support novel applications and system optimizations. In this survey, we reviewed the application scenarios of TSP and discussed both theoretical models, actual designs, and implementation of TSPSs. TSPSs relieve the burden of managing state consistency from users so that they can focus on the development of complex streaming applications. However, there is still a long way to go, and future studies, including standard API specifications, new types of accelerators, deep performance analysis, and the development of new toolchains can significantly push the progress forward.

11 Competing interests

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests:

The research leading to these results received funding from the National Research Foundation, Singapore and Infocomm Media Development Authority under its Future Communications Research & Development Programme under Grant Agreement No.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).

References

1. Apache flink 101 - the rise of stream processing and beyond: http://bigdatausecases.info/entry/apache-flink-101-the-rise-of-stream-processing-and-beyond. Last Accessed: 2020-07-24
2. Corsair force series nvme ssd. https://hothardware.com/news/corsair-mp600
3. Introducing kafka streams: Stream processing made simple, https://www.confluent.io/blog/introducing-kafka-streams-stream-processing-made-simple. Last Accessed: 2022-07-31
4. Maxdb, https://maxdb.sap.com/
5. Rocksdb. http://rocksdb.org/
6. Coral8, inc, http://www.coral8.com/ (2008)
7. treambase systems, inc., ttp://www.streambase.com/ (2008)
8. Apache smaza, https://samza.apache.org/learn/documentation/0.7.0/container/state-management.html (2018)
9. Data Artisans Streaming Ledger Serializable ACID Transactions on Streaming Data, https://www.data-artisans.com/blog/serializable-acid-transactions-on-streaming-data (2018)
10. Stateful stream processing in flink, https://ci.apache.org/projects/flink/flink-docs-stable/ops/state/state_backends.html (2018). URL https://cwiki.apache.org/confluence/display/FLINK/Stateful+Stream+Processing
11. Tikv, a distributed transactional key-value database, https://tikv.org/ (2020)
12. 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. In: CIDR'05, vol. 5, pp. 277–289 (2005)
13. 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. The VLDB Journal **12**(2), 120–139 (2003). DOI 10.1007/s00778-003-0095-z. URL http://dx.doi.org/10.1007/s00778-003-0095-z
14. 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). DOI 10.1145/3093742.3093929. URL http://doi.acm.org/10.1145/3093742.3093929
15. Affetti, L., Margara, A., Cugola, G.: Tspoon: Transactions on a stream processor. Journal of Parallel and Distributed Computing **140**, 65–79 (2020). DOI https://doi.org/10.1016/j.jpdc.2020.03.003. URL http://www.sciencedirect.com/science/article/pii/S0743731518305082
16. Akhter, A., Fragkoulis, M., Katsifodimos, A.: Stateful functions as a service in action. Proceedings of the VLDB Endowment **12**(12), 1890–1893 (2019)
17. 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 (2013-08). DOI 10.14778/2536222.2536229
18. Akidau, T., Bradshaw, R., Chambers, C., Chernyak, S., Fernndez-Moctezuma, R.J., Lax, R., McVeety, S., Mills, D., Perry, F., Schmidt, E., Whittle, S.: The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. Proceedings of the VLDB Endowment **8**, 1792–1803 (2015)
19. Alur, R., Hilliard, P., Ives, Z.G., Kallas, K., Mamouras, K., Niksic, F., Stanford, C., Tannen, V., Xue, A.: Synchronization schemas (2021)
20. Arasu, A., Babu, S., Widom, J.: The cql continuous query language: Semantic foundations and query execution. The VLDB Journal **15**(2), 121–142 (2006). DOI 10.1007/s00778-004-0147-z. URL http://dx.doi.org/10.1007/s00778-004-0147-z
21. Arasu, A., Cherniack, M., Galvez, E., Maier, D., Maskey, A.S., Ryvkina, E., Stonebraker, M., Tibbetts, R.: Linear road: A stream data management benchmark. In: Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30, Vldb '04, pp. 480–491. VLDB Endowment (2004). URL http://dl.acm.org/citation.cfm?id=1316689.1316732
22. Armbrust, M., Das, T., Sun, L., Yavuz, B., Zhu, S., Murthy, M., Torres, J., van Hovell, H., Ionescu, A., undefineduszczak, A., undefinedwitakowski, 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), 34113424 (2020). DOI 10.14778/3415478.3415560. URL https://doi.org/10.14778/3415478.3415560
23. 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, p. 419430. Association for Computing Machinery, New York, NY, USA (2004). DOI 10.1145/1007568.1007616. URL https://doi.org/10.1145/1007568.1007616
24. Babu, S., Srivastava, U., Widom, J.: Exploiting k-constraints to reduce memory overhead in continuous queries over data streams. ACM Trans. Database Syst. **29**(3), 545–580 (2004). DOI 10.1145/1016028.1016032. URL http://doi.acm.org/10.1145/1016028.1016032
25. Bailis, P., Ghodsi, A.: Eventual consistency today: Limitations, extensions, and beyond: How can applications be built on eventually consistent infrastructure given no guarantee of safety? Queue **11**(3), 2032 (2013). DOI 10.1145/2460276.2462076. URL https://doi.org/10.1145/2460276.2462076
26. Barga, R.S., Caituiro-Monge, H.: Event correlation and pattern detection in cedr. In: International Conference on Extending Database Technology, pp. 919–930. Springer (2006)
27. Bernstein, P., Newcomer, E.: Principles of Transaction Processing: For the Systems Professional. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1997)
28. Bernstein, P.A., Goodman, N.: Concurrency control in distributed database systems. ACM Comput. Surv. 1981 **13**(2), 185–221 (1981). DOI 10.1145/356842.356846. URL http://doi.acm.org/10.1145/356842.356846
29. 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, p. 934945. Association for Computing Machinery, New York, NY, USA (2009). DOI 10.1145/1516360.1516467. URL https://doi.org/10.1145/1516360.1516467
30. 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 (2009). DOI 10.1007/978-3-642-14559-9_2
31. 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). DOI 10.1145/2247596.2247622. URL http://doi.acm.org/10.1145/2247596.2247622
32. 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, p. 251264. Association for Computing Machinery, New York, NY, USA (2015). DOI 10.1145/2723372.2742783. URL https://doi.org/10.1145/2723372.2742783
33. 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*, vol. 332, pp. 265–275. ACM (2008). DOI 10.1145/1385989.1386023. URL https://doi.org/10.1145/1385989.1386023
34. 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 (2017-08). DOI 10.14778/3137765.3137777. URL https://doi.org/10.14778/3137765.3137777
35. 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, p. 26512658. Association for Computing Machinery, New York, NY, USA (2020). DOI 10.1145/3318464.3383131. URL https://doi.org/10.1145/3318464.3383131
36. Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., Tzoumas, K.: Apache flink: Stream and batch processing in a single engine. Bulletin of the IEEE Computer Society Technical Committee on Data Engineering **36**(4) (2015)
37. 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). DOI 10.14778/2733004.2733048. URL http://dx.doi.org/10.14778/2733004.2733048
38. Chandramouli, B., Goldstein, J., Barnett, M., DeLine, R., Fisher, D., Platt, J.C., Terwilliger, J.F., Wernsing, J.: Trill: A high-performance incremental query processor for diverse analytics. Proceedings of the VLDB Endowment **8**(4), 401–412 (2014). DOI 10.14778/2735496.2735503. URL http://dx.doi.org/10.14778/2735496.2735503
39. Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M.J., Hellerstein, J.M., Hong, W., Krishnamurthy, S., Madden, S.R., Reiss, F., Shah, M.A.: Telegraphcq: Continuous dataflow processing. In: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, Sigmod '03, pp. 668–668. Acm, New York, NY, USA (2003). DOI 10.1145/872757.872857. URL http://doi.acm.org/10.1145/872757.872857
40. Chandrasekaran, S., Franklin, M.: Remembrance of streams past: Overload-sensitive management of archived streams. In: VLDB (2004)
41. 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)
42. 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 Transactions on Computer Systems (TOCS) **26**(2), 1–26 (2008)
43. 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)
44. 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)
45. 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)
46. 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, p. 509514. Association for Computing Machinery, New York, NY, USA (2011). DOI 10.1145/1951365.1951426. URL https://doi.org/10.1145/1951365.1951426
47. Conway, N.: Cisc 499*: Transactions and data stream processing. Apr **6**, 28 (2008)
48. 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: Googles globally distributed database. ACM Transactions on Computer Systems (TOCS) **31**(3), 1–22 (2013)
49. Dubey, A., Hill, G.D., Escriva, R., Sirer, E.G.: Weaver: A high-performance, transactional graph database based on refinable timestamps. Proceedings of the VLDB Endowment **9**(11) (2016)
50. 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)
51. Garcia-Molina, H., Salem, K.: Sagas. SIGMOD Rec. **16**(3), 249259 (1987). DOI 10.1145/38714.38742. URL https://doi.org/10.1145/38714.38742
52. Gedik, B., Andrade, H., Wu, K.L., Yu, P.S., Doo, M.: Spade: the system s declarative stream processing engine. In: Proceedings of the 2008 ACM SIGMOD international conference on Management of data, pp. 1123–1134. Acm (2008)
53. Golab, L., Bijay, K.G., Özsu, M.T.: On concurrency control in sliding window queries over data streams. In: Y. Ioannidis, M.H. Scholl, J.W. Schmidt, F. Matthes, M. Hatzopoulos, K. Boehm, A. Kemper, T. Grust, C. Boehm (eds.) Advances in Database Technology - EDBT 2006, pp. 608–626. Springer Berlin Heidelberg, Berlin, Heidelberg (2006)
54. 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. 658669. Association for Computing Machinery, New York, NY, USA (2005). DOI 10.1145/1066157.1066232. URL https://doi.org/10.1145/1066157.1066232
55. Golab, L., Prahladka, P., Ozsu, M.T.: Indexing time-evolving data with variable lifetimes. In: 18th International Conference on Scientific and Statistical Database Management (SSDBM'06), pp. 265–274 (2006). DOI 10.1109/SSDBM.2006.29
56. Götze, P., Sattler, K.: Snapshot isolation for transactional stream processing. In: EDBT (2019)
57. Group, S., et al.: Stream: The stanford stream data manager. Tech. rep., Stanford InfoLab (2003)
58. Grulich, P.M., Breß, S., Zeuch, S., Traub, J., Bleichert, J.v., Chen, Z., Rabl, T., Markl, V.: Grizzly: Efficient stream processing through adaptive query compilation. In: Proceedings of the

ACM SIGMOD International Conference on Management of Data (SIGMOD 2020). Acm Sigmod (2020)
59. 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, p. 2732. Association for Computing Machinery, New York, NY, USA (2006). DOI 10.1145/1315903.1315910. URL https://doi.org/10.1145/1315903.1315910
60. Hoffman, M.D., Blei, D.M., Bach, F.: Online learning for latent dirichlet allocation. In: Proceedings of the 23rd International Conference on Neural Information Processing Systems - Volume 1, NIPS'10, p. 856864. Curran Associates Inc., Red Hook, NY, USA (2010)
61. Huang, Q., Lee, P.P.C.: Toward high-performance distributed stream processing via approximate fault tolerance. Proc. VLDB Endow. **10**(3), 7384 (2016). DOI 10.14778/3021924.3021925. URL https://doi.org/10.14778/3021924.3021925
62. Ishikawa, Y., Sugiura, K., Takao, D.: Fault tolerant data stream processing in cooperation with oltp engine. In: A. Mondal, H. Gupta, J. Srivastava, P.K. Reddy, D. Somayajulu (eds.) Big Data Analytics, pp. 3–14. Springer International Publishing, Cham (2018)
63. Kallman, R., Kimura, H., Natkins, J., Pavlo, A., Rasin, A., Zdonik, S., Jones, E.P.C., Madden, S., Stonebraker, M., Zhang, Y., Hugg, J., Abadi, D.J.: H-store: A high-performance, distributed main memory transaction processing system. Proc. VLDB Endow. **1**(2), 1496–1499 (2008). DOI 10.14778/1454159.1454211. URL http://dx.doi.org/10.14778/1454159.1454211
64. Katsifodimos, A., Fragkoulis, M.: Operational stream processing: Towards scalable and consistent event-driven applications. (2019)
65. Katsipoulakis, N.R., Labrinidis, A., Chrysanthis, P.K.: A holistic view of stream partitioning costs. Proc. VLDB Endow. **10**(11), 1286–1297 (2017-08). DOI 10.14778/3137628.3137639
66. Kipf, A., Pandey, V., Böttcher, J., Braun, L., Neumann, T., Kemper, A.: Scalable analytics on fast data. ACM Trans. Database Syst. **44**(1) (2019). DOI 10.1145/3283811. URL https://doi.org/10.1145/3283811
67. Koliousis, A., Weidlich, M., Castro Fernandez, R., Wolf, A.L., Costa, P., Pietzuch, P.: Saber: Window-based hybrid stream processing for heterogeneous architectures. In: Proceedings of the 2016 International Conference on Management of Data, Sigmod '16, pp. 555–569. Acm, Acm, New York, NY, USA (2016). DOI 10.1145/2882903.2882906. URL http://doi.acm.org/10.1145/2882903.2882906
68. 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, p. 10811092. Association for Computing Machinery, New York, NY, USA (2010). DOI 10.1145/1807167.1807290. URL https://doi.org/10.1145/1807167.1807290
69. Kulkarni, S., Bhagat, N., Fu, M., Kedigehalli, V., Kellogg, C., Mittal, S., Patel, J.M., Ramasamy, K., Taneja, S.: Twitter heron: Stream processing at scale. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Sigmod '15, pp. 239–250. Acm, Acm, New York, NY, USA (2015). DOI 10.1145/2723372.2742788. URL http://doi.acm.org/10.1145/2723372.2742788
70. Kumar, A., Wang, Z., Ni, S., Li, C.: Amber: A debuggable dataflow system based on the actor model. Proc. VLDB Endow. **13**(5), 740753 (2020). DOI 10.14778/3377369.3377381. URL https://doi.org/10.14778/3377369.3377381
71. 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). DOI 10.1145/3265723.3265739. URL http://doi.acm.org/10.1145/3265723.3265739
72. Li, J., et al.: Out-of-order processing: A new architecture for high-performance stream systems. Proc. VLDB Endow. (2008)
73. 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)
74. Liarou, E., Kersten, M.: Datacell: Building a data stream engine on top of a relational database kernel. In: VLDB PhD Workshop (2009)
75. 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)
76. Marz, N.: Trident API Overview: github.com/nathanmarz/storm/wiki/trident-apioverview. github.com/nathanmarz/storm/wiki/Trident-APIOverview
77. Meehan, J., Aslantas, C., Zdonik, S., Tatbul, N., Du, J.: Data ingestion for the connected world. In: CIDR (2017)
78. 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). DOI 10.14778/2831360.2831367. URL https://doi.org/10.14778/2831360.2831367
79. Meftah, S., Zhang, S., Veeravalli, B., Aung, K.M.M.: Revisiting the design of parallel stream joins on trusted execution environments. Algorithms **15**(6) (2022). DOI 10.3390/a15060183. URL https://www.mdpi.com/1999-4893/15/6/183
80. Miao, H., Jeon, M., Pekhimenko, G., McKinley, K.S., Lin, F.X.: Streambox-hbm: Stream analytics on high bandwidth hybrid memory. arXiv preprint arXiv:1901.01328 (2019)
81. 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)
82. Neumeyer, L., Robbins, B., Nair, A., Kesari, A.: S4: Distributed stream computing platform. In: 2010 IEEE International Conference on Data Mining Workshops, pp. 170–177. Ieee (2010)
83. Nguyen, T.M., Schiefer, J., Tjoa, A.M.: Sense & response service architecture (saresa): an approach towards a real-time business intelligence solution and its use for a fraud detection application. In: DOLAP'05, pp. 77–86. ACM (2005)
84. Noghabi, S.A., Paramasivam, K., Pan, Y., Ramesh, N., Bringhurst, J., Gupta, I., Campbell, R.H.: Samza: stateful scalable stream processing at linkedin. Proceedings of the VLDB Endowment **10**(12), 1634–1645 (2017)
85. Olson, M.A., Bostic, K., Seltzer, M.I.: Berkeley db. In: USENIX Annual Technical Conference, FREENIX Track, pp. 183–191 (1999)
86. 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)
87. 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)

88. 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, p. 788794. Association for Computing Machinery, New York, NY, USA (2013). DOI 10.1145/2480362.2480514. URL https://doi.org/10.1145/2480362.2480514
89. Oyamada, M., Kawashima, H., Kitagawa, H.: Data stream processing with concurrency control. SIGAPP Appl. Comput. Rev. **13**(2), 5465 (2013). DOI 10.1145/2505420.2505425. URL https://doi.org/10.1145/2505420.2505425
90. 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, p. 537554. USENIX Association, USA (2019)
91. 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). DOI 10.1109/icdew.2018.00025
92. Phua, C., Lee, V., Smith, K., Gayler, R.: A comprehensive survey of data mining-based fraud detection research. arXiv:1009.6119 (2010)
93. Poess, M., Rabl, T., Jacobsen, H.A., Caufield, B.: Tpc-di: The first industry benchmark for data integration. Proc. VLDB Endow. **7**(13), 13671378 (2014). DOI 10.14778/2733004.2733009. URL https://doi.org/10.14778/2733004.2733009
94. Ramnarayan, J., Mozafari, B., Wale, S., Menon, S., Kumar, N., Bhanawat, H., Chakraborty, S., Mahajan, Y., Mishra, R., Bachhav, K.: Snappydata: A hybrid transactional analytical store built on spark. In: Proceedings of the 2016 International Conference on Management of Data, SIGMOD '16, p. 21532156. Association for Computing Machinery, New York, NY, USA (2016). DOI 10.1145/2882903.2899408. URL https://doi.org/10.1145/2882903.2899408
95. 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, p. 495510. Association for Computing Machinery, New York, NY, USA (2016). DOI 10.1145/2882903.2882947. URL https://doi.org/10.1145/2882903.2882947
96. Rundensteiner, E.A., Ding, L., Sutherland, T., Zhu, Y., Pielech, B., Mehta, N.: Cape: Continuous query engine with heterogeneous-grained adaptivity. In: Proceedings of the Thirtieth international conference on Very large data bases-Volume 30, pp. 1353–1356. VLDB Endowment (2004)
97. Ryvkina, E., et al.: Revision processing in a stream processing engine: A high-level design. In: ICDE (2006)
98. Sahin, O.C., Karagoz, P., Tatbul, N.: Streaming event detection in microblogs: Balancing accuracy and performance. In: M. Bakaev, F. Frasincar, I.Y. Ko (eds.) Web Engineering, pp. 123–138. Springer International Publishing, Cham (2019)
99. Sattler, K.U.: Transactional stream processing on non-volatile memory,https://www.tu-ilmenau.de/dbis/research/active-projects/transactional-stream-processing/ (2019)
100. 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, p. 25232537. Association for Computing Machinery, New York, NY, USA (2020). DOI 10.1145/3318464.3380576. URL https://doi.org/10.1145/3318464.3380576
101. 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 - Volume 9827, DEXA 2016, p. 321335. Springer-Verlag, Berlin, Heidelberg (2016). DOI 10.1007/978-3-319-44403-1_20. URL https://doi.org/10.1007/978-3-319-44403-1_20
102. Shaikh, S.A., Kitagawa, H.: Streamingcube: Seamless integration of stream processing and olap analysis. IEEE Access **8**, 104,632–104,649 (2020). DOI 10.1109/ACCESS.2020.2999572
103. 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). URL https://www.usenix.org/conference/atc20/presentation/shillaker
104. Shuhao Zhang, He, J., Zhou, A.C., He, B.: Briskstream: Scaling Data Stream Processing on Multicore Architectures. In: Proceedings of the 2019 International Conference on Management of Data, SIGMOD '19, pp. 705–722. ACM, Amsterdam, Netherlands (2019). DOI 10.1145/3299869.3300067. URL https://doi.acm.org/10.1145/3299869.3300067
105. Silvestre, P.F., Fragkoulis, M., Spinellis, D., Katsifodimos, A.: Clonos: Consistent Causal Recovery for Highly-Available Streaming Dataflows, p. 16371650. Association for Computing Machinery, New York, NY, USA (2021). URL https://doi.org/10.1145/3448016.3457320
106. Stonebraker, M., Çetintemel, U., Zdonik, S.: The 8 requirements of real-time stream processing. SIGMOD Rec. **34**(4), 42–47 (2005). DOI 10.1145/1107499.1107504. URL http://doi.acm.org/10.1145/1107499.1107504
107. Stonebraker, M., Cetintemel, U.: Stream Processing, pp. 3771–3772. Springer New York, New York, NY (2018). DOI 10.1007/978-1-4614-8265-9_371. URL https://doi.org/10.1007/978-1-4614-8265-9_371
108. 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
109. Stonebraker, M., Rowe, L.A.: The design of postgres. ACM Sigmod Record **15**(2), 340–355 (1986)
110. Sturzrehm, H., Felber, P., Fetzer, C.: Tm-stream: An stm framework for distributed event stream processing. In: 2009 IEEE International Symposium on Parallel Distributed Processing, pp. 1–8 (2009). DOI 10.1109/IPDPS.2009.5161084
111. Tatbul, N.: Streaming data integration: Challenges and opportunities. In: 2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010), pp. 155–158 (2010). DOI 10.1109/ICDEW.2010.5452751
112. Tatbul, N.: Transactional Stream Processing, pp. 4205–4211. Springer New York, New York, NY (2018). DOI 10.1007/978-1-4614-8265-9_80704. URL https://doi.org/10.1007/978-1-4614-8265-9_80704
113. Tatbul, N., Zdonik, S., Meehan, J., Aslantas, C., Stonebraker, M., Tufte, K., Giossi, C., Quach, H.: Handling shared, mutable state in stream processing with correctness guarantees. IEEE Data Eng. Bull. **38**, 94–104 (2015)
114. Terry, D., Goldberg, D., Nichols, D., Oki, B.: Continuous queries over append-only databases. SIGMOD Rec. **21**(2), 321330 (1992). DOI 10.1145/141484.130333. URL https://doi.org/10.1145/141484.130333
115. Theodorakis, G., Koliousis, A., Pietzuch, P., Pirk, H.: Lightsaber: Efficient window aggregation on multi-core processors. Acm Sigmod (2020). DOI 10.1145/3318464.3389753
116. To, Q.C., Soto, J., Markl, V.: A survey of state management in big data processing systems. The VLDB Journal **27**(6), 847–872 (2018). DOI 10.1007/s00778-018-0514-9. URL https://doi.org/10.1007/s00778-018-0514-9
117. Tönjes, R., Barnaghi, P., Ali, M., Mileo, A., Hauswirth, M., Ganz, F., Ganea, S., Kjærgaard, B., Kuemper, D., Nechifor, S., et al.: Real time iot stream processing and large-scale

data analytics for smart city applications. In: poster session, EuCNC'14. sn (2014)
118. Toshniwal, A., Taneja, S., Shukla, A., Ramasamy, K., Patel, J.M., Kulkarni, S., Jackson, J., Gade, K., Fu, M., Donham, J., et al.: Storm@ twitter. In: Proceedings of the 2014 ACM SIGMOD international conference on Management of data, pp. 147–156 (2014)
119. Traub, J., Grulich, P.M., Cuéllar, A.R., Breß, S., Katsifodimos, A., Rabl, T., Markl, V.: Efficient window aggregation with general stream slicing. In: Edbt, pp. 97–108 (2019)
120. 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)
121. Vidyasankar, K.: Transactional properties of compositions of internet of things services. pp. 1–6 (2015). DOI 10.1109/ISC2.2015.7366218
122. Vidyasankar, K.: A transaction model for executions of compositions of internet of things services. Procedia Computer Science **83**, 195–202 (2016). DOI 10.1016/j.procs.2016.04.116
123. 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). DOI 10.1109/DEXA.2016.037
124. Vossen, G.: ACID Properties, pp. 1–3. Springer New York, New York, NY (2016). DOI 10.1007/978-1-4899-7993-3_831-2. URL https://doi.org/10.1007/978-1-4899-7993-3_831-2
125. Wang, D., Rundensteiner, E.A., Ellison III, R.T.: Active complex event processing over event streams. Proc. VLDB Endow. **4**(10), 634–645 (2011). DOI 10.14778/2021017.2021021. URL http://dx.doi.org/10.14778/2021017.2021021
126. Weikum, G., Vossen, G.: Dedication. In: G. Weikum, G. Vossen (eds.) Transactional Information Systems, The Morgan Kaufmann Series in Data Management Systems, p. vii. Morgan Kaufmann, San Francisco (2002). DOI https://doi.org/10.1016/B978-1-55860-508-4.50028-4. URL http://www.sciencedirect.com/science/article/pii/B9781558605084500284
127. Winter, C., Schmidt, T., Neumann, T., Kemper, A.: Meet me halfway: Split maintenance of continuous views. Proceedings of the VLDB Endowment **13**(11)
128. 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 (2017-03). DOI 10.14778/3067421.3067427
129. Wu, Y., Chan, C.Y., Tan, K.L.: Transaction healing: Scaling optimistic concurrency control on multicores. In: Proceedings of the 2016 International Conference on Management of Data, Sigmod '16, pp. 1689–1704. Acm, Acm, New York, NY, USA (2016). DOI 10.1145/2882903.2915202. URL http://doi.acm.org/10.1145/2882903.2915202
130. Wchter, H., Reuter, A.: The contract model (1991)
131. Zaharia, M., Das, T., Li, H., Hunter, T., Shenker, S., Stoica, I.: Discretized streams: Fault-tolerant streaming computation at scale. In: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, SOSP '13, p. 423438. Association for Computing Machinery, New York, NY, USA (2013). DOI 10.1145/2517349.2522737. URL https://doi.org/10.1145/2517349.2522737
132. 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). URL http://cidrdb.org/cidr2020/papers/p7-zeuch-cidr20.pdf
133. Zhang, S., He, B., Dahlmeier, D., Zhou, A.C., Heinze, T.: Revisiting the design of data stream processing systems on multi-core processors. In: 2017 IEEE 33rd International Conference on Data Engineering (ICDE), pp. 659–670. Ieee (2017-04). DOI 10.1109/icde.2017.119
134. 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). DOI 10.1109/ICDE48307.2020.00136
135. Zhang, S., Zhang, F., Wu, Y., He, B., Johns, P.: Hardware-conscious stream processing: A survey. SIGMOD Rec. **48**(4), 1829 (2020). DOI 10.1145/3385658.3385662. URL https://doi.org/10.1145/3385658.3385662
136. Zhang, Y., Chen, R., Chen, H.: Sub-millisecond stateful stream querying over fast-evolving linked data. In: Proceedings of the 26th Symposium on Operating Systems Principles, Sosp '17, pp. 614–630. Acm, New York, NY, USA (2017). DOI 10.1145/3132747.3132777. URL http://doi.acm.org/10.1145/3132747.3132777
137. Zhang, Y., Mueller, F.: Gstream: A general-purpose data streaming framework on gpu clusters. In: 2011 International Conference on Parallel Processing, pp. 245–254 (2011). DOI 10.1109/icpp.2011.22
138. Zhao, Y., Liu, Z., Wu, Y., Jiang, G., Cheng, J., Liu, K., Yan, X.: Timestamped state sharing for stream analytics. IEEE Transactions on Parallel and Distributed Systems pp. 1–1 (2021). DOI 10.1109/TPDS.2021.3073253
139. Zinkevich, M.: Online convex programming and generalized infinitesimal gradient ascent. In: Proceedings of the Twentieth International Conference on International Conference on Machine Learning, ICML'03, p. 928935. AAAI Press (2003)