A survey on the evolution of stream processing systems

Fragkoulis, Marios; Carbone, Paris; Kalavri, Vasiliki; Katsifodimos, Asterios

DOI
10.1007/s00778-023-00819-8

Publication date
2023

Document Version
Final published version

Published in
VLDB Journal

The VLDB Journal (2024) 33:507–541
https://doi.org/10.1007/s00778-023-00819-8

REGULAR PAPER

A survey on the evolution of stream processing systems

Marios Fragkoulis[1] · Paris Carbone[3,4] · Vasiliki Kalavri[5] · Asterios Katsifodimos[2]

Received: 10 August 2022 / Revised: 1 September 2023 / Accepted: 11 September 2023 / Published online: 22 November 2023

Abstract
Stream processing has been an active research field for more than 20 years, but it is now witnessing its prime time due to recent successful efforts by the research community and numerous worldwide open-source communities. This survey provides a comprehensive overview of fundamental aspects of stream processing systems and their evolution in the functional areas of out-of-order data management, state management, fault tolerance, high availability, load management, elasticity, and reconfiguration. We review noteworthy past research findings, outline the similarities and differences between the first ('00–'10) and second ('11–'23) generation of stream processing systems, and discuss future trends and open problems.

Keywords Stream processing · Fault-tolerance · Streaming analytics · Cloud applications

1 Introduction

Applications of stream processing technology have gone through a resurgence, penetrating multiple and very diverse industries. Nowadays, virtually all Cloud vendors offer first-class support for deploying managed stream processing pipelines, while streaming systems are used in a variety of use-cases that go beyond the classic streaming analytics (windows, aggregates, joins, etc.). Some examples include dynamic car-trip pricing, credit card fraud detection, predictive analytics, monitoring, and real-time traffic control. At the moment of writing, we are witnessing a trend towards using stream processors to build more general event-driven architectures [96], large-scale continuous ETL and analytics, and microservices [91].

✉ Asterios Katsifodimos
a.katsifodimos@tudelft.nl

Marios Fragkoulis
marios.fragkoulis@deliveryhero.com

Paris Carbone
parisc@kth.se ; paris.carbone@ri.se

Vasiliki Kalavri
vkalavri@bu.edu

1 Delivery Hero Research, Berlin, Germany

2 Delft University of Technology, Delft, The Netherlands

3 KTH Royal Institute of Technology, Stockholm, Sweden

4 RISE, Stockholm, Sweden

5 Boston University, Boston, USA

During the last 20 years, streaming technology has evolved significantly, under the influence of database and distributed systems. The notion of streaming queries was first introduced in 1992 by the Tapestry system [148], and was followed by lots of research on stream processing in the early 00 s. Fundamental concepts and ideas originated in the database community and were implemented in prototype systems such as TelegraphCQ [48], Stanford's STREAM, NiagaraCQ [51], Auroral/Borealis [12], and Gigascope [54]. Although these prototypes roughly agreed on the data model, they differed considerably on querying semantics [21, 33]. This research period also introduced various systems challenges, such as sliding-window aggregation [22, 107], fault-tolerance and high availability [30, 137], as well as load balancing and shedding [144]. This first wave of research was highly influential on commercial stream processing systems that were developed in the following years (roughly during 2004 – 2010), such as IBM System S, Esper, Oracle CQL/CEP and TIBCO. These systems focused—for the most part—on streaming window queries and complex event processing (CEP). This era of systems was mainly characterized by scale-up architectures, processing ordered event streams.

The second generation of streaming systems was a result of research that started roughly after the introduction of MapReduce [61] and the popularization of Cloud Computing. The focus shifted towards not only distributed, data-parallel processing engines and shared-nothing architectures on commodity hardware, but also on the design of systems that can support the mainstream MapReduce-like

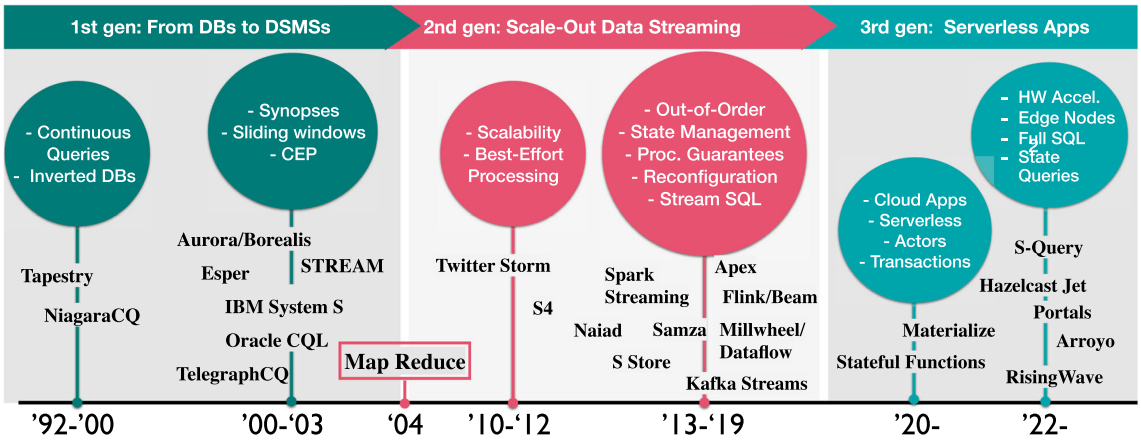


Fig. 1 An overview of the evolution of stream processing systems and respective domains of focus

user-defined function (UDF)-based programming models. As a result, systems such as Millwheel [14], Storm [3], Spark Streaming [164], and Apache Flink [37] first exposed primitives for expressing streaming computations as hard-coded dataflow graphs and transparently handled data-parallel execution on distributed clusters. The Google Dataflow model [16] re-introduced older ideas such as out-of-order processing [108] and punctuations [155], proposing a unified parallel processing model for streaming and batch computations. Stream processors of this era are converging towards fault-tolerant, scale-out processing of massive out-of-order streams.

Figure 1 presents a schematic categorization of influential streaming systems into three generations and highlights each era's domains of focus. Although the foundations of stream processing have remained largely unchanged over the years, stream processing systems have transformed into sophisticated and scalable engines, producing correct results in the presence of failures. Early systems and languages were designed as extensions of relational execution engines, with the addition of windows. Modern streaming systems have evolved in the way they reason about completeness and ordering (e.g., out-of-order computation) and have witnessed architectural paradigm shifts that constituted the foundations of processing guarantees, reconfiguration, and state management. At the moment of writing, we observe new variants of streaming systems focusing on integrating data streaming with either cloud services and serverless apps or edge computing and specialized hardware. The needs of IoT and edge computing have attracted the creation of more hardware-aware [167] and less resource-heavy systems [72] for event-based pipelines. At the same time, we observe a tendency towards full SQL support, both as an add-on capability in older systems [159], but also as a foundation for new cloud streaming database systems [4, 6, 8]. In the space of event-based cloud and serverless systems, we currently witness an integration of the actor and data-streaming paradigms, combining the flexibility of actors with the guarantees and performance of streaming engines [2, 13, 34, 140].

The evolution of stream processing systems has concerned the research community before [39, 59, 78]. This survey extends prior work by providing a systematic and comprehensive investigation of the evolution of fundamental functional areas rather than presenting the state of the field at a particular point in time. To the best of our knowledge, this is also the first attempt at understanding the underlying reasons why certain early techniques and designs prevailed in modern systems while others were abandoned. Further, by examining how ideas survived, evolved, and were often reinvented, we reconcile the terminology used by the different generations of streaming systems.

1.1 Contributions

With this survey paper, we make the following contributions:

- We summarize existing approaches to streaming systems design and categorize early and modern stream processors in terms of underlying assumptions and mechanisms.
- We compare early and modern stream processing systems with regard to out-of-order data management, state management, fault-tolerance, high availability, load management, elasticity, and reconfiguration.
- We highlight foundational works that have influenced today's streaming systems design.
- We establish a common nomenclature for fundamental streaming concepts, often described by inconsistent terms in different systems and communities.
- We provide a refined definition of availability for stream processing systems.

1.2 Related surveys and research collections

We view the following surveys as complementary to ours and recommend them to readers interested in diving deeper into a particular aspect of stream processing or those who seek a comparison between streaming technology and advances from adjacent research communities.

Cugola and Margara [55] provide a view of stream processing with regard to related technologies, such as active databases and complex event processing systems, and discuss their relationship with data-streaming systems. Further, they provide a categorization of streaming languages and streaming-operator semantics. The language aspect is further covered in another recent survey [82], which focuses on the languages developed to address the challenges in very large data streams. It characterizes streaming languages in terms of data model, execution model, domain, and intended user audience. Röger and Mayer [134] present an overview of recent work on parallelization and elasticity approaches of streaming systems. They define a general system model, which they use to introduce operator-parallelization strategies and parallelism-adaptation methods. Their analysis also aims at comparing elasticity approaches originating in different research communities. Hirzel et al. [83] present an extensive list of logical and physical optimizations for streaming query plans. They present a categorization of streaming optimizations in terms of their assumptions, semantics, applicability scenarios, and trade-offs. They also present experimental evidence to reason about profitability and guide system implementers in selecting appropriate optimizations. To, Soto, and Markl [150] survey the concept of state and its applications in big data management systems, also covering aspects of streaming state. Finally, Dayarathna and Perera [58] present a survey of the advances of the last decade with a focus on system architectures, use-cases, and hot research topics. They summarize recent systems in terms of their features, such as what types of operations they support, their fault-tolerance capabilities, their use of programming languages, and their best reported performance.

We refer the reader to Garofalakis et al. [69] for a comprehensive coverage of related topics, such as theoretical foundations of streaming data management and streaming algorithms. That collection focuses on major contributions of the first generation of streaming systems. It reviews basic algorithms and synopses, fundamental results in stream data mining, streaming languages and operator semantics, and a set of representative applications from different domains.

1.3 Survey organization

We begin by presenting the essential elements of the domain in Sect. 2. Then we elaborate on each of the important functionalities offered by stream processing systems: out-of-order data management (Sect. 3), state management (Sect. 4), fault tolerance and high availability (Sect. 5), and load management, elasticity, and reconfiguration (Sect. 6). Each one of these sections contains a *first-generation vs. second-generation* discussion that compares early to contemporary approaches, and a summary of open problems. In Sect. 7, we summarize design considerations for stream systems along each of these dimensions, we highlight our major findings, and we discuss future prospects.

Summarizing tables Roughly in every section, this survey provides a summarizing table, containing a set of systems, as well as their characteristics or features discussed by the respective section. Note that the set of systems that are considered and compared against each other in each section may differ. This difference is because a system (or paper) may not support a certain characteristic. For instance, Apache Storm is not part of Table 1, as it does not consider timestamps or order of event arrivals.

2 Preliminaries

In this section, we provide necessary background and explain fundamental stream processing concepts on which the rest of this survey relies. We discuss the key requirements of a streaming system, introduce the basic streaming data models, and give a high-level overview of the architecture of early and modern streaming systems.

2.1 Requirements of streaming systems

A data stream is a data set that is produced incrementally over time, rather than being available in full before its processing begins [69]. Data streams are real-time data that might be unbounded. Therefore, stream processing systems can neither store the entire stream in an accessible way nor can they control the data arrival rate or order. In contrast to traditional data-management infrastructure, streaming systems have to process elements on-the-fly using limited memory. Stream elements arrive continuously and bear at least one timestamp, which can be assigned at the source or on arrival.

A streaming query ingests events and produces results in a continuous manner, using a single pass or a limited number of passes over the data. Streaming query processing is challenging for multiple reasons. First, continuously producing updated results might require storing historical information about the stream seen so far in a compact representation that can be queried and updated efficiently. Such summary representations are known as *sketches* or *synopses*. Second, in order to handle high input rates, certain queries might not afford to continuously update indexes and materialized views. Third, stream processors cannot rely on the assump-

tion that state can be reconstructed from associated inputs. In contrast to batch queries that are short-lived and process fixed-size inputs, streaming queries are long-running. As a result, reconstructing their intermediate state may require re-processing the entire stream history. Instead, to achieve acceptable performance, streaming operators need to leverage incremental computation.

The aforementioned characteristics of data streams and continuous queries provide a set of unique requirements for streaming systems, other than the evident performance ones of low latency and high throughput. Given the lack of control over the input order, a streaming system needs to produce correct results when receiving out-of-order and delayed data (cf. Sect. 3). It needs to implement mechanisms that estimate a stream's progress and reason about result completeness. Further, the long-running nature of streaming queries demands that streaming systems manage accumulated state (cf. Sect. 4) and guard it against failures (cf. Sect. 5). Finally, having no control over the data input rate requires stream processors to be adaptive so that they can handle workload variations without sacrificing performance (cf. Sect. 6).

2.2 Streaming data models

There exist many theoretical streaming data models, mainly serving the purpose of studying the space requirements and computational complexity of streaming algorithms and understanding which streaming computations are practical. For instance, a stream can be modeled as a dynamic one-dimensional vector [69]. The model defines how this dynamic vector is updated when a new element of the stream becomes available. While theoretical streaming data models are useful for algorithm design, early stream processing systems instead adopted extensions of the *relational* data model. Recent streaming dataflow systems, especially those influenced by the MapReduce philosophy, place the responsibility of data stream modeling on the application developer.

2.2.1 Relational streaming model

In the relational streaming model, as implemented by first-generation systems [11, 12, 20, 31, 48, 54], a stream is interpreted as describing a changing relation over a common schema. Notably, these systems assumed that stream elements bear timestamps or sequence numbers, allowing for defining windows over streams. Streams themselves are either produced by external sources and update relation tables or are produced by continuous queries and update materialized views. An operator outputs event streams that describe the changing view computed over the input stream according to the relational semantics of the operator. Thus, both the semantics and the relation schema are imposed by the system.

2.2.2 Dataflow streaming model

The dataflow streaming model is represented as a dataflow graph, that is, a directed graph G = (E, V), where vertices in V represent operators and edges in E denote data streams. The dataflow streaming model, as implemented by systems of the second generation [16, 37, 164], does not impose any strict schema or semantics on the input stream elements, other than the presence of a timestamp. While some systems, such as Naiad [120], require that all stream elements bear a logical timestamp, other systems, such as Flink [37] and Dataflow [16], expect the declaration of a *time domain*. Applications can operate in one of three modes: (i) *event* (or application) time is the time when events are generated at the sources, (ii) *ingestion* time is the time when events arrive at the system, and (iii) *processing* time is the time when events are processed in the streaming system. Modern dataflow streaming systems can ingest any type of input stream, irrespectively of whether its elements represent additions, deletions, replacements or deltas. The application developer is responsible for imposing the semantics and writing the operator logic to update state accordingly and produce correct results. Designating keys and values is also usually not required at ingestion time, however, keys must be defined when using certain data-parallel operators.

3 Managing event order and timeliness

Event order is typically enforced using an event timestamp. An event timestamp is contained in each record and denotes the event time when the record's data were generated at the source. The event timestamps dictate the order of data in the stream and they are considered part of the stream's semantics [113]. Depending on the computations to perform, a streaming system may have to process stream records in a certain order to provide semantically correct results [142]. However, in the general case, a stream's records arrive out of order [104, 155] for reasons explained in Sect. 3.1.
Out-of-order data records [142, 153] arrive at a streaming system from an input source after records with later event time timestamps.

In the rest of the paper, we use the terms *disorder* [113] and *out-of-order* [14, 108] interchangeably to refer to the disturbance of order in a stream's data records. Reasoning about order and managing disorder are fundamental considerations for the operation of streaming systems.

We highlight the causes of disorder in Sect. 3.1, we clarify the relationship between disorder in a stream's records and processing progress in Sect. 3.2, and we outline the two key system architectures for managing out-of-order data in Sect. 3.3. Then, we describe the consequences of disorder in Sect. 3.4 and present mechanisms for managing disorder in

Sect. 3.5. Finally, in Sect. 3.6, we discuss the differences of out-of-order data management in early and modern systems and we present open problems in Sect. 3.7.

3.1 Causes of disorder

Disorder in data streams may be due to stochastic factors that are external to a streaming system or to the operations taking place inside the system.

The most common external factor that introduces disorder to streams is the network [99, 142]. Depending on the network's reliability, bandwidth, and load, the routing of some stream records can take longer to complete compared to the routing of others, leading to a different arrival order in terms of event time in a streaming system. Even if the order of records in an individual stream is preserved, ingestion from multiple sources, such as sensors, even with synchronized clocks, typically results in a disordered collection of records. Notably, if the sources do not feature synchronized clocks, which is often the case, the generated timestamps of records may not correspond to the real time of production. Consequently, if the timestamps used to specify order do not represent event time, this cause of disorder is impossible to fix.

External factors aside, specific operations on streams break record order. First, join processing takes two streams and produces a shuffled combination of the two, since a parallel join operator repartitions the data according to the join attribute [157] and outputs join results by order of match [77, 90]. Second, windowing based on an attribute different to the ordering attribute reorders the stream [54]. Third, data prioritization [130, 156] by using an attribute different to the ordering one also changes the stream's order. Finally, the union operation on two unsynchronized streams yields a stream with all records of the two input streams interleaving each other in random order [12].

3.2 Disorder and processing progress

In order to manage disorder, streaming systems need to detect and measure processing progress. *Progress* regards how much the processing of a stream's records has advanced over time. Processing progress can be defined and quantified with the aid of an attribute A of a stream's records that orders the stream. The processing of the stream progresses when the smallest value of A among the unprocessed records increases over time [108]. A then is a *progressing attribute* and the smallest (unprocessed) value of A per se, is a measure of progress because it denotes how far in the processing of records the system has reached since the beginning. Processing progress can be quantified using more than one attribute of a stream's records.

A streaming system's capacity to track processing progress enables the system to decide a *lateness bound* in order to quantify the level of disorder that a streaming system can accept while it makes processing progress. To elaborate, lateness bound is a measure of how late, with respect to the rest of the stream, data can be to be accepted for processing by a streaming system. It typically corresponds to a time measure or count of stream records. For example, a lateness bound of 1 s makes a streaming system's window operator to compute the window's function 1 s later than the window's time span. The time extension allows the inclusion of out-of-order records to the window computation.

3.3 System architectures for managing disorder

Two main architectural archetypes have influenced the design of streaming systems with respect to managing disorder: (i) in-order processing systems [12, 21, 54, 142], and (ii) out-of-order processing systems [14, 37, 108, 120].

In-order processing systems manage disorder using one of three main strategies. First, they can assume input streams are ordered and discard late data [12, 21, 54]. Second, they can buffer and reorder input streams up to a lateness bound to enforce stream order [12, 21, 142]. Then, they forward the reordered records for processing and clear the corresponding buffers. Finally, they can admit late data without reordering up to a lateness bound [12]. Although the last strategy supports disordered input, it entails that downstream operators should also quantify and enforce a lateness bound independently. In order to do so effectively, they need to track processing progress, which is hard to achieve without a system-wide progress tracking mechanism. Therefore, in-order systems commonly enforce and preserve stream order and use it to deduce processing progress.

In out-of-order processing systems, operators or a global authority produce progress information using any of the mechanisms detailed in Sect. 3.5.1, and propagate it to the dataflow graph. The information typically reflects the oldest unprocessed record in the system and establishes a lateness bound for admitting and/or processing out-of-order records. In contrast to in-order systems, records are processed without delay in their arrival order, as long as they do not exceed the lateness bound. Even when in-order systems allow disorder, they apply it on an operator basis, which is inefficient and uncertain without a system-wide progress-tracking mechanism.

3.4 Effects of disorder

In unbounded data processing, disorder can impede progress [108] or lead to wrong results if ignored [142].

Disorder affects processing progress when the operators that comprise the topology of the computation require

ordered input. Various implementations of *join* and *aggregate* rely on ordered input to produce correct results [12, 142]. When operators in in-order systems receive out-of-order records, they typically reorder them prior to including them in the window they belong. Reordering, however, imposes processing overhead, memory space overhead, and latency. Out-of-order systems, on the other hand, track progress and process data in whatever order they arrive, up to the lateness bound, and remove processing state related to data earlier than the lateness bound. As a sidenote, order-insensitive operators [12, 108, 142], such as *apply, project, select, dupelim*, and *union*, are agnostic to disorder in a stream and produce correct results even when presented with disordered input.

Ignoring out-of-order data can lead to incorrect results in many use cases, since the output is computed on an incomplete subset of the input data. Thus, a streaming system needs to be capable of processing out-of-order data and incorporate their effect to the computation. Embracing data disorder is important for unblocking the processing of blocking operators and purging corresponding computation state. We next discuss how it can be achieved with disorder management mechanisms.

3.5 Mechanisms for managing disorder

In this section, we elaborate on influential mechanisms for managing disorder in unbounded data, namely slack [12], heartbeats [142], low-watermarks [108], pointstamps [120], and triggers [16]. These mechanisms quantify a lateness bound using a metric, such as time, and are leveraged by streaming systems to track processing progress. If records arrive after the lateness bound expires, triggers can be used to update computation results in *revision processing* [11].

We also discuss punctuations [155], a generic mechanism for communicating information across the dataflow graph, that has been heavily used as a vehicle in managing disorder. Punctuations are metadata annotations embedded in data streams. A punctuation is itself a stream record, which consists of a set of patterns each identifying an attribute of a stream data record.

3.5.1 Progress tracking mechanisms

We detail and depict slack, heartbeats, low-watermark, and pointstamps. Figure 2 showcases the differences between slack, heartbeats, and low-watermarks. The figure depicts a simple aggregation operator that counts records in 4-second event-time tumbling windows starting at t=1. The operator awaits for some indication that event time has advanced past the end timestamp of a window, so that it computes and outputs an aggregate per window. The indication varies according to the progress-tracking mechanism. The input to this operator are seven records containing only a timestamp from t=1 to t=7. The timestamp signifies the event time in seconds that the record was produced in the input source. Each record contains a different timestamp and all records are dispatched from a source in ascending order of timestamp. Due to network latency, the records may arrive to the streaming system out of order.

3.5.1.1 Slack is a simple mechanism that involves waiting for out-of-order data for a fixed amount of a certain metric. Slack originally denoted the number of records intervening between the actual occurrence of an out-of-order record and the position it would have in the input stream if it arrived on time. However, it can also be quantified in terms of elapsed time. Essentially, slack marks a fixed grace period for late records.

Figure 2a presents the slack mechanism. In order to accommodate out-of-order records, the operator admits out-of-order records up to *slack=1*. Thus, the operator, having admitted records with t=1 and t=2 (not depicted in the figure), will receive the record with t=4. The timestamp of the record coincides with the maximum timestamp of the first window for interval [0, 4). Normally, this record would cause the operator to close the window and compute and output the aggregate, but because of the slack value, the operator will wait to receive one more record. The next record with t=3 belongs to the first window and is included there. At this point, slack also moves forward and this event finally triggers the window computation, which outputs C=3 for t=[1, 2, 3]. In contrast, the operator will not accept t=5 at the tail of the input, because it arrives two records after its natural order and is not covered by the slack value.

3.5.1.2 A heartbeat is an alternative to slack that consists of an external signal carrying progress information about a data stream. It contains a timestamp indicating that all succeeding stream records will have a timestamp larger than the heartbeat's timestamp. Heartbeats can either be generated by an input source or deduced by the system by observing environment parameters, such as network latency bound, application clock skew between input sources, and out-of-order data generation [142].

Figure 2b depicts the heartbeat mechanism. An input manager buffers and orders the incoming records by timestamp. The number of records buffered, two in this example (t = 5, t = 6), is of no importance. The source periodically sends a heartbeat to the input manager, i.e. a signal with a timestamp. Then, the input manager dispatches to the operator all records with timestamp less or equal to the timestamp of the heartbeat in ascending order. Although heartbeats are external to the input stream, in the Figure we position heartbeats (h = 2, h = 4) alongside input records just to show the order of events. For instance, when the heartbeat h = 2, which carries a timestamp t = 2, arrives in the input manager (not shown in the figure), the input manager dispatches the records with timestamp t =

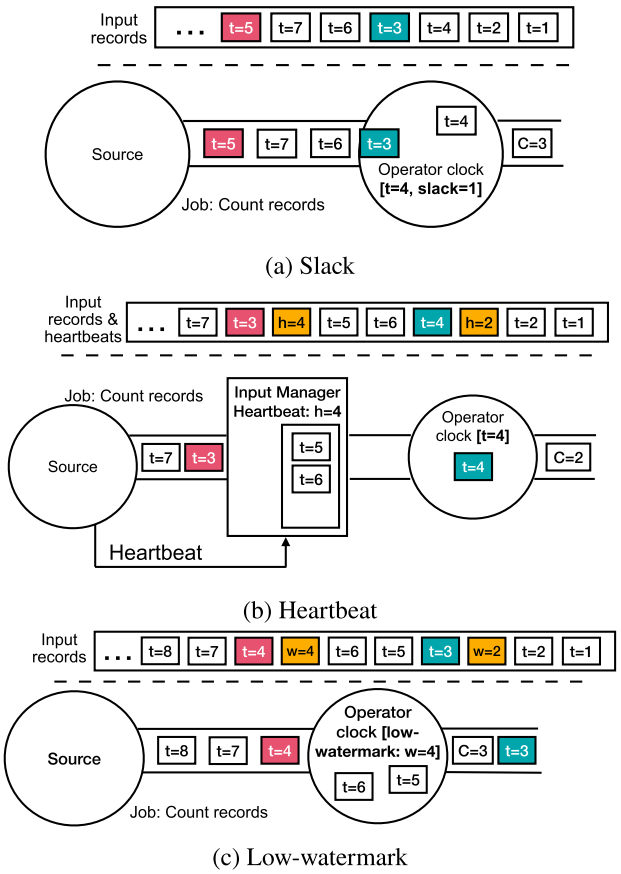


Fig. 2 Mechanisms for managing disorder

1 and t = 2 to the operator. The input manager then receives records with t = 4, t = 6, and t = 5 in this order and puts them in the right order. When the heartbeat h = 4, which carries a timestamp t = 4, arrives, the input manager dispatches the record with timestamp t = 4 to the operator. This record triggers the computation of the first window for interval [0, 4). The operator outputs C = 2 counting two records with t = [1, 2] not depicted in the figure. The input manager ignores the incoming record with timestamp t = 3, as it is older than the latest heartbeat with timestamp t = 4.

3.5.1.3 The low-watermark for an attribute *A* of a stream is the lowest value of *A* within a certain subset of the stream. Thus, future records will typically bear a higher value than the current low-watermark for the same attribute. Often, *A* is a record's event time timestamp. The mechanism is used by a streaming system to track processing progress via the low-watermark for *A*, to admit out-of-order data whose attribute *A*'s value is not smaller than the low-watermark. Further, it can be used to remove state that is maintained for *A*, such as the corresponding hash table entries of a streaming join computation. In addition, watermarks have also been leveraged to accurately estimate stream progress for unblocking windowed computations that lead to throughput and latency performance improvements via optimized scheduling [63].

Figure 2c presents the low-watermark mechanism, which signifies the oldest pending work in the system. Here, punctuations carrying the low-watermark timestamp decide when windows will be closed and computed. After receiving two records with t = 1 and t = 2, the corresponding low-watermark for t = 2 (which is propagated downstream), and record t = 3, the operator receives record t = 5. Since this record carries an event time timestamp greater or equal to 4, which is the end timestamp of the first window, it could be the one to cause the window to fire or close. However, this approach would not account for out-of-order data. Instead, the window closes when the operator receives the low-watermark with t = 4. At this point, the operator computes C = 3 for t = [1, 2, 3] and assigns records with t = [5, 6] to the second window with interval [4, 8). The operator will not admit record t = 4 because it is not greater (more recent) than the current low-watermark value t = 4.

3.5.1.4 Comparison among heartbeats, slack, low-watermark, and punctuations. Heartbeats and slack are both external to a data stream. Heartbeats are signals communicated from an input source to a streaming system's ingestion point. Differently from heartbeats, which are an internal mechanism of a streaming system hidden from users, slack is part of the query specification provided by users [12].

Heartbeats and low-watermarks are similar in terms of progress-tracking logic. However, one important difference sets them apart. The low-watermark generalizes the concept of the oldest value, which signifies the current progress point, to any progressing attribute of a stream record, not just timestamps.

In contrast to heartbeats and slack, *punctuations* provide a channel for communicating progress information such as a record attribute's low-watermark produced by an operator [108], event time skew [142], or slack [12]. Thus, punctuations can convey which data no longer appear in an input stream; for instance, the data records with smaller timestamps than a specific value. Punctuations are useful in other functional areas of a streaming system as well, such as state management, monitoring, and flow control.

3.5.1.5 Pointstamps, like punctuations, are embedded in data streams, but a pointstamp is attached to each stream data record as opposed to a punctuation, which forms a separate record. A pointstamp, is a pair of timestamp and location that positions data records on a vertex or edge of the dataflow graph at a specific point in time. An unprocessed record p at location l with timestamp t could result in another unprocessed record p' at location l' with timestamp t' when p can arrive at location l' before or at timestamp t'. An unprocessed record p at location l with timestamp t is in the *frontier* of processing progress when no other unprocessed records

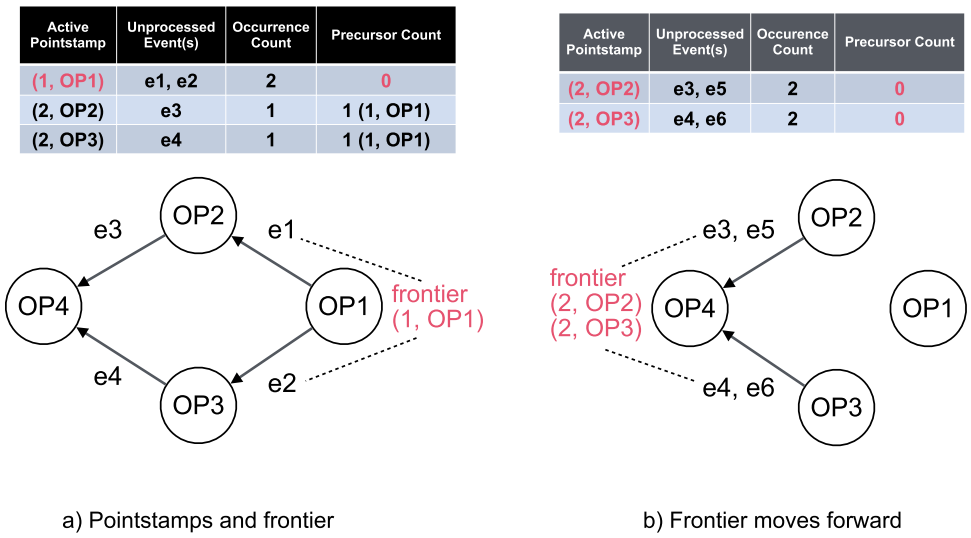


Fig. 3 High-level workflow of pointstamps and frontier

could result in p. Thus, when the aforementioned record p is processed, the frontier moves on. The system enforces that future records will bear a greater timestamp than the records that generated them. This modeling of processing progress traces the course of data records on the dataflow graph with timestamps and tracks the dependencies between unprocessed events in order to compute the current frontier. Under this light, the function of a frontier is similar to a low-watermark.

The example shown in Fig. 3 showcases how pointstamps and frontiers work. The example in Fig. 3a includes three active pointstamps. Poinstamps are active when they correspond to one or more unprocessed events. Pointstamp (1, OP1) is in the frontier of processing progress, because there are no active pointstamps that could result in pointstamp (1, OP1). The number of pointstamps that can result in another pointstamp are specified by the precursor count. Consequently, the precursor count of pointstamp (1, OP1) is zero.

In the frontier, notifications for unprocessed events can be delivered. Accordingly, unprocessed events e1 and e2 can be delivered to OP2 and OP3 respectively. The occurrence count is 2 because both events e1 and e2 bear the same pointstamp. Looking at this snapshot of the dataflow graph, it is easy to see that pointstamp (1, OP1) could result in pointstamps (2, OP2) and (2, OP3). Therefore, the precursor count of each of the latter two pointstamps is 1.

A bit later, as Fig. 3b depicts, events e1 and e2 are delivered to OP2 and OP3, respectively. Their processing results in the generation of new events e5 and e6, which bear the same pointstamp as unprocessed events e3 and e4, respectively. Since there are no more unprocessed events with timestamp 1, and the precursor counts of pointstamps (2, OP2) and (2, OP3) are 0, the frontier moves on to these active pointstamps. Consequently, all four event notifications can be delivered. The obsolete pointstamp (1, OP1) is removed from its location, since it corresponds to no unprocessed events. The same will happen to pointstamps (2, OP2) and (2, OP3), following the delivery of events e3, e4, e5, and e6. Although this example is made simple for demonstration purposes, the progress tracking mechanism has the power to track the progress of arbitrary iterative and nested computations.

3.5.2 Tracking progress of out-of-order data in cyclic queries

Cyclic queries require special treatment for tracking progress. A cyclic query always contains a binary operator, such as a join or a union. The output produced by the binary operator meets a loop further in the dataflow graph that connects back to one of the binary operator's input channels. In a progress model that uses punctuations for instance, the binary operator forwards a punctuation only when it appears in both of its input channels, otherwise it blocks waiting for both to arrive. Since one of the binary operator's input channels depends on its own output channel, a stall is inevitable.

Chandramouli et al. [46] propose an operator for detecting progress in cyclic streaming queries on the fly. The operator introduces a speculative punctuation in the loop that is derived from the passing events' timestamps. While the punctuation flows in the loop, the operator observes the stream's records to validate its guess. When the speculative punctuation re-enters the operator and is validated, it becomes a regular punctuation that carries progress information downstream. Then, a new speculative punctuation is generated and is fed to the loop. By combining a dedicated operator, speculative output, and punctuations, this work is able to track progress and tolerate disorder in cyclic streaming queries. The approach entails that a loop consists of operators that a) are able to make correct speculations, b) make forward progress, and c) will not block due to a punctuation. Oper-

ators in the loop are able to revise the original speculative punctuation. The approach can be applied in systems that provide speculative output for strongly convergent queries, which provide finite results for finite inputs with finite derivations for each result.

In Naiad [120, 121], the general progress-tracking model features logical, i.e. processing-time, multidimensional timestamps attached to events. Each timestamp consists of the input batch to which an event belongs and an iteration counter for each loop the event traverses. As in Chandramouli et al. [46], Naiad supports cyclic queries by utilizing a special operator. However, the operator is used to increment the iteration counter of events entering a loop. To ensure progress, the system allows event handlers to dispatch only messages with larger timestamps than the timestamp of events being currently processed. This restriction imposes a partial order over all pending events. The order is used to compute the earliest logical time of events' processing completion in order to deliver notifications for producing output. Naiad's progress-tracking mechanism enables a) scalability by providing efficient delivery of notifications to dataflow nodes and b) incremental computation that avoids redundant work.

3.5.3 Revision processing

Revision processing is the update of computations in face of late, updated, or retracted data, which require the modification of previous outputs in order to provide correct results. Revision processing made its debut in Borealis [11]. From there on, it has been combined with in-order processing architectures [45, 122], as well as out-of-order processing architectures [16, 17, 31, 99]. In some approaches, revision processing works by *storing* incoming data and *revising* computations in face of late, updated, or retracted data [16, 17, 31]. Other approaches *replay* affected data, *revise* computations, and propagate the revision messages to update all affected results up to the present [11, 122, 135]. Finally, a third line of approaches maintain multiple *partitions*, i.e. divisions of the data, that capture events with different levels of lateness and *consolidate* partial results [45, 99].

Notably, revision processing introduces a set of challenges, namely operator support, performance overhead, and limitations regarding the recency of data that can be revised. Revision processing can add non-trivial processing overhead to a streaming system, especially for downstream stateful operators that will need to recompute entire windows. The overhead can increase sharply with the increase of revision data. Finally, since it is impossible to hold all original input data eternally to be able to revise them, revision processing is restricted to a subset containing the most recent data.

3.5.3.1 Store and revise. Microsoft's CEDR [31], StreamInsight [17], and Google's Dataflow [16] buffer or store stream data and process late events, updates, and deletions incrementally by revising the captured values and updating the computations.

The dataflow model [16] divides the concerns for out-of-order data into three dimensions: the event time when late data are processed, the processing time when corresponding results are materialized, and how later updates relate to earlier results. The mechanism that decides the emission of updated results and how the refinement will happen is called a *trigger*. Triggers are signals that cause a computation to be repeated or updated when a set of specified rules fire.

One important rule regards the arrival of late input data. Triggers ensure output correctness by incorporating the effects of late input into the computation results. Triggers can be defined based on watermarks, processing time, data arrival metrics, and combinations of those; they can also be user-defined. Triggers support three refinement policies, *accumulating* where new results complement older ones, *discarding* where new results overwrite older ones, and *accumulating and retracting* where new results overwrite older ones and older results are retracted. Retractions, or compensations, are also supported in StreamInsight [17].

3.5.3.2 Replay and revise. Dynamic revision [11] and *speculative processing* [122] replay an affected past data subset when a revision record is received. An optimization of this scheme relies on two revision processing mechanisms, upstream processing and downstream processing [135]. Both are based on a special-purpose operator, called a *connection point*, that intervenes between two regular operators and stores records output by the upstream operator. According to the upstream revision processing, an operator downstream from a connection point can ask for a set of records to be replayed, so that it can calculate revisions based on old and new results. Alternatively, the operator can ask from the downstream connection point to retrieve a set of output records related to a received revision record. Under circumstances, the operator can calculate correct revisions by incorporating the net effect of the difference between the original record and its revised one to the old result.

Dynamic revision emits delta revision messages, which contain the difference of the output between the original and the revised value. This approach keeps the input message history at an operator in the connection point of its input queue. Since keeping all messages is infeasible, there is a bound in the history of messages kept. Messages that go further back from this bound cannot be replayed and, thus, revised. Dynamic revision differentiates between stateless and stateful operators. A stateless operator will evaluate both the original (t) and the revised message (t'), emitting the delta of their output. For instance, if the operator is a filter, t is true and t' is not, then the operator will emit a deletion message for t. A stateful operator, on the other hand, has to

process many messages in order to emit an output. Thus, an aggregation operator has to re-process the whole window for both a revised message and the original message contained in that window in order to emit revision messages. Dynamic revision is implemented in Borealis.

Speculative processing, on the other hand, applies state snapshot recovery if no output has been produced for a disordered input stream. Otherwise, it retracts all produced output. In speculative processing, because revision processing is opportunistic, no history bound is set.

3.5.3.3 *Partition and consolidate*. *Order-independent processing* [99] and *impatience sort* [45] partially process independent data partitions in parallel and consolidate partial results. The order-independent processing approach opens a new partition when a record is received after its corresponding progress indicator. A new query plan instance processes this partition using standard out-of-order processing techniques. The impatience sort approach features an online sorting operator, which incrementally orders the input arriving at each partition, so that it is emitted in order. The approach uses punctuations to bound the disorder, as opposed to order-independent processing, which can handle events arriving arbitrarily late.

In order-independent processing, partitioning is left for the system to decide, while in impatience sort it is specified by the user. In order-independent processing, records that are too old to be considered in their original partition are included in the partition which has the record with the closest data. When no new data enter an ad-hoc partition for a long time, the partition is closed and destroyed by means of a heartbeat. Ad-hoc partitions are window-based; when an out-of-order record is received that does not belong to one of the ad-hoc partitions, a new ad-hoc partition is introduced. An out-of-order record with a more recent timestamp than the window of an ad-hoc partition causes that partition to flush results and close. Order-independent processing is implemented in Truviso [99].

On the contrary, in impatience sort, users specify reorder latencies, such as $1ms$, $100ms$, and $1s$, that define the buffering time for ingesting and sorting out-of-order input records. According to the specified reorder latencies, the system creates different partitions of in-order input streams. After sorting, a union operator merges and synchronizes the output of a partition P with the output of a partition L that features lower reorder latency than P. Thus, the output will incorporate partial results provided by L with later updates that P contains. This way, applications that require fast but partial results can subscribe to a partition with small reorder latency. By letting applications choose the desired extent of reorder latency, this design provides for different trade-offs between completeness and freshness of results. Impatience sort is implemented in Microsoft Trill.

3.6 First generation versus second generation

The importance of event order in data stream processing was obvious since its early days [27], leading to the first wave of simple intuitive solutions. Early approaches involved buffering and reordering arriving records using some measure for adjusting the frequency and lateness of data dispatched to a streaming system in order [12, 48, 142]. A few years later, the introduction of out-of-order processing [108] improved throughput, latency, and scalability for window operations by keeping track of processing progress without ordering records. In the meantime, revision processing [11] was proposed as a strategy for dealing with out-of-order data reactively. In the years that followed, in-order, out-of-order, and revision processing were extensively explored, often in combination with one another [16, 17, 31, 99, 122]. Modern streaming systems implement a refinement of these original concepts. Interestingly, concepts devised several years ago, like low-watermarks, punctuations, and triggers, which advance the original revision processing, were popularized recently by streaming systems such as Millwheel [14] and the Google Dataflow model [16], Flink [37], and Spark [23]. Table 1 presents how both first generation and modern streaming systems implement out-of-order data management.

3.7 Open problems

Managing data disorder entails architecture support and flexible mechanisms. There are open problems at both levels.

First, which architecture is better is an open debate. Although many of the latest streaming systems adopt an out-of-order architecture, opponents point to the architecture's implementation and maintenance complexity. In addition, revision processing, which is used to reconcile out-of-order records is daunting at scale because of the challenging state size. On the other hand, in-order processing is resource-hungry and loses events if they arrive after the disorder bound.

Second, applications receiving data streams from different sources may need to support multiple notions of event time, one per incoming stream, for instance. However, streaming systems to date cannot support multiple time domains.

Finally, data streams from different sources may have disparate latency characteristics that render their watermarks unaligned. Tracking the processing progress of those applications is challenging for today's streaming systems.

4 State management

State captures all internal side-effects of a continuous stream computation. The state includes, for example, active windows, buckets of records, partial or incremental aggregates

Table 1 Event order management in streaming systems

System	Architecture			Progress-tracking			
	In-order	Out-of-order	Revision	Mechanism	Communication	Disorder bound metric	Revision approach
Aurora [12, 52]	✓			Slack	User config	Number of records	–
STREAMS [142]	✓			Heartbeat	Signal to input manager	Timestamp (event time skew, network latency, out-of-order bound)	–
Borealis [11]		✓	✓	History bound	System config	Number of records or time units	Replay past data, enter revised values, issue delta output
Gigascope [87]		✓		Low-watermark	Punctuation	Timestamp	–
Timestream [129]	✓			Low-watermark	Punctuation	Timestamp	–
Millwheel [14]		✓		Low-watermark	Signal to central authority	Timestamp	–
Naiad [120]		✓	✓	Pointstamp	Part of data record	Multidimensional timestamp	Incremental processing of updated data via structured loops
Trill [44]	✓			Low-watermark	Punctuation	Timestamp	–
Streamscope [109]	✓			Low-watermark	Punctuation	Timestamp; sequence number	–
Samza [124]		✓	✓	Low-watermark	Punctuation	Timestamp	Find, roll back, recompute affected input windows
Flink [37]		✓	✓	Low-watermark	Punctuation	Timestamp	Store and Recompute/Revise
Dataflow [16]		✓	✓	Low-watermark	Signal to central authority	Timestamp	Discard and recompute; accumulate and revise; custom
Spark [23]		✓	✓	Slack	User config	Number of seconds	Discard and recompute; accumulate and revise

used in an application, as well as possibly some user-defined variables created and updated during the execution of a stream pipeline. This section provides an overview of known approaches, current directions, and discussions of open problems in the context of state management.

4.1 Managing stream processing state

The area of stream state management is still an active research field, incorporating methods on how state should be declared in a stream application, as well as how it should be scaled and partitioned. Furthermore, state management considers state persistence methods for long-running applications, and defines system guarantees and properties to maintain whenever a change in the system occurs, such as failures or reconfiguration, e.g., changing the degree of parallelism of a given operator.

State partitioning State needs to be partitioned in order to parallelize computations across different keys. For instance, consider the count of orders of a given customer, during a time window of a month. The state of the customer in this case is the count of orders. A parallel streaming application would partition the different customers on multiple workers.

State changes during reconfiguration Streaming applications have an inherent need to run continuously over long periods of time. However, during long executions, a lot of issues may arise. First, the statistics of the input data (e.g., distribution of keys) or the input throughput of the incoming stream may change. The opposite problem, i.e., allocating more resources than needed, is also problematic because it leads to excessive resource utilization. Finally, failures can happen: the probability of servers failing (e.g., disk or network failures) is very high for an application that processes data continuously for days or months. These failures, require state reconfiguration: typically specific keys, need to be assigned to different nodes to balance the workload, avoiding over- or under-utilization of resources. We review approaches to state reconfiguration in Sect. 6.3.2 and discuss how state management decisions may affect the design of reconfiguration mechanisms in Sect. 7.1.

Most of these research issues were introduced in part within the context of early streaming processing systems, such as Aurora and Borealis [41]. Specifically, Borealis set the foundations in formulating many of these problems, such as the need for embedded state, persistent store access, and failure recovery protocols. In Table 2, we categorize data stream processing systems according to their respective state management approaches. The rest of this section offers an overview of each of the topics in stream state management along with past and currently employed approaches.

4.2 Programmability and responsibility

State in a programming model can be either implicitly or explicitly declared and used. We define *state programmability* as the ability of a streaming system to allow its users to declare and manipulate state. For example, state can be a local variable within a stateful `map` function, storing a counter. Programmability in state requires support from the underlying execution engine, a feature that directly affects the engine's complexity. Different system trends have influenced both how state can be exposed in a data stream programming model, as well as how it should be scoped and managed. In this section, we discuss different approaches and their trade-offs. As shown in Table 2, most systems allow their users to declare custom, user-defined state. Those that do not, focus more on providing a high-level SQL interface on top of a dataflow processor allowing only their internal operators to define and use state within stateful operations (e.g., joins, windows, aggregates).

State management responsibility An orthogonal aspect to programmability is state management *responsibility*, which entails the obligation of maintaining state by either the user or the system. State maintenance includes allocating memory or disk space for storing application variables, persisting changes to disk and recovering state entries from durable storage if needed upon system recovery. The first generation of data-parallel stream processing systems, such as Storm [152] and S4 [123], required user-managed state. In such systems, stateful processing was either implemented with no guarantees, making use of custom in-memory data structures or, often implemented using external key-value stores that cover certain scalability and persistence needs. For the rest of the systems available, state management concerns have been internally handled by the streaming systems themselves through the use of explicit state APIs or non-programmable, yet internally managed, state abstractions.

4.2.1 Discussion

In the early days of data stream management when main memory was scarce, state had a facilitating role, supporting the implementation of system operators, such as CQL's join, filter, and sort as employed in STREAM [19]. We term this type of state, defined by the designers of a given system and used by the internal operators of that system, *system-defined state*. A common term used to describe that type of state was "synopsis". Typically, users of such systems were oblivious of the underlying state and its implicit nature resembled the use of intermediate results in DBMSs. Systems such as STREAM, as well as Aurora Borealis [41], attached special synopses to a stream application's dataflow graph supporting different operators, such as a window max,

Table 2 State management features in streaming systems

System	Programmable state	State Mgmt responsibility	State Mgmt architecture			Storage medium			
			In-memory	Out-of-core	External	Resilient store Local	Resilient store Remote	Ephemeral	None
Aurora/Borealis [52]	✗	System	✓					✓	
STREAM [19]	✗	System	✓				✓		✓
TelegraphCQ [136]	✗	System	✓			✓			
S4 [123]	✓	User			✓				✓
Storm (1.0) [152]	✓	User			✓				✓
Spark(1.0) [23]	✓	System	✓				✓		
Trident [9]	✓	System	✓	✓			✓		
SEEP [65]	✓	System	✓				✓		
Naiad [120]	✓	System	✓				✓		
TimeStream [129]	✓	System	✓				✓		
Millwheel [14]	✓	System			✓		✓		
Flink [36, 37]	✓	System	✓	✓			✓		
Kafka-Streams [5]	✗	System	✓	✓			✓		
Samza [124]	✓	System	✓	✓		✓			
Streamscope [109]	✓	System	✓				✓		✓
S-Store [42, 147]	✗	System			✓	✓			

a join index or input source buffers for offsets. A noteworthy feature in STREAM was the capability to re-use synopses compositionally to define other synopses in an application internally in the system. Overall, synopses have been one of the first forms of state in early stream processing systems, primarily for stream processing over shared-memory. Several of the issues regarding state, including fault tolerance and load balancing, were already considered back then, for example in Borealis. However, the lack of user-defined state limited the expressive power of that generation of systems to a subset of relational operations. Furthermore, the use of over-specialized data structures was somewhat oblivious to the needs of reconfiguration, which requires state to be flexible and easy to partition.

In the post-MapReduce era, there was a primary focus in compute scalability with systems like Storm [3] allowing the composition of distributed pipelines of tasks. For application flexibility and simplicity, many of these systems did not offer state management, leaving both declaration and management of state to programmers. User-declared and managed state was either defined and used within the working memory and scope provided by the hosting framework or defined and persisted externally, using an existing key-value storage or database system (e.g. Redis [7, 106]). In summary, application-managed state offers flexibility and gives expert users implementation freedom. However, no state management support is offered from the system's side. As a result, the user has to reason about persistence, whether the state fits in the main-memory, and all necessary third-party storage system dependencies. These choices require a combination of deep expertise, and additional engineering work to integrate stream and storage technologies.

Currently, most stream-processing systems allow a level of freedom for user-defined state through a form of a stateful processing API. This feature enables streaming applications to define custom state, while also granting the underlying system access to state information in order to employ data management mechanisms for persistence, scalability and fault tolerance. State information includes the data types used, serializers and deserializers as well as read and write operations known at runtime. The main limitation of user-defined, system-managed state is the lack of direct control on data structures that materialize that state (e.g., for custom optimizations).

4.3 State management architecture

The state management architecture refers to the way that a streaming system stores and manages its internal or user-defined state. We identify three distinct stateful processing directions in the architecture of data stream runtime systems:

- *In-memory* architectures store state using in-memory data structures. This approach is able to support state that is within main-memory available in each node executing stream operators.
- *External memory* architectures make use of multiple levels of storage media, such as non-volatile memory or

hard disks to store state and process, i.e., using memory outside the address space of a stream operator. The term "out-of-core" is also frequently used to describe data structures, algorithms, and embedded databases that build on external memory. This approach allows exploiting fast main-memory access within each compute node, while also supporting a growing number of state entries that are split and archived in secondary storage. We observe that the out-of-core data structure of choice used in most systems is a variant of an index, such as FASTER [47] or an LSM-Tree [125], such as RocksDB/LevelDB.[1]

– *Remote memory* architectures decouple compute and state, offloading state handling to an external database or key-value store. This approach enables more modular system designs (state and compute decoupling which is very Cloud-friendly) and effective re-use of several desired properties of database systems (e.g., ACID transactions, consistency guarantees, auto-scaling) in support of more complex guarantees in the context of data streaming. The use of external state was predominant within applications in Apache Storm. The lack of system-managed state necessitated users to store all of their state in an external system. In this architecture, when state access is needed, the streaming operator has to reach out to the external system, increasing its latency. Google's Millwheel, the cloud engine of Beam/Google Dataflow, is a representative example of system-managed external state architecture. Millwheel builds on the capabilities of BigTable [50] and Spanner [53] (e.g., blind atomic writes). Tasks in Millwheel are effectively stateless. They do keep recent local changes in memory but overall they commit every single output and state update to BigTable as a single transaction. This behavior means that Millwheel is using an external store for both persisting every single working state per key but also all necessary logs and checkpoints needed for recovery and non-idempontent updates.

4.3.1 Storage medium

Another aspect of stateful streaming, auxiliary to the underlying fault-tolerance mechanisms, is the management of state used for recovery and reconfiguration. As shown in Sect. 5.1.1 there are different options. Recovery state is preferably making use of a *resilient store* that is either *local* to each stateful operator, or in a *remote* resilient store. In the case of Aurora and Borealis [12, 41], recovery state is maintained in non-resilient ephemeral space (e.g., operator process memory). Systems that cache data for recovery in memory, such as output tuples, do not fall in this category.

[1] https://www.github.com/google/leveldb.

4.3.2 Discussion

Stream processing has been influenced by general trends in scalable computing. State and compute have gradually evolved from a scale-up task-parallel execution model to the more common scale-out data-parallel model with related implications in state representations and operations that can be employed. Persistent data structures have been widely used in database management systems ever since they were conceived. The idea of employing internal and external persistence strategies was uniformly embraced in more recent generations of systems. Section 4.4 covers different architectures and presents examples of how modern systems can support large volumes of state, beyond what can fit in memory, within unbounded executions. Another foundational transitioning step in stream technology was the development and adoption of transactional-level guarantees. Section 5.1.1 gives an overview of the state of the art and covers the semantics of transactions in data streaming, alongside implementation methodologies.

4.4 Scalability and state management

Scalable state has been the main incentive of the second generation of stream processing systems which automated deployment and partitioning of data stream computations. The need for scalable state was driven by the availability of voluminous unbounded data streams. In high-volume streaming computations, the space complexity for stream state is linear to the ever-increasing input consumed by a stream processor. This section discusses types of scalable state, as well as scalable system architectures that can sustain support for partitioning, persisting, and committing changes to large volumes of state.

4.4.1 Parallel versus global stateful operations

To employ data-parallelism in a stateful computation (e.g., an aggregate on a given key) the state of the computation also has to be partitioned across different operator instances. However, partitioning state is not always possible (e.g., when an aggregate has to be performed across all keys of a stream). Scalable state takes two forms in a streaming application, typically referred to as *partitioned* and *non-partitioned* state (also referred to as *global* state). Depending on the nature of a specific operation, one or both of these state types can be employed.

Partitioned state Partitioned state is the most common way to enable data-parallel computation on massive data streams. Partitioned state assigns key-wise logical partitions of state to compute tasks, where each logical task handles a specific key. This approach is enabled in the API level through an

additional operation that is invoked prior to stateful processing that lifts the scope from task- to key-based processing such as "keyBy" in Apache Flink or "groupBy" in Beam and Kafka-Streams. Note that logical partitioning (i.e., which logical operator takes over a given key) differs from physical partitioning (i.e., which nodes take over the computations on a given set of keys). Typically, multiple keys (or key ranges) are assigned to a given compute node.

Non-partitioned state Non-partitioned state is mapped as a singleton to physical compute tasks. Such non-partitioned state is typically used in two ways. First, it can be used in order to compute global aggregates over the complete input stream. Second, it can be used to calculate aggregates at the level of the physical operator (e.g., count how many keys have been processed per operator). Task-level state can also be useful for keeping offsets when consuming logs from a physical stream source task. Because non-partitioned state either deals with operator-local computations or with global aggregates, its use is not scalable and should be used with caution by practitioners.

4.5 First versus second generation

State has been central to stream processing. The notion of state itself has been addressed with many names, such as "summary", "synopsis", "sketch" or "stream table" and it reflects the evolution of data stream management along the years. Early systems [12, 19, 27, 48] (circa 2000–2010) hid state and its management from the user. Most continuous processing operators at that time, such as those of the time-varying relational model of CQL [21] in STREAM [19], were implemented using internal in-memory data structures. Overall, the purpose of state was to support the creation of a limited set of operators offered by each system.

A decade later, scalable data computing systems based on the MapReduce [61] architecture allowed for arbitrary user-defined logic to be scaled and executed reliably using distributed middleware and partitioned file systems. Following the same trend, many existing data management models were revisited and re-architectured with scalability in mind (e.g., NoSQL and NewSQL databases). Similarly, a growing number of scalable data stream processing systems [14, 16, 37, 118] married principles of scalable computing with stream semantics and models that were identified in the past (e.g. out-of-order processing [108, 142]).

As of today, modern stream processors can compile and execute graphs of long-running operators with complete, user-defined (yet system-managed) state that is fault-tolerant and reconfigurable given a clear set of transactional guarantees[14, 36, 64].

4.6 Open problems

Data streaming covers many data management needs today that go beyond real-time analytics, which was the original purpose of the stream processing technology. The transitions of stateful processing showcase this trend.

The decoupling of state programming from state persistence resembles the concept of data independence in databases. Systems are converging in terms of semantics and operations on state while, at the same time, many new methods employed on embedded databases (e.g., LSM-trees, state indexing, externalized state) are helping stream processors to evolve in terms of performance capabilities. Recent work [24, 88, 105] showcases the potential of workload-aware state management, adapting state persistence and access to the individual operators of a dataflow graph. To this end, an increasing number of "pluggable" systems [47, 169] for local state management with varying capabilities are being adopted by stream processors. This trend opens new capabilities for optimization and sophisticated, yet transparent, state management that can automate the process of selecting the right physical plan and reconfigure that plan while continuous applications are executed.

5 Fault tolerance

Fault tolerance is a system's ability to continue its operation as if no failures have occurred. Without fault tolerance, streaming systems, which process potentially unbounded data, would have to repeat data processing from the beginning if the state of a computation was lost during a failure. Worse, it is uncertain whether an input data stream could be recovered following a failure, such that it can be processed again. Finally, the scale of distributed streaming system deployments highlights further the importance of fault tolerance. The larger the scale, the more a failure can affect the system's operation.

We start this section with an account of processing semantics, which characterize the levels of correctness that a streaming system can achieve considering failure scenarios (Sect. 5.1). In Sect. 5.2 we describe the important role of state snapshots in fault tolerance. We devote Sect. 5.3 to the output-commit problem in stream processing, which regards the correctness of a system's output as observed by the outside world. Section 5.4 presents the literature on the availability of streaming systems, which is closely related to fault tolerance. Finally, the section ends with a comparison between first and second generation systems with respect to fault tolerance (Sect. 5.5) and open problems (Sect. 5.6).

Table 3 Fault-tolerance in streaming systems

System	Processing semantics			Replication			Recovery data			Transaction granularity
		Least State	Exactly-once Output		Active	Passive	None	State	Output	None
Aurora* [52]	✓				✓			✓		No
TelegraphCQ [136]		✓		✓			✓	✓		No
Borealis [11, 30]	✓			✓			✓	✓		No
S4 [123]	✓					✓			✓	No
Seep [64, 65]			✓		✓		✓	✓		Epoch-level
Naiad [120]		✓			✓		✓	✓		Epoch-level
Timestream [129]			✓		✓		✓	✓		Epoch-level
Millwheel [14]			✓		✓		✓	✓		Record-level
Storm [152]	✓					✓			✓	No
Trident [9]			✓		✓		✓			Batch-level
S-Store [42, 147]		✓			✓		✓			Batch-level
Trill [44]		✓			✓		✓			No
Heron [100]	✓					✓			✓	No
Streamscope [109]			✓	✓	✓	✓	✓		✓	Epoch-level
Streams [56]		✓			✓		✓			Epoch-level
Samza [124]	✓				✓		✓			Epoch-level
Flink [36, 37]		✓			✓		✓			Epoch-level
Spark [23]		✓			✓		✓			Batch-level

5.1 Processing semantics

Processing semantics conveys how a system's state is affected by failures. Typically, all systems in the literature are able to produce correct results in failure-free executions. But to mask a failure completely is hard, especially in the stream processing domain where the output should be delivered as soon as it is produced.

In recent years, the stream processing domain has settled on the terms *at least-once* and *exactly-once* to characterize the processing semantics [23, 37, 56, 109, 124]. At most-once is also part of the nomenclature but it is mostly obsolete, as systems opt to support one of the two stronger levels. At least-once processing semantics means that the system will produce the same results as a failure-free execution with the addition of duplicate records as a side-effect of recovery. We detail the consistency guarantees of different systems in Table 3.

Exactly-once processing lends itself to two different interpretations. A system may support exactly-once processing semantics within its boundaries ensuring that any inconsistencies or duplicate execution carried out on recovery is not part of its state. We call this level of semantics exactly-once processing semantics on *state*.

It should be noted that most systems in this category still assume that the computations they apply, as well as the system's functions, are deterministic, which is often not the case; processing-time windows and operators processing input from multiple sources are two prime examples of nondeterminism. With nondeterminism at play, the system's state on recovery can diverge. Clonos [138] provides exactly-once processing including nondeterministic computations by means of causal consistency. It keeps determinants about nondeterministic computations in a resilient manner and uses them to regenerate the exact computational state following a failure.

A system with exactly-once processing semantics on state can still produce duplicate output. One notable example is while recovering from a failure. This outcome is possible, because, as opposed to the system's state, the output cannot be rolled back. The output can be consumed immediately by external applications. This problem has been termed the *output-commit problem* [62] in the distributed-systems literature. Systems that produce the same output under failure as a failure-free execution support exactly-once processing semantics on *output*. In Sect. 5.3 we elaborate how streaming systems address the output-commit problem.

5.1.1 State consistency

Consistent stream processing has been an open research problem for quite some time due to the challenging nature of distributed processing of unbounded streams, but also due to

the lack of a formal definition of the problem itself. Consistency relates to the guarantees that a system can provide in face of failures.

The Lambda architecture With the advent of cloud computing, a design pattern called the "lambda architecture" became mainstream. The lambda architecture proposed the separation of systems across different layers according to their specialization and reliability capabilities. Hadoop was reliable in terms of processing guarantees (i.e., exactly-once processing by atomically processing batches of data), thus, it could execute correct computation. However, Hadoop-based solutions suffered from high latency. On the other hand, early stream-processing systems could achieve low latency but they did not offer consistency guarantees— they mostly guaranteed at-least-once semantics.

At the same time, databases had formal guarantees. For example, a set of transactions would be processed using ACID guarantees. In the context of data streaming, though, it took some time in order to decide on the need for exactly-once processing, and the need for input logging for a possible replay. We discuss processing guarantees in the following.

Consistent state in stream processing A stream processor today is a distributed system consisting of different concurrently executing tasks. Source tasks subscribe to input streams that are typically recorded in a partitioned log, such as Kafka, and therefore, input streams can be replayed. Sink tasks commit output streams to the outside world. Every task in this system can contain its own state. For example, source tasks need to keep the current position of their input streams in their state. A system execution can be often modeled through the concept of "concurrent actions" [35]. An action includes: invoking stream task logic on an input event, mutating its state, and producing output events. Every action happening in such a system causes other actions. Effectively, just a single record sent by a source contributes to state updates throughout the whole pipeline and to output events created by the sinks. If a specific action is lost or happens more than once, then the complete system enters into an *inconsistent* state.

Fault tolerance is an integral aspect of streaming systems that significantly affects their state consistency. We analyze the fault tolerance strategies of existing streaming systems in Sect. 5.1.2. In addition, due to causal dependencies on state, the order of action execution is also critical. Note that causality here concerns only the execution order of task actions after events enter the system, not the order in which events are ingested. Therefore, a system can tolerate out-of-order streams with respect to event timestamps. However, this does not mean that it is capable of maintaining causal dependencies within its dataflow execution among the event streams that have been processed (in either ingestion order). Existing reliable stream processors either define a transaction out of each action or a coarse-grained set of actions that we call *epochs*. We explain these approaches in more detail, next.

5.1.2 Properties of consistency mechanisms

Managing failures in a distributed streaming system entails maintaining snapshots of state, migrating state, and scaling out operators while affecting the healthy parts of the system as little as possible. Table 3 presents the fault-tolerance strategies of known streaming systems arranged in order of publication appearance, from past to present. We analyze the strategies across the following three dimensions.

1. Replication considers the use of additional computational resources for recovering an execution. We adopt the terminology of Hwang et al. [85] that classifies replication as either *active*, where two instances of the same execution run in parallel, or *passive*, where each running stateful operator dispatches its checkpointed state to a standby operator.

2. Recovery data addresses what data are regularly stored for recovery purposes. Data may include the *state* of each operator and the *output* it produces. In addition, many fault tolerance strategies need to replay tuples of input streams during recovery in order to reprocess them. For this purpose, input streams are persistently stored, typically in message brokers like Apache Kafka. Stream processing consumer tasks only need to store read input positions from input logs within their state for recovery.

3. Transaction Granularity considers the categorization of systems by the frequency at which they obtain snapshots of their state into Record-, Epoch- or Batch-level snapshot frequency.

The table is meant to be read both horizontally, to describe a specific system's approach to fault tolerance, and vertically, to uncover how the different building blocks shape the landscape of fault tolerance in stream processing. Two remarks are necessary. Streamscope [109] presents and evaluates three distinct fault-tolerance strategies; an active replication-based strategy, a passive one, and a strategy that relies on recomputing state by replaying data from input streams. Furthermore, the state column in the recovery data dimension captures not only checkpointed state but also state metadata that allow recomputing the state, such as a changelog [124] or state dependencies [129].

The table reveals four interesting patterns. First, of all columns, two accumulate the majority of checkmarks: passive replication and storing state for recovery. This pattern is perhaps the most visible on the table and signifies that passive replication by storing state is, unsurprisingly, a very popular option for streaming systems. One typical recovery approach is to restore the latest checkpoint of a failed operator in a new node and replay input that appeared after the checkpoint. Variations of this approach include saving in-

flight tuples along with the state and maintaining in-flight tuples in upstream nodes. Second, storing in-flight tuples for recovery is not preferred anymore, although it was a popular option for streaming systems in the past. One major development that explains this shift is the advent of message brokers, such as Apache Kafka [98], which can produce an ordered stream of data starting from a specific point in time using an offset, which is, for instance, provided by a streaming system on recovery. Third, while past systems strived to support exactly-once output processing semantics, later systems opt for exactly-once semantics on state and outsource the deduplication of output to external systems. We will elaborate on this aspect in Sect. 5.3.

5.2 Fault tolerance and state snapshots

Snapshots are persisted copies of the stream-processing states for the purposes of fault recovery and reconfiguration. The consistency properties and employed mechanisms in each system impose different requirements on how often such copies need to be obtained. We differentiate systems by the frequency at which they record snapshots of their state into Record- and Epoch-level snapshot frequency.

Record-granularity An opposite extreme to the epoch-based approach is to snapshot after each record. This approach, as seen in Millwheel [14], requires the stream processor to obtain a copy of all changes that occurred in the state after the complete processing of a each stream record. This copy includes the newly updated states and produced output records.

Epoch-granularity is typically achieved in the form of application-level snapshots. Most commonly, systems employ a form of asynchronous consistent snapshotting, such as the Chandy–Lamport algorithm [49]. In each epoch, i.e., either periodically or after a certain number of records have been ingested by the system, each operator records a copy of its state. The batch-level snapshotting seen in systems such as Spark Streaming and Trident/Storm adopts a strict micro-batching processing paradigm: i.e., a batch execution is submitted after collecting a sizable number of records, and the state of an operator is stored right after a given batch has been processed. S-Store orchestrates the batch granularity as a series of ACID transactions on top of a relational database.

5.2.1 State durability at record granularity

A form of consistent processing in data streaming is employing a transaction per local action. Google's Millwheel, the cloud runtime for the dataflow data streaming service, employs such a strategy. Millwheel uses BigTable to commit each full compute action which includes: input events, state transitions and generated output. The act of committing these actions is also called a "strong production" in Millwheel.

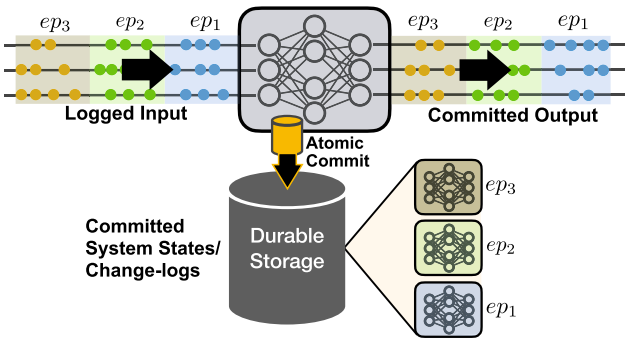


Fig. 4 Transactional epoch commits in data streaming

Persisting state of an operator per output event, is an approach which seemingly induces high latency overhead. However, traditional database optimizations can be used to speed up commit and state read times. Write ahead logging, blind writes, bloom filters, and batch commits at the storage layer can be used to reduce the commit latency. More importantly, since the order of actions is predefined at commit time, state persistence on a per-event basis also guarantees deterministic executions. In addition, this approach has important effects on consistency as perceived by applications that consume the system's output. This benefit follows from the fact that "exactly-once processing" in this context relates to each action being atomically committed, as we discuss in Sect. 5.3. The core observation behind the correctness of this approach is based on the fact that each action is invoked by an atomically committed action. Following the natural causal order of actions, the transitive closure of all actions in the system is, therefore, also atomically committed.

5.2.2 State durability at epoch granularity

The frequency of recordings per state in epoch-level processing is more coarse-grained compared to the per-record granularity.

In Fig. 4 we depict the overall approach, marking input, system states, and outputs with a distinct epoch identifier. Epochs can be defined through markers at the logged input of the streaming application. A system execution can be instrumented to process each epoch and commit the state of the entire task graph after each epoch is processed. If a failure or other reconfiguration action happens during the execution of an epoch, then the system can roll back to a previously committed epoch and recover its execution. The term "exactly-once processing" in this context relates to each epoch being atomically committed. In Sect. 5.1.2, where we presented the different levels of processing semantics in streaming, we called this flavor *exactly-once processing on*

state. The rest of this section focuses on various approaches used to commit stream epochs.

Strict two-phase epoch commits A common coordinated protocol to commit epochs is a strict two-phase commit where Phase 1 corresponds to the full processing of an epoch and Phase 2 ensures persisting the state of the system at the end of the computation.

This approach was popularized by Apache Spark [164] through the use of periodic "micro-batching" and it is an effective strategy when using batch processing systems for unbounded processing. The main downside of this approach is the risk of low task utilization due to synchronous execution, since each task has to wait for all other tasks to finish their current epoch. Drizzle [158] mitigates this problem by chaining multiple epochs in a single atomic commit. S-Store employs a similar approach [116], where each database transaction corresponds to an epoch of the input stream that is already stored in the same database.

Asynchronous two-phase epoch commits For pure dataflow systems, strict two-phase committing is problematic, since tasks are uncoordinated and long-running. Furthermore, it is feasible to achieve the same functionality asynchronously through consistent snapshotting algorithms, known from classic distributed systems literature [35]. Consistent snapshotting algorithms exhibit beneficial properties because they do not require pausing a streaming application. Furthermore, they acquire a snapshot of a consistent cut in a distributed execution [49]. In other words, they capture the global states of the system during a "valid" execution. Throughout different implementations we can identify i) unaligned and ii) aligned snapshotting protocols.

I. Unaligned (Chandy–Lamport [49]**)** snapshots provide one of the most efficient methods to obtain a consistent snapshot. Several stream processors currently support this approach, such as IBM Streams and more recently seen as an experimental configuration option in Flink [10]. The core idea is to insert a punctuation or "marker" into the regular stream of events and use that marker to separate all actions that come before and after the snapshot, while the system is running. A caveat of unaligned snapshots is the need to record input (a.k.a. in-flight) events that arrive to individual tasks until the protocol is complete. In addition to space overhead for logged inputs, unaligned snapshots require more processing during recovery, since logged inputs need to be replayed (similarly to redo logs in database recovery with fuzzy checkpoints).

II. Aligned snapshots Aligned snapshots aim to improve performance during recovery and minimize reconfiguration complexity exhibited by unaligned snapshots. The main difference is prioritizing input stream events that are expected before the snapshot and, thus, end up solely with states that reflect a complete computation of an epoch and no in-flight events as part of a snapshot. For example, Flink's epoch snapshotting mechanism [36, 38] resembles the Chandy Lamport algorithm in terms of using markers to identify epoch frontiers. However, it additionally employs an alignment phase that synchronizes markers within tasks before disseminating them further. Alignment is achieved through partially blocking input channels where markers were previously received until all input channels have transferred all messages corresponding to a particular epoch.

In summary, unaligned snapshots are meant to offer the best runtime performance but sacrifice recovery times due to the redo-phase needed upon recovery. Whereas, aligned snapshots can lead to slower commit times due to the alignment phase, while providing a set of beneficial properties. First, aligned snapshots reflect a complete execution of an epoch, which is useful in use cases where snapshot-isolated queries need to be supported on top of data streaming [159]. Furthermore, aligned snapshots yield the lowest reconfiguration footprint, as well as set the basis for live reconfiguration within the alignment phase as exhibited by Chi [112].

5.3 The output-commit problem

The output-commit problem [62] specifies that a system should only publish its output to the outside world, under the certainty that the system can recover its state from where the output was published. This is to ensure that every output is only published once, since output cannot be retracted once it is made available to the outside world. If output is published twice, then the system manifests inconsistent behavior with respect to the outside world. The systems comprising the outside world fall out of the problem's scope and, thus, it is assumed that they cannot fail. An important instance of this problem manifests when a system is restoring some previous consistent state due to a failure.

The output-commit problem is relevant in streaming systems, which typically conform to a distributed architecture and process unbounded data streams. In this setting, the side effects of failures are difficult to mask. For example, assume that a streaming system with exactly-once processing semantics on state takes a snapshot and, shortly afterward, one of its operators crashes. After the operator took a snapshot of its state, it continued to produce output until it crashed. When the operator recovers using the most recent snapshot, it will process the input data that succeeded the snapshot operation again. Consequently, it will re-produce the output that it had produced under its normal operation after taking the last snapshot and before suffering the crash.

Streaming systems that solve the output-commit problem have received multiple descriptions in the literature, including that they provide (1) exactly-once processing semantics

on output, (2) output exactly-once, (3) precise recovery [85], and (4) strong productions [14].

Although the problem is relevant and hard, solutions in the stream-processing domain are scattered in the literature pertaining to each system in isolation. We group the various solutions into three categories: transaction-based, progress-based, and lineage-based. We now describe each of those, focusing on the assumptions they involve. Each of the three types of techniques uses a different characteristic of the input or computation, to identify whether a certain tuple has appeared previously. Transaction-based techniques use tuple identity, while progress-based techniques use order. Finally, lineage-based techniques use input–output dependencies. Finally, we provide two more categories of solutions, special sink operators and external sinks that do solve the problem in practice, but, strictly speaking, they do not meet the problem's specification because they are either specific or external to a streaming system.

Transaction-based Millwheel [14] and Trident [9] rely on committing unique ids with records to eliminate duplicate retries. Millwheel assigns a unique id to each record entering the system and commits every record it produces to a highly-available storage system before sending it downstream. Downstream operators acknowledge received records. Millwheel assumes no input ordering or determinism. Trident, on the other hand, batches records into a transaction, which is assigned a unique transaction id and applies a state update to the state backend. Assuming that transactions are ordered, Trident can accurately ignore retried batches by checking the transaction id.

Progress-based Seep [64, 65] uses timestamp comparison to deliver exactly-once output, relying on the order of timestamps. Each operator generates increasing scalar timestamps and attaches them to records. Seep checkpoints the state and output of each operator together with the vector timestamps of the latest records from each upstream operator that affected the operator's state. On recovery, the latest checkpoint is loaded to a new operator, which replays the checkpointed output records and processes replayed records sent by its upstream operators. Downstream operators discard duplicate records based on the timestamps. The system assumes deterministic computations that do not rely on system time or random input.

Lineage-based Timestream [129] and Streamscope [109] use dependency tracking to provide exactly-once output. During normal operation, both systems track operator input and output dependencies by uniquely identifying records with sequence numbers. Streamscope persists records with their identifiers asynchronously. Both systems store operator dependencies periodically in an asynchronous manner. In Streamscope, however, each operator checkpoints individually not only its dependencies but also its state.

On recovery, Timestream retrieves the dependencies of failed operators by contacting upstream nodes recursively, until all inputs required to rebuild the state, are available. Streamscope follows a similar process, but starts from a failed operator's checkpoint snapshot. For each input sequence number in the snapshot that is not found in persistent storage, Streamscope contacts upstream operators, which may have to recompute the record starting from their most relevant snapshot that can produce the output record given its sequence number. Finally, both systems use garbage collection to discard obsolete dependencies but in subtly different manners.

Timestream computes the input records required by upstream operators in reverse topological order from the final output to the original input and discards unneeded ones. Streamscope does the same, but instead of computing dependencies, it uses low watermarks per operator and per stream to discard older, unneeded snapshots and records. In Timestream, storing dependencies asynchronously can lead to duplicate recomputation, but downstream operators bearing the correct set of dependencies can discard them. Streamscope applies the same process only if duplicate records cannot be found in persistent storage. Both Timestream and Streamscope assume deterministic computation and input in terms of order and values.

The progress-based and lineage-based solutions are vulnerable to failures of the last operator(s) on the dataflow graph, which produces the final output, since both solutions rely on downstream operators for filtering duplicate records. In contrast, transaction-based approaches do not require a downstream operator for deduplication, since they can use the unique id of a record to check whether it is a duplicate (Table 4).

Special sink operators Streams [56] implements special sinks for retracting output from files and databases. The application of this approach solves the output-commit problem for specific use cases, but it is not applicable in general, since it defies the core assumption of the problem that output cannot be retracted.

External sinks Some systems like Streams [56], Flink [37], and Spark [23] provide exactly-once semantics on state and outsource the output-commit problem to external sinks that support idempotent writes, such as Apache Kafka.

One way to categorize the solutions provided by special sink operators and external sinks, is *optimistic* and *pessimistic output techniques*. Optimistic output techniques publish their output immediately and retract, or update it if needed. Pessimistic output techniques use a form of write-ahead log, to write the output they will publish, if everything

Table 4 Assumptions that systems make for solving the output-commit problem

System	Assumptions
Millwheel	External high-throughput transactional store
Timestream	Deterministic computation and input
Streamscope	Deterministic computation and input
Trident	Deterministic computation and input, ordering of transactions
Seep	Deterministic computation, monotonically- increasing logical clock, records ordered by timestamp

goes well, until the output is permanently committed [36]. Optimistic output techniques, which resemble multi-version concurrency control from the database world, include modifiable and versioned output destinations, while pessimistic output techniques include transactional sinks and similar tools.

5.4 High availability

Existing studies of high availability in stream processing proposed an active replication approach [30, 136], a passive replication approach [75, 86, 102], or a hybrid active-passive replication approach [81, 143, 168]. Finally, two benchmark evaluations assessed the approaches above, under simulated experiments [43, 85].

Active replication Flux [136] implements active replication by duplicating the computation and coordinating the progress of the two replicas. Flux restores operator state and in-flight data of a failed partition while the other partition continues to process input. A new primary dataflow that runs following a failure quiesces when a new secondary dataflow is ready in a standby machine, in order to copy the state of its operators to the new secondary. In contrast, Borealis [30] has nodes address upstream node failures by switching to a live replica of the failed upstream node. If a replica is not available, the node can produce tentative output for incomplete input to avoid the recovery delay. The approach sacrifices consistency to optimize availability, but guarantees eventual consistency.

Passive replication Hwang et al. [86] propose that a server in a cluster has one or more servers as backup where it ships independent parts of its checkpointed state. When a node fails, its backup servers that hold parts of its checkpointed state initiate recovery in parallel by starting to execute the operators of the failed node whose state they have and collecting the input tuples they have missed from the checkpointed state they possess.

SGuard [102] and Clonos [138] save computational resources in an alternative fashion, by checkpointing state asynchronously to a distributed file system. Upon failure, a node is selected to run a failed operator. The operator's state is loaded from the file system and its in-memory state is reconstructed before it can join the job. Beyond asynchronous checkpointing, a new checkpoint mechanism [75] preserves output tuples until an acknowledgment is received from all downstream operators. Next, an operator trims its output tuples and takes a checkpoint. The authors show that passive replication still requires longer recovery time than active replication, but with 90% less overhead due to reduced checkpoint size.

Hybrid replication Zwang et al. [168] propose a hybrid approach to replication, which operates in passive mode under normal operation, but switches to active mode using a suspended pre-deployed secondary copy when a transient failure occurs. According to their experimental results, their approach saves 66% recovery time compared to passive replication and produces 80% less message overhead than active replication.

Alternatively, Heinze et al. [81] propose to dynamically choose the replication scheme for each operator, either active replication or upstream backup, in order to reduce the recovery overhead of the system by limiting the peak latency under failure below a threshold. The upstream backup approach maintains that each upstream operator preserves its output data until its downstream operators process them. If an operator fails, the upstream operators will replay the output data they preserve, such that the substitute operator of the failed operator can reconstruct the state of its predecessor. Similarly, Su et al. [143] counter correlated failures by passively replicating processing tasks, except for a dynamically selected set that is actively replicated.

Benchmarking of high availability approaches In their seminal work, Hwang et al. [85] model and evaluate the recovery time and runtime overhead of four recovery approaches, active standby, passive standby, upstream backup, and amnesia (i.e., dropping data for faster recovery), across different types of query operators. The simulated experiments suggest that active standby achieves near-zero recovery time at the expense of high overhead in terms of resource utilization, while passive standby produces worse results in terms of both metrics compared to active standby. However, passive

standby poses the only option for arbitrary query networks. Upstream backup has the lowest runtime overhead at the expense of longer recovery time. With a similar goal, Shrink [43], a distributed systems emulator, evaluates the models of five different resiliency strategies with respect to uptime SLA and resource reservation. The strategies differ across three axes, single-node vs multi-node, active vs passive replication, and checkpoint vs replay. According to the experiments with real queries on real advertising data using Trill [44], active replication with periodic checkpoints is proved advantageous in many streaming workloads, although no single strategy is appropriate for all workloads.

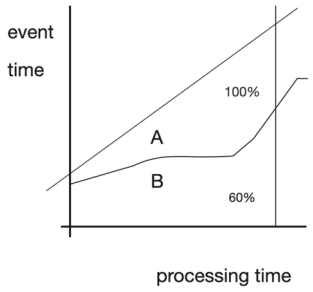


Fig. 5 Measuring availability with the slack between processing time and event time over time

5.5 First generation versus second generation

In the early years, streaming systems put emphasis on approximate (at-most- or at-least-once) results, while modern systems maintain exactly-once processing semantics over their state under failures. Although past systems lacked in terms of consistency, mainly due to state management aspects, they strived to solve the output-commit problem. Instead, a typical avenue for modern systems that is gaining traction is to outsource the deduplication of output to external systems. Finally, while streaming systems used to store their output to enable replaying tuples to downstream operators recovering from a failure, now, systems increasingly rely on replayable input sources for replaying input subsets.

At the same time, it was very common to implement high availability using active replication. In contrast, modern systems tend to leverage passive replication, especially by allocating extra resources on demand, which is appropriate in Cloud settings.

5.6 Open problems

We highlight three open problems in the scope of fault tolerance and high availability in streaming systems. These regard novel solutions to the output-commit problem, defining and measuring availability in stream processing, and configuring availability for different application requirements.

First, the importance of the output-commit problem has the prospect to increase as streaming systems are used in novel ways, such as running event-driven applications. Although we presented five different types of solutions, these suffer from computational cost, strong assumptions, limited applicability, and freshness of output results. New classes of solutions are required that score better in these dimensions.

Second, the literature of high availability in stream processing has significantly enhanced the availability of streaming systems throughout the years. But, to the best of our knowledge, there has been scant research on the semantics of availability for stream processing in particular. The generic definition of availability for computer systems by Gray et al. [73] relates availability merely to failures. According to the definition, a system is available when it responds to requests with correct results, which is termed as service accomplishment. In stream processing however, processing is continuous and potentially unbounded. Responding with correct results becomes more challenging.

The factors that may impair availability in streaming include software and hardware failures, overload, backpressure, and types of processing stall, such as checkpoints, state migration, garbage collection, and calls to external systems. The common denominator among those factors, is that the system falls behind input. This lag may not be a problem for other types of systems, such as databases where non updated responses can be perceived as adequate by applications, as long as the serializability property is satisfied. Streaming systems, though, are typically expected to continuously keep processing up with the input in order to provide fresh results.

This survey contributes a more refined definition of availability for stream processing as follows. *A streaming system is available when it can provide output corresponding to the processing of its most recent input.* This definition affects how availability is measured. An appropriate way would be via progress tracking mechanisms, such as *the slack between processing time and event time over time*, which quantifies the system's processing progress with respect to the input as per Fig. 5. The plot depicts the slack between event time and processing time over time. The surface enclosing A amounts to 100% availability, while the surface containing B equals 60% availability.

Finally, availability is a prime non-functional characteristic of a streaming system and non-trivial to reason about, as we have shown. Providing user-friendly ways to specify availability, as a contract that the system will satisfy during its operation, can significantly improve the position of streaming systems in production environments. Configuring availability in this way has the potential to impact resource utilization, performance overhead during normal operation, recovery time, and consistency.

6 Load management, elasticity, and reconfiguration

Due to the push-based nature of streaming inputs from external data sources, stream processors have no control over the rate of incoming events. Satisfying Quality of Service (QoS) under workload variations has been a long-standing research challenge in stream processing systems.

To avoid performance degradation when input rates exceed system capacity, the stream processor needs to take actions that will ensure sustaining the load. One such action is *load shedding*: temporarily dropping excess tuples from inputs or intermediate operators in the streaming execution graph. Load shedding trades off result accuracy for sustainable performance and is suitable for applications with strict latency constraints that can tolerate approximate results.

When result correctness is more critical than low latency, dropping tuples is not an option. If the load increase is transient, the system can instead choose to reliably buffer excess data and process it later, once input rates stabilize. Several systems employ *back-pressure*, a fundamental load management technique applicable to communication networks that involve producers and consumers. Nevertheless, to avoid running out of available memory during load spikes, *load-aware scheduling* and rate control can be applied.

A more recent approach that aims at satisfying QoS while guaranteeing result correctness under variable input load is *elasticity*. Elastic stream processors are capable of adjusting their configuration and scaling their resource allocation in response to load. Dynamic scaling methods are applicable to both centralized and distributed settings. Elasticity not only addresses the case of increased load, but can additionally ensure no resources are left unused when the input load decreases.

Next, we review load shedding (Sect. 6.1), load-aware scheduling and flow control (Sect. 6.2), and elasticity techniques (Sect. 6.3). As in previous sections, we conclude with a discussion of first generation vs. modern systems and open problems.

6.1 Load shedding

Load shedding [28, 144, 145, 154] is the process of discarding data when input rates increase beyond system capacity. The system continuously monitors query performance and if an overload situation is detected, it selectively drops tuples according to a QoS specification. Load shedding is commonly implemented by a standalone component integrated with the stream processor. The load shedder continuously monitors input rates or other system metrics and can access information about the running query plan. Its main functionality consists of detecting overload (*when* to shed load) and deciding what actions to take in order to maintain acceptable latency and minimize result quality degradation. These actions presume answering the questions of *where* (in the query plan), *how many*, and *which* tuples to drop.

Detecting overload is a crucial task, as an incorrectly triggered shedding action can cause unnecessary result degradation. To facilitate the decision of *when*, load shedding components rely on statistics gathered during execution. The more knowledge a load shedder has about the query plan and its execution, the more accurate decisions it can make. For this reason, many stream processors restrict load shedding to a predefined set of operators, such as those that do not modify tuples, i.e. filter, union, and join [57, 90, 144]. Other operator-restricted load shedding techniques target window operators [28, 146], or even more specifically, query plans with `SUM` or `COUNT` sliding window aggregates [28]. An alternative, operator-independent approach is to frame load shedding as a feedback control problem [154]. The load shedder relies on a dynamic model that describes the relationship between average tuple delay (latency) and input rate.

Once the load shedder has detected overload, it needs to perform the actual load shedding. This includes the decision of where in the query plan to drop tuples from, as well as which tuples and how many. The question of where is equivalent to placing special *drop operators* in the best positions in the query plan. In general, drop operators can be placed at any location in the query plan, however, they are often placed at or near the sources. Dropping tuples early avoids wasting work later, but it might affect results of multiple queries if the stream processor operates on a shared query network. Alternatively, a load shedding road map (LSRM) can be used [144]. This map is a pre-computed table that contains materialized load-shedding plans, ordered by the amount of load shedding they will cause.

The question of which tuples to drop is relevant when load shedding takes into account the *semantic* importance of tuples with respect to results quality. A *random* dropping strategy has been applied to sliding window aggregate queries to provide approximate results by inserting random sampling operators in the query plan [28]. *Window-aware* load shedding [146] applies shedding to entire windows instead of individual tuples, while *concept-driven* load shedding [92] is a semantic dropping strategy that selects tuples to discard based on the notion of concept drift.

6.2 Scheduling and flow control

When load bursts are transient and a temporary increase in latency is preferred to missing results, back-pressure and flow control can provide load management without sacrificing accuracy. Flow control methods include buffering excess load, load-aware scheduling that prioritizes operators with the objective to minimize the backlog, regulating the transmission rate, and throttling the producer. Flow control and

back-pressure techniques do not consider application-level quality requirements, such as the semantic importance of input tuples. Their main requirement is availability of buffer space at the sources or intermediate operators and that any accumulated load is within the system capacity limits, so that it will be eventually possible to process the data backlog.

Load-aware scheduling tackles the overload problem by selecting the *order* of operator execution and by adapting the *resource allocation*. For instance, backlog can be reduced by dynamically selecting the order of executing filters and joins [25, 29]. Alternatively, adaptive scheduling [26, 40] modifies the allocation of resources given a static query plan. The objective of load-aware scheduling strategies is to select an operator execution order that minimizes the total size of input queues in the system. The scheduler relies on knowledge about operator selectivities and processing costs. These statistics are either assumed to be known in advance, or need to be collected periodically during runtime. Operators are assigned priorities that reflect their potential to minimize intermediate results, and, consequently, the size of queues. Online shuffle grouping [132] is an adaptive per-tuple scheduling technique that aims to reduce the imbalance caused by non-uniform tuple execution times. It relies on sketches to continuously monitor tuple execution times and uses a greedy scheduling algorithm to assign tuples to parallel tasks in an online fashion.

Back-pressure and flow control In a network of consumers and producers such as a streaming execution graph with multiple operators, back-pressure has the effect that all operators slow down to match the processing speed of the slowest consumer. If the bottleneck operator is far down the dataflow graph, back-pressure propagates to upstream operators, eventually reaching the data stream sources. To ensure no data loss, a persistent input message queue, such as Apache Kafka, and adequate storage space are required.

Buffer-based back-pressure implicitly controls the flow of data via buffer availability. Considering a fixed amount of buffer space, a bottleneck operator will cause buffers to gradually fill up along its dataflow path. Figure 6a demonstrates buffer-based flow control when the producer and the consumer run on the same machine and share a buffer pool. When a producer generates a result, it serializes it into an output buffer. If the producer and consumer run on the same machine and the consumer is slow, the producer might attempt to retrieve an output buffer when none is available. The producer's processing rate will, thus, slow down according to the rate the consumer is recycling buffers back into the shared buffer pool. The case when the producer and consumer are deployed on different machines and communicate via TCP is shown in Fig. 6b. If no buffer is available on the consumer side, the TCP window mechanism will inform the sender to halt data transmission. The producer can use a threshold to control how much data is in-flight and it is slowed down if it cannot put new data on the wire.

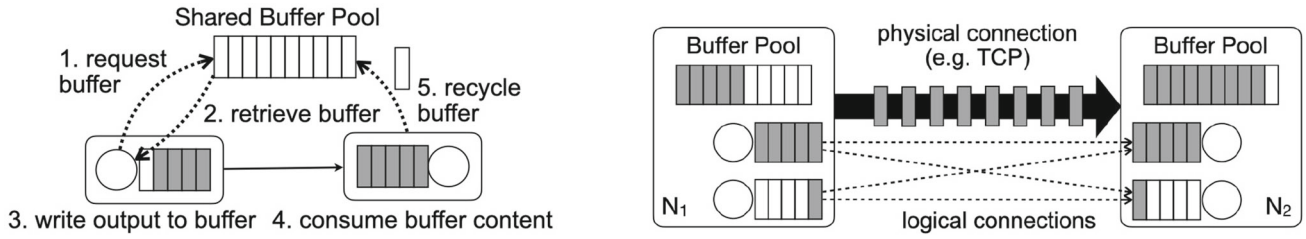
Credit-based flow control (CFC) [101] is a link-by-link, per virtual channel congestion control technique used in ATM network switches. In a nutshell, CFC uses a credit system to signal the availability of buffer space from receivers to senders. This classic networking technique turns out to be very useful for load management in modern, highly-parallel stream processors and is implemented in Apache Flink [1]. Figure 7 shows how the scheme works for a hypothetical dataflow. Parallel tasks are connected via virtual channels multiplexed over TCP connections. Each task informs its senders of its buffer availability via credit messages. This way, senders always know whether receivers have the required capacity to handle data messages. When the credit of a receiver drops to zero (or a specified threshold), back-pressure appears on its virtual channel. An important advantage of this per-channel flow control mechanism is that back-pressure is inflicted on pairs of communicating tasks only and does not interfere with other tasks sharing the same TCP connection. This aspect is crucial in the presence of data skew, where a single overloaded task could otherwise block the flow of data to all other downstream operator instances. On the downside, the additional credit announcement messages might increase end-to-end latency.

6.3 Elasticity

The approaches of load shedding and back-pressure are designed to handle workload variations in a *statically provisioned* stream processor or application. Stream processors deployed on cloud environments or clusters can make use of a dynamic pool of resources. *Dynamic scaling* or *elasticity* is the ability of a stream processor to vary the resources available to a running computation in order to handle workload variations efficiently. Building an elastic streaming system requires a *policy* and a *mechanism*. The policy component implements a control algorithm that collects performance metrics and decides when and how much to scale. The mechanism effects the configuration change. It handles resource allocation, work re-assignment, and state migration, while guaranteeing result correctness. Table 5 summarizes the dynamic scaling capabilities and characteristics of elastic streaming systems.

6.3.1 Elasticity policies

A *scaling policy* involves two separate decisions. First, it needs to detect the symptoms of an unhealthy or inefficient computation and decide whether scaling is necessary. Symptom detection is a well-understood problem and can be addressed using conventional monitoring tools. Second,



(a) Local exchange (b) Remote exchange

Fig. 6 Buffer-based flow control. Circles denote operator tasks and edges denote data dependencies. In the case of local exchange, tasks are processes on the same physical node. In the case of remote exchange, the producer and consumer tasks are processes on separate physical nodes (shown as N_1 and N_2).

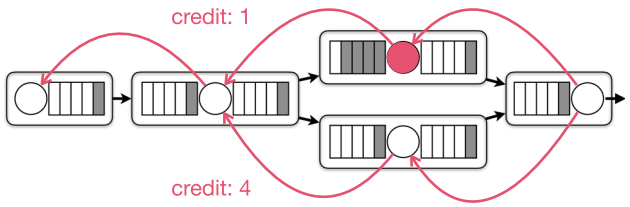


Fig. 7 Credit-based flow control in a dataflow graph. Receivers regularly announce their credit upstream (gray and white squares indicate full and free buffers, respectively) (color figure online)

the policy needs to identify the causes of exhibited symptoms (e.g. a bottleneck operator) and propose a scaling action. This is a challenging task which requires performance analysis and prediction. It is common practice to place the burden of scaling decisions on application users who have to face conflicting incentives. They can either plan for the highest expected workload, possibly incurring high cost, or they can choose to be conservative and risk degraded performance. Automatic scaling refers to scaling decisions transparently handled by the streaming system in response to load. Commercial streaming systems that offer automatic scaling include Google Cloud Dataflow [95], Heron [100], and IBM System S [71], while DS2 [89], Seep [64] and StreamCloud [76] are recent research prototypes with such support.

In Table 5, we categorize policies into *heuristic* and *predictive*. Heuristic policies rely on empirically predefined rules and are often triggered by thresholds or observed conditions while predictive policies make scaling decisions guided by analytical performance models.

Heuristic policy controllers gather coarse-grained metrics, such as CPU utilization, observed throughput, queue sizes, and memory utilization, to detect suboptimal scaling. CPU and memory utilization can be inadequate metrics for streaming applications deployed in cloud environments due to multi-tenancy and performance interference [131]. StreamCloud [76] and Seep [64] try to mitigate the problem by separating user time and system time, but preemption can make these metrics misleading. For example, high CPU usage caused by a task running on the same physical machine as a dataflow operator can trigger incorrect scale-ups (false positives) or prevent correct scale-downs (false negatives). Google Cloud Dataflow [95] relies on CPU utilization for scale-down decisions only but still suffers false negatives. Dhalion [66] and IBM Streams [71] also use congestion and back-pressure signals to identify bottlenecks. These metrics are helpful for identifying bottlenecks but they cannot detect resource over-provisioning.

Predictive policy controllers build an analytical performance model of the streaming system and formulate the scaling problem as a set of mathematical functions. Predictive approaches include queuing theory [67, 111, 151], control theory [18, 93, 115], and instrumentation-driven linear performance models [89]. Thanks to their closed-form analytical formulation, predictive policies are capable of making multi-operator decisions in one step.

6.3.2 Elasticity mechanisms

Elasticity mechanisms are concerned with realizing the actions indicated by the policy. They need to ensure correctness and low-latency redistribution of accumulated state when effecting a reconfiguration. To ensure correctness, many streaming systems rely on the fault-tolerance mechanism to provide reconfiguration capabilities. When adding new workers to a running computation, the mechanism needs not only re-assign work to them but also migrate any necessary state that these new workers will now be in charge of. Elasticity mechanisms need to complete a reconfiguration as quickly as possible and at the same time minimize performance disruption. We review the main methods for state redistribution, reconfiguration, and state transfer next. We focus on systems with embedded state, as reconfiguration mechanisms are significantly simplified when state is external.

State redistribution State redistribution must preserve key semantics, so that existing state for a particular key and all

Table 5 Elasticity policies and mechanisms in streaming systems

System	Policy		Objective		Reconfiguration			State migration	
	Heuristic	Predictive	Latency	Throughput	Stop-and-restart	Partial pause	Live	At-Once	Progressive
Borealis [11]	✓		✓	✓	n/a			n/a	
StreamCloud [76]	✓			✓		✓		✓	
Seep [64]	✓		✓	✓		✓		✓	
IBM Streams [71]	✓			✓		✓		✓	
FUGU [79, 80]	✓			✓		✓		✓	
Nephele [111]		✓	✓						
DRS [67]		✓	✓						
MPC [115]		✓	✓			✓		✓	
CometCloud [151]		✓	✓				✓	n/a	
Chronostream [160]	n/a		n/a				✓	✓	
ACES [18]		✓	✓	✓	n/a			n/a	
Stella [161]	✓			✓					
Google Dataflow [95]	✓		✓	✓					
Dhalion [66]	✓			✓	✓			✓	
DS2 [89]		✓		✓	✓			✓	
Spark Streaming [23, 163]	✓			✓	✓			✓	
Megaphone [84]						✓			✓
Turbine [117]	✓			✓	✓			✓	
Rhino [119]	n/a		n/a			✓		✓	

future events with this key are routed to the same worker. For that purpose, most systems use hashing methods. *Uniform hashing* evenly distributes keys across parallel tasks. It is fast to compute and requires no routing state but might incur high migration cost. When a new node is added, state is shuffled across existing and new workers. It also causes random I/O and high network communication. Thus, it is not particularly suitable for adaptive applications. *Consistent hashing* and variations are more often preferred. Workers and keys are mapped to multiple points on a ring using multiple random hash functions. Consistent hashing ensures that state is not moved across workers that are present before and after the migration. When a new worker joins, it becomes responsible for data items from multiple of the existing nodes. When a worker leaves, its key space is distributed over existing workers. Apache Flink [37] uses a variation of consistent hashing in which state is organized into *key groups* and those are mapped to parallel tasks as ranges. On reconfiguration, reads are sequential within each key group, and often across multiple key groups. The metadata of key-group-to-task assignments are small and it is sufficient to store key-group range boundaries. The number of key groups limits the maximum number of parallel tasks to which keyed state can be scaled.

Hashing techniques are simple to implement and do not require storing any routing state, however, they do not perform well under skewed key distributions. *Hybrid partitioning* [70] combines consistent hashing and an explicit mapping to generate a compact hash function that provides load balance in the presence of skew. The main idea is to track the frequencies of the partitioning key values and treat normal keys and popular keys differently. The mechanism uses the lossy counting algorithm [114] in a sliding window setting to estimate heavy hitters, as keeping exact counts would be impractical for large key domains. DKG [133] is a similar key-grouping mechanism that explicitly maps popular keys to sub-streams together with groups of less popular keys to achieve load balance.

Reconfiguration strategy Regardless of the re-partitioning strategy used, if the elasticity policy makes a decision to change an application's resources, the mechanism will have to transfer some amount of state across workers on the same or different physical machines.

The *stop-and-restart* strategy halts the computation, takes a state snapshot of all operators, and then restarts the application with the new configuration. Even though this mechanism is simple to implement and it trivially guarantees correctness, it unnecessary stalls the entire pipeline even if only one or few operators need to be rescaled. As shown in Table 5, this strategy is common in modern systems.

Partial pause and restart, introduced by FLUX [137], is a less disruptive strategy that only blocks the affected dataflow subgraph temporarily. The affected subgraph con-

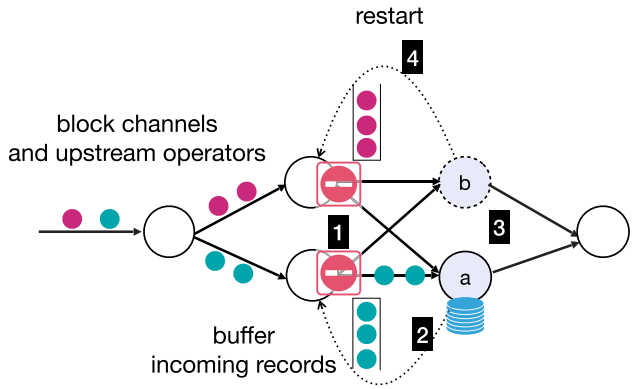


Fig. 8 An example of the partial-pause-and-restart protocol. To move state from operator a to b, the mechanism executes the following steps: (1) Pause a's upstream operators, (2) extract state from a, (3) load state into b, and (4) send a restart signal from b to upstream operators

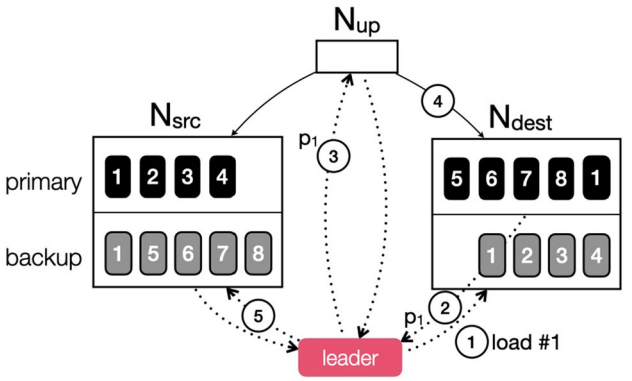


Fig. 9 An example of the proactive replication protocol. To move slice #1 from N_{src} to N_{dest}, the mechanism executes the following steps: (1) the leader instructs N_{dest} to load slice #1, (2) N_{dest} loads slice #1 and sends ack to the leader, (3) the leader notifies upstream operators to replay events, (4) upstream start rerouting events to N_{dest}, (5) the leader notifies N_{src} that the transfer is complete and N_{src} moves slice #1 to the backup group.

tains the operator to be scaled, as well as upstream channels and upstream operators. Figure 8 shows an example of the protocol. To migrate state from operator a to operator b, the mechanism will execute the following steps: (1) First, it *pauses* a's upstream operators and stops pushing tuples to a. Paused operators start buffering input tuples in their local buffers. operator a continues processing tuples in its buffers until they are empty. (2) Once a's buffers are empty, it extracts its state and sends it to operator b. (3) Operator b loads the state and (4) sends a *restart* signal to upstream operators. Once upstream operators receive the signal they can start processing tuples again.

Systems like ChronoStream [160] and CometCloud [151] perform reconfiguration in a nearly *live* manner by leveraging a proactive replication strategy. The core idea is to maintain state backup copies in multiple nodes. To this end, state is organized into smaller partitions, each of which can be transferred independently. Nodes have a set of primary state slices and a set of secondary state slices. Figure 9 shows an example of ChronoStream's protocol.

State transfer Another important decision to make when migrating state from one worker to another is whether the state is moved *all-at-once* or in a *progressive* manner. If a large amount of state needs to be transferred, moving it in one operation might cause high latency during re-configuration. Alternatively, *progressive* migration [84] moves state in smaller pieces and flattens latency spikes by interleaving state transfer with processing. On the downside, progressive state migration might lead to longer migration duration.

6.4 First generation versus second generation

Comparing early to modern approaches, we make the following observations. While load shedding was popular among early stream processors, modern systems do not favor the approach of degrading result quality anymore. Another important difference is that load management approaches in first-generation systems used to affect the execution of multiple queries as they formed a shared dataflow plan (cf. Sect. 2). Queries in modern systems are typically executed as independent jobs, thus, back-pressure on a certain query will not affect the execution of other queries running on the same cluster. Scaling down is a quite recent requirement that was not a matter of concern before cloud deployments. The dependence on persistent queues for providing correctness guarantees is another recent characteristic, mainly required by systems employing back-pressure. Finally, while early load shedding and load-aware scheduling techniques assume a limited set of operators whose properties and characteristics are stable throughout execution, modern systems implement general load management methods that are applicable even if cost and selectivity vary or are unknown.

6.5 Open problems

Adaptive scheduling methods have been studied so far in the context of simple query plans with operators whose selectivities and costs are fixed and known. It is unclear whether these methods generalize to arbitrary plans, operators with UDFs, general windows, and custom joins. Load-aware scheduling can further cause starvation and increased per-tuple latency, as low-priority operators with records in their input buffers would need to wait a long time during bursts. Finally, existing methods are restricted to streams that arrive in timestamp order and do not support out-of-order or delayed events.

Re-configurable stream processing is a quite recent research area, where stream processors are designed to not only be capable of adjusting their resource alloca-

tion but other elements of their runtime as well. Elasticity, the ability of a stream processor to dynamically adjust resource allocation can be considered as a special case of re-configuration. Others include code updates for bug fixes, version upgrades, or business-logic changes, execution-plan switching , dynamic scheduling and operator placement, as well as skew and straggler mitigation. So far, each of the aforementioned re-configuration scenarios have been largely studied in isolation. To provide general re-configuration and self-management, future systems will need to take into account how optimizations interact with each other.

7 Lessons learned and the road ahead

7.1 Discussion of design considerations

In the previous sections, we examined the evolution of progress tracking, state management, fault tolerance, and load management throughout the generations of streaming systems. While we have so far discussed each of these concerns in isolation, in practice, architectural decisions often have to simultaneously consider multiple of these aspects. In the following, we discuss how functional choices for handling time, state, fault tolerance, and reconfiguration can impact one another or become incompatible.

Managing event order and timeliness does not only affect semantics and result completeness, but can also have significant impact on the design of the state management and fault tolerance components. To illustrate this association, let us consider systems employing low watermarks for managing disorder. Watermarks inherently encode a trade-off between latency and result completeness, which, in turn directly affects the size of state that needs to be buffered and checkpointed. Slow watermarks can potentially lead to higher state size and longer checkpoint duration. The architecture of the watermark propagation mechanism can also affect the recovery and reconfiguration duration. External watermark management, as in Google Dataflow [15], enables faster recovery and reconfiguration, as the central authority can be queried to retrieve progress information. In contrast, in-band mechanisms, such as the one in Flink, need to propagate watermarks from the sources to the affected parts of the dataflow. Interestingly, we observe that systems with external state management tend to rely on external progress tracking as well, though in-band approaches are not incompatible.

It comes as no surprise that the approach to state management must be designed with fault tolerance and reconfiguration as first-class concerns. In fact, modern systems often rely on the same mechanism (e.g. consistent snapshots) for fault tolerance and reconfiguration. Decoupling state from compute substantially simplifies recovery and load management, as the external state store and the compute resources can be scaled and configured independently. Finally, we note that the load management approach directly impacts the result semantics. For example, load shedding is inherently incompatible with exactly-once guarantees.

7.2 Evolution take-aways

The typical stream processing system architecture has evolved significantly over the last three decades. While early systems extended relational execution engines with data streaming capabilities, the design of modern systems evolved to address new application demands and exploit advances in cloud computing and hardware. Table 6 summarizes our main findings concerning the evolution of stream processing systems. In this section, we aim to shed light on the reasons why some approaches persisted throughout generations, while others had to be adjusted or abandoned.

While early systems often returned approximate results, later generations rejected the notion that stream processing is a synonym of approximate computation. In particular, streaming systems that originated as extensions of the MapReduce model focused on providing exact and correct results, even under failures. To some extent, approximate pro-

Table 6 Evolution of streaming systems

	1st generation	2nd–3rd generation
Results	Approximate or exact	Exact
Programming interface	SQL extensions, CQL	UDF-heavy—Java, Scala, Python, SQL-like, etc.
Query plans	Global, optimized, with pre-defined operators	Independent, with custom operators
Query execution	(Mostly) scale-up	Distributed
Parallelism	Pipeline	Data, pipeline, task
Time and progress	Heartbeats, slack, punctuations	Low-watermark, frontiers
State management	Shared synopses, in-memory	Per query, partitioned, persistent, larger-than-memory
Fault tolerance	HA-focused, limited correctness guarantess	Distributed snapshots, exactly-once
Load management	Load shedding, load-aware scheduling	Backpressure, elasticity

cessing was a necessity in early systems that were deployed on restricted resources. By carefully dropping some events or emitting early incomplete results, these systems achieved high availability and low latency. In contrast, streaming systems that were designed for cloud environments could leverage the capability of adjusting their resource requirements according to the workload. As a result, these systems persist or redistribute excess load to avoid data loss and guarantee exact outputs. Despite differences in the reaction mechanism, we emphasize that the problem of detecting and quantifying overload is fundamental in both early and modern systems and it has been consistently addressed with continuous monitoring and feedback control.

In terms of programming interface, we observe a full circle. As first-generation streaming systems evolved from database management systems, the first programming interfaces for data stream queries were designed around SQL-like languages. On the other hand, second-generation systems are UDF-centric and favor general-purpose programming languages, such as Java, Scala, and Python. However, as stream processing tools are becoming widespread, we witness a trend to return to extensions for streaming SQL [32] to accommodate a larger variety of users and use cases.

Over the years, the design of streaming query execution engines has also gradually transitioned from mainly centralized to mainly distributed, exploiting data, pipeline, and task parallelism. Most modern streaming systems target shared-nothing in-house or on-cloud clusters. This shift has also impacted architectural decisions in query scheduling, optimization, and deployment. 1st-generation systems commonly share resources among multiple queries that are jointly optimized and executed. On the contrary, modern systems provide per-query resource allocation and guide users to develop and deploy independent applications, even if they ingest events from common sources. While this prevailing approach may lead to higher resource requirements and redundant computation, it offers more flexibility. Separate query deployments enable faster and easier fault recovery and reconfiguration.

Regarding time, order, and progress, many of the inventions of the past proved to have survived the test of time, since they continue to hold a place in modern streaming systems. In particular, Millwheel and the Google Dataflow Model popularized punctuations, watermarks, the out-of-order architecture, and triggers for revision processing. Streaming state management witnessed a major shift, from specialized in-memory synopses to large partitioned and persistent state supported today. To some extent, this change is a consequence of shifting from approximate to exact computation and from centralized to distributed and cloud-based deployments. As a result, fault tolerance and high availability also shifted towards passive replication and exactly-once processing. Many approaches to fault tolerance and high availability that are in use today, such as active and passive replication and upstream backup, were already proposed in 1st-generation systems. However, later generations refined these techniques and extended them to guarantee exactly-once semantics.

In state management, we identify the most radical changes seen in data streaming so far. The most obvious advances relate to the scalability of state and long-term persistence in unbounded executions. Today's systems have invested thoroughly in providing transactional guarantees that are in par with those of current database management systems. Transactional stream processing has pivoted data streaming beyond data analytics use cases and has also opened new research directions in terms of efficient methods for backing and accessing state that grows in unbounded terms. Stream state and compute are gradually being decoupled, and this trend allows for better optimizations, wider interoperability with storage technologies as well as novel semantics for shared and external state.

7.3 A future outlook

As we detailed throughout this survey, the landscape of data streaming systems has changed significantly in recent years. Yet, the road ahead for data streaming systems is still evidently long and full of transformations. We list out ongoing and future trends in stream processing in the key categories of: serverless/cloud, query capabilities, edge and hardware acceleration.

Serverless and cloud The advent of serverless computing has introduced new opportunities for processing and analyzing data streams with greater flexibility and cost efficiency. However, it also brings forth new challenges, such as handling stateful operations, orchestrating and managing resources in a highly volatile distributed environment, and ensuring low-latency processing in the face of the inherent unpredictability of serverless platforms. As data streaming applications increasingly adopt serverless architectures, addressing these challenges becomes crucial to harness the full potential of serverless computing for real-time data processing and analytics. The ability to tune scheduling of stream tasks is also particularly promising in serverless and cloud deployments, where virtualization necessitates runtime adaptation capabilities [110]. Lachesis [127] is one example of a middleware designed for scheduling stream programs at runtime. Meces [74] presents an effective method to frequently migrate state across dynamic reconfigurations of stream processing pipelines, whereas, Xu et.al [162] also investigate the prospect of maintaining SLAs in multi-tenant environments. Future research in serverless and cloud adaptation for streaming systems is essential to gain a deeper understanding of the performance trade-offs associated with state management

approaches, as well as to explore reconfiguration strategies in the context of virtualized and highly dynamic infrastructures.

Query capabilities With the increased adoption of stateful stream engines, a recent research focus targets the enhancement of the stateful capabilities beyond local-only dataflow access to the states. S-Query [159] examines the possibility of exposing operator states for external queries with different isolation levels. Snapshotted state enables read-only queries using snapshot isolation. Provenance in stream queries is another emergent aspect of stream processing. Provenance has multiple uses that are particularly interesting in stream execution, such as explainability as well as profiling of streaming queries. Ananke [126] is one example of a provenance execution strategy. Future research on query capabilities will further need to address the challenges of SQL integration (e.g., streams and tables [32]) and develop new standards in how users can interact with stream pipelines, beyond the restricted dataflow model.

Programming models for stateful serveless functions Streaming dataflow systems at the moment are only programmable through functional-programming style dataflow APIs. As a result, general cloud applications such as payment processing, reservation systems, inventory keeping, and low-latency business workflows need to be rewritten by programmers to match the event-driven dataflow paradigm. Although it is possible to rewrite lots of applications in this paradigm, it takes a considerable amount of programmer training and effort to do so. We argue that streaming dataflow systems would benefit from new programming abstractions [128], for them to be adopted by programmers for general cloud applications.

Transactional serverless functions Although it has been shown that it is possible to introduce transaction coordination among stateful functions running on top of a streaming dataflow system [60], we argue that a more efficient implementation of transactions would require the dataflow system to be aware of transaction boundaries and to incorporate transaction processing into its state management and fault-tolerance protocols. The most important reason for using dataflow systems for general cloud applications, is that modern streaming dataflow systems offer message processing guarantees (exactly-once processing). As a result, programmers do not need to "pollute" their business logic with consistency checks, state rollbacks, timeouts, retries, and idempotency [94, 103].

Edge streaming The emergence of 5 G and compatible edge hardware has created great interest in moving data stream pipelines to edge nodes (e.g., sensor devices, wearables, mobiles, base-stations etc.). Moving processing closer to the sources can lead to unmatched low processing-latency and create pre-processing cost savings. Two major challenges that are being addressed in this setting are low-end IoT hardware execution and the decentralization of data streaming pipelines. Edgewise [68] proposes a new schedulling methodology for streaming tasks on IoT nodes employing a more resource-lightweight processing and IO without heavy compromises in performance. Similarly, Hazelcast Jet [72] removes the cost of heavy persistent state and transactional guarantees to compromise for low-latency and in-memory processing. Portals [140, 141] proposes the use of pre-defined transactional processing by dividing streams into atomic segments called "atoms". Atoms in portals instrument transactional execution when different stream pipelines across edge and cloud need to communicate data. Future research in edge streaming will continue to investigate lightweight techniques and their integration with data streaming systems to enable efficient and low-latency processing in resource-constrained environments.

Hardware acceleration Optimized code generation in stream processing is currently on the rise due to the increased adoption of accelerators in the form of GPUs and FPGAs, as well as the proliferation of enhanced memory and storage technologies, such as NVMEs. Among ongoing examples, Brisk [166] is a showcases of stream scalability on shared-memory multicore architectures. FineStream [165] enhances the performance of window-based execution for CPU-GPU integrated architectures. A typical challenge when interfacing with GPUs is the need to write custom driver instructions for specific stream operators. One practical direction in hardware acceleration is automated GPU execution directly from translated bytecode code fragments in JVM-specific streaming frameworks (e.g., Flink) as demonstrated in the Tornado/GraalVM projects [97]. The prospect of using FPGAs for data streaming has also been demonstrated in the Fleet project [149]. Future research will need to explore the potential of emerging hardware technologies, such as RISC-V instruction sets, as well as new paradigms. Near-Memory Computing (NMC) [139] is one such paradigm that is particularly attractive for stream processing, which aims to co-locate computing tasks with their corresponding address space in the same chip, thereby, avoiding the Von-Neumann data transfer bottleneck between RAM and CPU, which is common in streaming applications.

Acknowledgements We thank the anonymous VLDBJ reviewers for their detailed and valuable feedback on prior drafts of this paper. This work was partially supported by a Google DAPA award, WASP NESTS (Data-Bound Computing), and the Dutch Research Council (NWO) Vidi project No. 19708.

References

1. Apache Flink. http://flink.apache.org/. Last access: July (2023)
2. Apache flink statefun documentation. https://nightlies.apache.org/flink/flink-statefun-docs-stable/. Last access: 2023-04-08
3. Apache Storm. http://storm.apache.org/. Last access: July (2023)
4. Arroyo. https://github.com/ArroyoSystems/arroyo. Last access: July (2023)
5. Introduction to Kafka Streams. http://www.confluent.io/blog/introducing-kafka-streams-stream-processing-_made-simple. Last access: July (2023)
6. Materialize documentation. https://materialize.com/docs/. Last access: July (2023)
7. Redis. https://redis.io/. Last access: July (2023)
8. Risingwave. https://github.com/risingwavelabs/risingwave. Last access: July (2023)
9. The Trident Stream Processing Programming Model. http://storm.apache.org/releases/0.10.0/Trident-tutorial.html. Last access: July (2023)
10. Unaligned Checkpoints - Flink. https://nightlies.apache.org/flink/flink-docs-release-1.17/docs/ops/state/checkpointing_under_backpressure/#unaligned-checkpoints. Last access: July (2023)
11. Abadi, D.J., Ahmad, Y., Balazinska, M., Çetintemel, U., Cherniack, M., Hwang, J., Lindner, W., Maskey, A., Rasin, A., Ryvkina, E., Tatbul, N., Xing, Y., Zdonik, S.B.: The design of the borealis stream processing engine. In Second Biennial Conference on Innovative Data Systems Research, CIDR 2005, Asilomar, CA, USA, pp. 277–289, (2005)
12. Abadi, D.J., Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., Zdonik, S.B.: Aurora: a new model and architecture for data stream management. VLDB J. **12**(2), 120–139 (2003)
13. Akhter, A., Fragkoulis, M., Katsifodimos, A.: Stateful functions as a service in action. Proc. VLDB Endow. **12**(12), 1890–1893 (2019)
14. 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)
15. Akidau, T., Begoli, E., Chernyak, S., Hueske, F., Knight, K., Knowles, K., Mills, D., Sotolongo, D.: Watermarks in stream processing systems: semantics and comparative analysis of apache flink and google cloud dataflow. Proc. VLDB Endow. **14**(12), 3135–3147 (2021)
16. Akidau, T., Bradshaw, R., Chambers, C., Chernyak, S., Fernández-Moctezuma, R.J., Lax, R., McVeety, S., Mills, D., Perry, F., Schmidt, E. et al.: The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. In: VLDB, (2015)
17. Ali, M., Chandramouli, B., Goldstein, J., Schindlauer, R.: The extensibility framework in Microsoft StreamInsight. In 2011 IEEE 27th International Conference on Data Engineering, pp. 1242–1253. IEEE, (2011)
18. Amini, L., Jain, N., Sehgal, A., Silber, J., Verscheure, O.: Adaptive control of extreme-scale stream processing systems. In 26th IEEE International Conference on Distributed Computing Systems (ICDCS'06), pp. 71–71. IEEE, (2006)
19. Arasu, A., Babcock, B., Babu, S., Cieslewicz, J., Datar, M., Ito, K., Motwani, R., Srivastava, U., Widom, J.: Stream: the Stanford data stream management system. Data Stream Manage. Process. High-Speed Data Streams, 317–336 (2016) <error l="303" c="Invalid <error l="301" c="Invalid <error l="302" c="Invalid command: paragraph not started." /> command: paragraph not started." /> command: paragraph not started." />
20. Arasu, A., Babcock, B., Babu, S., Datar, M., Ito, K., Nishizawa, I., Rosenstein, J., Widom, J.: STREAM: the stanford stream data manager. In Halevy, A.Y., Ives, Z.G., Doan, A. (eds.) Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003, p. 665. ACM, (2003)
21. Arasu, A., Babu, S., Widom, J.: The CQL continuous query language: semantic foundations and query execution. VLDB J. **15**(2), 121–142 (2006)
22. Arasu, A., Widom, J.: Resource sharing in continuous sliding-window aggregates. In Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB, pp. 336–347 (2004)
23. Armbrust, M., Das, T., Torres, J., Yavuz, B., Zhu, S., Xin, R., Ghodsi, A., Stoica, I., Zaharia, M.: In Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, pp. 601–613. ACM, (2018)
24. Asyabi, E., Wang, Y., Liagouris, J., Kalavri, V., Bestavros, A.: A new benchmark harness for systematic and robust evaluation of streaming state stores. In Proceedings of the Seventeenth European Conference on Computer Systems, EuroSys '22, pp. 559-574, New York, NY, USA, (2022). Association for Computing Machinery
25. Avnur, R., Hellerstein, J. M.: Eddies: Continuously adaptive query processing. In Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, pp. 261–272. ACM, (2000)
26. Babcock, B., Babu, S., Datar, M., Motwani, R.: Chain: operator scheduling for memory minimization in data stream systems. In Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, pp. 253–264. ACM, (2003)
27. Babcock, B., Babu, S., Datar, M., Motwani, R., Widom, J.: Models and issues in data stream systems. In Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, pp. 1–16. ACM, (2002)
28. Babcock, B., Datar, M., Motwani, R.: Load shedding for aggregation queries over data streams. In Proceedings of the 20th International Conference on Data Engineering, ICDE 2004, pp. 350–361, (2004)
29. Babu, S., Motwani, R., Munagala, K., Nishizawa, I., Widom, J.: Adaptive ordering of pipelined stream filters. In Proceedings of the ACM SIGMOD International Conference on Management of Data, pp 407–418, (2004)
30. Balazinska, M., Balakrishnan, H., Madden, S.R., Stonebraker, M.: Fault-tolerance in the Borealis distributed stream processing system. ACM TODS **33**(1), 44 (2008)
31. Barga, R.S., Goldstein, J., Ali, M.H., Hong, M.: Consistent streaming through time: a vision for event stream processing. In Third Biennial Conference on Innovative Data Systems Research, CIDR pp. 363–374 (2007)
32. Begoli, E., Akidau, T., Hueske, F., Hyde, J., Knight, K., Knowles, K.L.: One SQL to rule them all - an efficient and syntactically idiomatic approach to management of streams and tables. In Pro-

ceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, pp. 1757–1772. ACM
33. Botan, I., Derakhshan, R., Dindar, N., Haas, L.M., Miller, R.J., Tatbul, N.: SECRET: a model for analysis of the execution semantics of stream processing systems. Proc. VLDB Endow. **3**(1), 232–243 (2010)
34. Burckhardt, S., Gillum, C., Justo, D., Kallas, K., McMahon, C., Meiklejohn, C.S.: Durable functions: semantics for stateful serverless. Proc. ACM Program. Lang. **5**(OOPSLA), 1–27 (2021)
35. Carbone, P.: Scalable and Reliable Data Stream Processing. PhD thesis, KTH Royal Institute of Technology (2018)
36. 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)
37. Carbone, P., Ewen, S., Haridi, S., Katsifodimos, A., Markl, V., Tzoumas, K.: Apache Flink: Stream and batch processing in a single engine. IEEE Data Eng. Bull., **38** (2015)
38. Carbone, P., Fóra, G., Ewen, S., Haridi, S., Tzoumas, K.: Lightweight asynchronous snapshots for distributed dataflows. (2015) arXiv preprint arXiv:1506.08603
39. Carbone, P., Fragkoulis, M., Kalavri, V., Katsifodimos, A.: Beyond analytics: The evolution of stream processing systems. In Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, pp. 2651–2658
40. Carney, D., Çetintemel, U., Rasin, A., Zdonik, S. B., Cherniack, M., Stonebraker, M.: Operator scheduling in a data stream manager. In Proceedings of 29th International Conference on Very Large Data Bases, VLDB 2003, pp. 838–849, (2003)
41. Çetintemel, U., Abadi, D., Ahmad, Y., Balakrishnan, H., Balazinska, M., Cherniack, M., Hwang, J.-H., Madden, S., Maskey, A., Rasin, A. et al.: The aurora and borealis stream processing engines. In Data Stream Management, pp. 337–359. Springer (2016)
42. Çetintemel, 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.B.: S-store: a streaming newsql system for big velocity applications. Proc. VLDB Endow. **7**(13), 1633–1636 (2014)
43. Chandramouli, B., Goldstein, J.: Shrink—prescribing resiliency solutions for streaming. Proc. VLDB Endow. **10**(5), 505–516 (2017)
44. Chandramouli, B., Goldstein, J., Barnett, M., DeLine, R., Platt, J.C., Terwilliger, J.F., Wernsing, J.: Trill: a high-performance incremental query processor for diverse analytics. Proc. VLDB Endow. **8**(4), 401–412 (2014)
45. Chandramouli, B., Goldstein, J., Li, Y.: Impatience is a virtue: revisiting disorder in high-performance log analytics. In 34th IEEE International Conference on Data Engineering, ICDE 2018, pp. 677–688, (2018)
46. Chandramouli, B., Goldstein, J., Maier, D.: On-the-fly progress detection in iterative stream queries. Proc. VLDB Endow. **2**(1), 241–252 (2009)
47. Chandramouli, B., Prasaad, G., Kossmann, D., Levandoski, J.J., Hunter, J., Barnett, M.: FASTER: a concurrent key-value store with in-place updates. In Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, pp. 275–290
48. Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M.J., Hellerstein, J.M., Hong, W., Krishnamurthy, S., Madden, S., Reiss, F., Shah, M.A.: TelegraphCQ: Continuous dataflow processing. In Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, p. 668
49. Chandy, K.M., Lamport, L.: Distributed snapshots: determining global states of distributed systems. ACM Trans. Comput. Syst. (TOCS) **3**(1), 63–75 (1985)
50. 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 TOCS **26**(2), 26 (2008)
51. Chen, J., DeWitt, D.J., Tian, F., Wang, Y.: NiagaraCQ: a scalable continuous query system for internet databases. pp. 379–390
52. Cherniack, M., Balakrishnan, H., Balazinska, M., Carney, D., Çetintemel, U., Xing, Y., Zdonik, S. B.: Scalable distributed stream processing. In First Biennial Conference on Innovative Data Systems Research, CIDR (2003)
53. Corbett, J.C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J.J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., et al.: Spanner: Google's globally distributed database. ACM TOCS **31**(3), 22 (2013)
54. Cranor, C. D., Johnson, T., Spatscheck, O., Shkapenyuk, V.: Gigascope: A stream database for network applications. In Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, pp. 647–651
55. Cugola, G., Margara, A.: Processing flows of information: from data stream to complex event processing. ACM Comput. Surv. **44**(3), 61 (2012)
56. da Silva, G.J., Zheng, F., Debrunner, D., Wu, K., Dogaru, V., Johnson, E., Spicer, M., Sariyüce, A.E.: Consistent regions: guaranteed tuple processing in IBM streams. Proc. VLDB. Endow. **9**(13), 1341–1352 (2016)
57. Das, A., Gehrke, J., Riedewald, M.: Approximate join processing over data streams. In Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, pp. 40–51
58. Dayarathna, M., Perera, S.: Recent advancements in event processing. ACM Comput. Surv. **51**(2), 35 (2018)
59. de Assuncao, M.D., da Silva Veith, A., Buyya, R.: Distributed data stream processing and edge computing: a survey on resource elasticity and future directions. J. Netw. Comput. Appl. **103**, 1–17 (2018)
60. de Heus, M., Psarakis, K., Fragkoulis, M., Katsifodimos, A.: Distributed transactions on serverless stateful functions. In Proceedings of the 15th ACM International Conference on Distributed and Event-based Systems, pp. 31–42 (2021)
61. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. In 6th Symposium on Operating System Design and Implementation (OSDI 2004), pp. 137–150. USENIX Association
62. Elnozahy, E.N.M., Alvisi, L., Wang, Y.-M., Johnson, D.B.: A survey of rollback-recovery protocols in message-passing systems. ACM Comput. Surv. **34**(3), 34 (2002)
63. Farhat, O., Daudjee, K., Querzoni, L.: Klink: Progress-aware scheduling for streaming data systems. In Proceedings of the 2021 ACM SIGMOD International Conference on Management of Data, pp. 485–498
64. Fernandez, R.C., Migliavacca, M., Kalyvianaki, E., Pietzuch, P.R.: Integrating scale out and fault tolerance in stream processing using operator state management. In Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD (2013), pp. 725–736
65. Fernandez, R.C., Migliavacca, M., Kalyvianaki, E., Pietzuch, P.R.: Making state explicit for imperative big data processing. In 2014 USENIX Annual Technical Conference, USENIX ATC '14, pp. 49–60
66. Floratou, A., Agrawal, A., Graham, B., Rao, S., Ramasamy, K.: Dhalion: self-regulating stream processing in heron. Proc. VLDB Endow. **10**(12), 1825–1836 (2017)
67. Fu, T.Z.J., Ding, J., Ma, R.T.B., Winslett, M., Yang, Y., Zhang, Z.: DRS: auto-scaling for real-time stream analytics. IEEE/ACM Trans. Netw. **25**(6), 15 (2017)

68. Fu, X., Ghaffar, T., Davis, J.C., Lee, D.: Edgewise: a better stream processing engine for the edge. In 2019 USENIX Annual Technical Conference, USENIX ATC 2019, pp. 929–946 (2019)
69. Garofalakis, M., Gehrke, J., Rastogi, R.: Data stream management: processing high-speed data streams. Springer, Berlin (2007)
70. Gedik, B.: Partitioning functions for stateful data parallelism in stream processing. VLDBJ **23**(4), 517–539 (2014)
71. Gedik, B., Schneider, S., Hirzel, M., Wu, K.L.: Elastic scaling for data stream processing. IEEE Trans. Parallel Distrib. Syst. **25**(6), 17 (2014)
72. Gencer, C., Topolnik, M., Durina, V., Demirci, E., Kahveci, E.B., Gürbüz, A., Bartók, J., Gierlach, G., Hartman, F., Yilmaz, U., Lukás, O., Dogan, M., Mandouh, M., Fragkoulis, M., Katsifodimos, A.: Hazelcast jet Low-latency stream processing at the 99.99th percentile. Proc. VLDB Endow. **14**(12), 3110–3121 (2021)
73. Gray, J., Siewiorek, D.P.: High-availability computer systems. Computer **24**(9), 10 (1991)
74. Gu, R., Yin, H., Zhong, W., Yuan, C., Huang, Y.: Meces: latency-efficient rescaling via prioritized state migration for stateful distributed stream processing systems. In: USENIX Annual Technical Conference. USENIX ATC, pp. 539–556 (2022)
75. Gu, Y., Zhang, Z., Ye, F., Yang, H., Kim, M., Lei, H., Liu, Z.: An empirical study of high availability in stream processing systems. In Middleware 2008, ACM/IFIP/USENIX 9th International Middleware Conference, p. 23
76. Gulisano, V., Jiménez-Peris, R., Patiño-Martínez, M., Soriente, C., Valduriez, P.: StreamCloud: an elastic and scalable data streaming system. IEEE Trans. Parallel Distrib. Syst. **23**(12), 15 (2012)
77. Hammad, M.A., Franklin, M.J., Aref, W.G., Elmagarmid, A.K.: Scheduling for shared window joins over data streams. In Proceedings of 29th International Conference on Very Large Data Bases, VLDB (2003), pp. 297–308
78. Heinze, T., Aniello, L., Querzoni, L., Jerzak, Z.: Cloud-based data stream processing. In The 8th ACM International Conference on Distributed Event-Based Systems, DEBS '14, pp. 238–245
79. Heinze, T., Jerzak, Z., Hackenbroich, G., Fetzer, C.: Latency-aware elastic scaling for distributed data stream processing systems. In The 8th ACM International Conference on Distributed Event-Based Systems, DEBS '14, pp. 13–22
80. Heinze, T., Pappalardo, V., Jerzak, Z., Fetzer, C.: Auto-scaling techniques for elastic data stream processing. In The 8th ACM International Conference on Distributed Event-Based Systems, DEBS '14, pp. 318–321
81. Heinze, T., Zia, M., Krahn, R., Jerzak, Z., Fetzer, C.: An adaptive replication scheme for elastic data stream processing systems. In Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems, DEBS '15, pp. 150–161
82. Hirzel, M., Baudart, G., Bonifati, A., Della Valle, E., Sakr, S., Akrivi Vlachou, A.: Stream processing languages in the big data era. SIGMOD Record, 47(2), (2018)
83. Hirzel, M., Soulé, R., Schneider, S., Gedik, B., Grimm, R.: A catalog of stream processing optimizations. ACM Comput. Surv. **46**(4), 34 (2014)
84. Hoffmann, M., Lattuada, A., McSherry, F., Kalavri, V., Liagouris, J., Roscoe, T.: Megaphone: Latency-conscious state migration for distributed streaming dataflows. Proc. VLDB Endow. **12**(9), 1002–1015 (2019)
85. Hwang, J., Balazinska, M., Rasin, A., Çetintemel, U., Stonebraker, M., Zdonik, S.B.: High-availability algorithms for distributed stream processing. In Proceedings of the 21st International Conference on Data Engineering, ICDE 2005, pp. 779–790
86. Hwang, J., Xing, Y., Çetintemel, U., Zdonik, S.B.: A cooperative, self-configuring high-availability solution for stream processing. In Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, pp. 176–185
87. Johnson, T., Muthukrishnan, S., Shkapenyuk, V., Spatscheck, O.: A heartbeat mechanism and its application in gigascope. In Proceedings of the 31st International Conference on Very Large Data Bases, pp. 1079–1088, (2005)
88. Kalavri, V., Liagouris, J.: In support of workload-aware streaming state management. In 12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20), (2020)
89. Kalavri, V., Liagouris, J., Hoffmann, M., Dimitrova, D.C., Forshaw, M., Roscoe, T.: Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows. In 13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, pp. 783–798
90. Kang, J., Naughton, J.F., Viglas, S.: Evaluating window joins over unbounded streams. In Proceedings of the 19th International Conference on Data Engineering, pp. 341–352
91. Katsifodimos, A., Fragkoulis, M.: Operational stream processing: towards scalable and consistent event-driven applications. In Advances in Database Technology - 22nd International Conference on Extending Database Technology, EDBT 2019, pp. 682–685
92. Katsipoulakis, N.R., Labrinidis, A., Chrysanthis, P.K.: Concept-driven load shedding: reducing size and error of voluminous and variable data streams. In IEEE International Conference on Big Data (IEEE BigData 2018), pp. 418–427
93. Khoshkbarforoushha, A., Khosravian, A., Ranjan, R.: Elasticity management of streaming data analytics flows on clouds. J. Comput. Syst. Sci. **89**, 24–40 (2017)
94. Killalea, T.: The hidden dividends of microservices. ACM Queue, (2016)
95. Kirpichov, E., Denielou, M.: No shard left behind: dynamic work rebalancing in Google Cloud Dataflow. https://cloud.google.com/blog/big-data/2016/05/no-shard-left-behind-dynamic-work-rebalancing-in-google-cloud-dataflow. Last access: July (2023)
96. Kleppmann, M., Beresford, A.R., Svingen, B.: Online event processing: achieving consistency where distributed transactions have failed. ACM Queue, (2019)
97. Kotselidis, C., Diamantopoulos, S., Mylonas, G.: A big data software paradigm for heterogeneous cloud deployments. Inf. Intell. Syst. Appl. **1**(1), 6–10 (2020)
98. Kreps, J., Narkhede, N., Rao, J. et al.: Kafka: a distributed messaging system for log processing. NetDB, (2011)
99. Krishnamurthy, S., Franklin, M.J., Davis, J., Farina, D., Golovko, P., Li, A., Thombre, N.: Continuous analytics over discontinuous streams. In Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, pp. 1081–1092
100. 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, pp. 239–250
101. Kung, H. T., Blackwell, T., Chapman, A.: Credit-based flow control for ATM networks: Credit update protocol, adaptive credit allocation and statistical multiplexing. In Proceedings of the ACM SIGCOMM 1994 Conference on Communications Architectures, Protocols and Applications, pp. 101–114
102. Kwon, Y., Balazinska, M., Greenberg, A.G.: Fault-tolerant stream processing using a distributed, replicated file system. Proc. VLDB Endow. **1**(1), 574–585 (2008)
103. Laigner, R., Zhou, Y., Salles, M.A.V., Liu, Y., Kalinowski, M.: Data management in microservices: State of the practice, challenges, and research directions. Proc. VLDB Endow. **14**(13), (2021)

104. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7), 558–565 (1978)
105. Lee, G., Maeng, J., Park, J., Seo, J., Cho, H., Yang, Y., Um, T., Lee, J., Lee, J.W., Chun, B.-G.: Flowkv: a semantic-aware store for large-scale state management of stream processing engines. In Proceedings of the Eighteenth European Conference on Computer Systems, EuroSys '23, pp. 768–783, New York, NY, USA, (2023). Association for Computing Machinery
106. Leibiusky, J., Eisbruch, G., Simonassi, D.: Getting started with Storm. O'Reilly Media, Inc., (2012)
107. Li, J., Maier, D., Tufte, K., Papadimos, V., Tucker, P.A.: No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. SIGMOD Rec. **34**(1), 39–44 (2005)
108. Li, J., Tufte, K., Shkapenyuk, V., Papadimos, V., Johnson, T., Maier, D.: Out-of-order processing: a new architecture for high-performance stream systems. Proc. VLDB Endow. **1**(1), 274–288 (2008)
109. Lin, W., Fan, H., Qian, Z., Xu, J., Yang, S., Zhou, J., Zhou, L.: Streamscope: continuous reliable distributed processing of big data streams. In 13th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2016, pp. 439–453
110. Liu, X., Buyya, R.: Resource management and scheduling in distributed stream processing systems: a taxonomy, review, and future directions. ACM Comput. Surv. (CSUR) **53**(3), 1–41 (2020)
111. Lohrmann, B., Janacik, P., Kao, O.: Elastic stream processing with latency guarantees. In 35th IEEE International Conference on Distributed Computing Systems, ICDCS 2015, pp. 399–410
112. Mai, L., Zeng, K., Potharaju, R., Xu, L., Suh, S., Venkataraman, S., Costa, P., Kim, T., Muthukrishnan, S., Kuppa, V., et al.: Chi: a scalable and programmable control plane for distributed stream processing systems. Proc. VLDB Endow. **11**(10), 1303–1316 (2018)
113. Maier, D., Li, J., Tucker, P.A., Tufte, K., Papadimos, V.: Semantics of data streams and operators. In Database Theory - ICDT 2005, 10th International Conference, vol. 3363, pp. 37–52
114. Manku, G. S., Motwani, R.: Approximate frequency counts over data streams. In Proceedings of 28th International Conference on Very Large Data Bases, VLDB 2002, pp. 346–357
115. Matteis, T.D., Mencagli, G.: Elastic scaling for distributed latency-sensitive data stream operators. In 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing, PDP 2017, pp. 61–68
116. Meehan, J., Tatbul, N., Zdonik, S., Aslantas, C., Çetintemel, 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)
117. Mei, Y., Cheng, L., Talwar, V., Levin, M.Y., Jacques-Silva, G., Simha, N., Banerjee, A., Smith, B., Williamson, T., Yilmaz, S., Chen, W., Chen, G.J.: Turbine: Facebook's service management platform for stream processing. In 36th IEEE International Conference on Data Engineering, ICDE 2020, pp. 1591–1602
118. Migliavacca, M., Eyers, D., Bacon, J., Papagiannis, Y., Shand, B., Pietzuch, P.: SEEP: scalable and elastic event processing. In Middleware Posters and Demos, (2010)
119. Monte, B.D., Zeuch, S., Rabl, T., Markl, V.: Rhino: efficient management of very large distributed state for stream processing engines. In Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference, pp. 2471–2486
120. Murray, D.G., McSherry, F., Isaacs, R., Isard, M., Barham, P., Abadi, M.: Naiad: a timely dataflow system. In ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP, pp. 439–455
121. Murray, D.G., McSherry, F., Isard, M., Isaacs, R., Barham, P., Abadi, M.: Incremental, iterative data processing with timely dataflow. Commun. ACM **59**(10), 75–83 (2016)
122. Mutschler, C., Philippsen, M.: Reliable speculative processing of out-of-order event streams in generic publish/subscribe middlewares. In The 7th ACM International Conference on Distributed Event-Based Systems, DEBS '13, pp. 147–158
123. Neumeyer, L., Robbins, B., Nair, A., Kesari, A.: S4: distributed stream computing platform. In ICDMW 2010, The 10th IEEE International Conference on Data Mining Workshops, pp. 170–177
124. Noghabi, S.A., Paramasivam, K., Pan, Y., Ramesh, N., Bringhurst, J., Gupta, I., Campbell, R.H.: Stateful scalable stream processing at LinkedIn. Proc. VLDB Endow. **10**(12), 1634–1645 (2017)
125. O'Neil, P., Cheng, E., Gawlick, D., O'Neil, E.: The log-structured merge-tree (lsm-tree). Acta Inf. **33**(4), 351–385 (1996)
126. Palyvos-Giannas, D., Havers, B., Papatriantafilou, M., Gulisano, V.: Ananke: a streaming framework for live forward provenance. Proc. VLDB Endow. **14**(3), 391–403 (2020)
127. Palyvos-Giannas, D., Mencagli, G., Papatriantafilou, M., Gulisano, V.: Lachesis: a middleware for customizing os scheduling of stream processing queries. In Proceedings of the 22nd International Middleware Conference, pp. 365–378 (2021)
128. Psarakis, K., Zorgdrager, W., Fragkoulis, M., Salvaneschi, G., Katsifodimos, A.: Stateful entities: object-oriented cloud applications as distributed dataflows. In 27th International Conference on Extending Database Technology, pp. 15–21, (2024)
129. Qian, Z., He, Y., Su, C., Wu, Z., Zhu, H., Zhang, T., Zhou, L., Yu, Y., Zhang, Z.: Timestream: reliable stream computation in the cloud. In Eighth Eurosys Conference 2013, EuroSys '13, pp. 1–14
130. Raman, V., Raman, B., Hellerstein, J. M.: Online dynamic reordering for interactive data processing. In VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, pp. 709–720, (1999)
131. Rameshan, N., Liu, Y., Navarro, L., Vlassov, V.: Hubbub-scale: towards reliable elastic scaling under multi-tenancy. In IEEE/ACM 16th International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2016, pp. 233–244 (2016)
132. Rivetti, N., Anceaume, E., Busnel, Y., Querzoni, L., Sericola, B.: Online scheduling for shuffle grouping in distributed stream processing systems. In Proceedings of the 17th International Middleware Conference, pp. 1–12, (2016)
133. Rivetti, N., Querzoni, L., Anceaume, E., Busnel, Y., Sericola, B.: Efficient key grouping for near-optimal load balancing in stream processing systems. In Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems, pp. 80–91, (2015)
134. Röger, H., Mayer, R.: A comprehensive survey on parallelization and elasticity in stream processing. ACM Comput. Surv. **52**(2), 1–37 (2019)
135. Ryvkina, E., Maskey, A., Cherniack, M., Zdonik, S. B.: Revision processing in a stream processing engine: a high-level design. In Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, p. 141
136. Shah, M.A., Hellerstein, J.M., Brewer, E.A.: Highly-available, fault-tolerant, parallel dataflows. In Weikum, G., König, A.C. Deßloch, S. (eds.), Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 827–838
137. Shah, M.A., Hellerstein, J.M., Chandrasekaran, S., Franklin, M.J.: Flux: an adaptive partitioning operator for continuous query systems. In Proceedings of the 19th International Conference on Data Engineering, pp. 25–36
138. Silvestre, P.F., Fragkoulis, M., Spinellis, D., Katsifodimos, A.: Clonos: consistent causal recovery for highly-available streaming dataflows. In SIGMOD '21: International Conference on Management of Data, pp. 1637–1650
139. Singh, G., Chelini, L., Corda, S., Javed Awan, A., Stuijk, S., Jordans, R., Corporaal, H., Boonstra, A.-J.: A review of near-memory

computing architectures: opportunities and challenges. In 2018 21st Euromicro Conference on Digital System Design (DSD), pp. 608–617, Prague, August (2018). IEEE
140. Spenger, J., Carbone, P., Haller, P.: Portals: an extension of dataflow streaming for stateful serverless. In Proceedings of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, pp. 153–171, (2022)
141. Spenger, J., Huang, C., Haller, P., Carbone, P.: Portals: a showcase of multi-dataflow stateful serverless. Proc. VLDB Endow. **16**(12), 4054–4057 (2023)
142. Srivastava, U., Widom, J.: Flexible time management in data stream systems. In Proceedings of the Twenty-third ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, pp. 263–274
143. Su, L., Zhou, Y.: Tolerating correlated failures in massively parallel stream processing engines. In 32nd IEEE International Conference on Data Engineering, ICDE 2016, pp. 517–528
144. Tatbul, N., Çetintemel, U., Zdonik, S., Cherniack, M., Stonebraker, M.: Load shedding in a data stream manager. In VLDB, (2003)
145. Tatbul, N., Çetintemel, U., Zdonik, S. B.: Staying FIT: efficient load shedding techniques for distributed stream processing. In Proceedings of the 33rd International Conference on Very Large Data Bases 2007, pp. 159–170
146. Tatbul, N., Zdonik, S.B.: Window-aware load shedding for aggregation queries over data streams. In Proceedings of the 32nd International Conference on Very Large Data Bases 2006, pp. 799–810
147. Tatbul, N., Zdonik, S.B., 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)
148. Terry, D.B., Goldberg, D., Nichols, D.A., Oki, B.M.: Continuous queries over append-only databases. In Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data, pp. 321–330
149. Thomas, J., Hanrahan, P., Zaharia, M.: Fleet: a framework for massively parallel streaming on FPGAs. In ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, pp. 639–651
150. To, Q.-C., Soto, J., Markl, V.: A survey of state management in big data processing systems. VLDBJ **27**(6), 847–872 (2018)
151. Tolosana-Calasanz, R., Montes, J.D., Rana, O.F., Parashar, M.: Feedback-control and queueing theory-based resource management for streaming applications. IEEE Trans. Parallel Distrib. Syst. **28**, 1061–1075 (2017)
152. 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 SIGMOD, (2014)
153. Traub, J., Grulich, P.M., Cuellar, A.R., Breß, S., Katsifodimos, A., Rabl, T., Markl, V.: Efficient window aggregation with general stream slicing. In Advances in Database Technology - 22nd International Conference on Extending Database Technology, EDBT 2019, pp. 97–108
154. Tu, Y., Liu, S., Prabhakar, S., Yao, B.: Load shedding in stream databases: a control-based approach. In Proceedings of the 32nd International Conference on Very Large Data Bases 2006, pp. 787–798
155. Tucker, P.A., Maier, D., Sheard, T., Fegaras, L.: Exploiting punctuation semantics in continuous data streams. IEEE Trans. Knowl. Data Eng. **15**(3), 555–568 (2003)
156. Urhan, T., Franklin, M.J.: Dynamic pipeline scheduling for improving interactive query performance. In VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, pp. 501–510
157. Urhan, T., Franklin, M.J.: Xjoin: a reactively-scheduled pipelined join operator. *IEEE Data Eng. Bull.*, **23**, (2000)
158. Venkataraman, S., Panda, A., Ousterhout, K., Armbrust, M., Ghodsi, A., Franklin, M.J., Recht, B., Stoica, I.: Drizzle: Fast and adaptable stream processing at scale. In Proceedings of the 26th Symposium on Operating Systems Principles 2017, pp. 374–389
159. Verheijde, J., Karakoidas, V., Fragkoulis, M., Katsifodimos, A.: S-QUERY: opening the black box of internal stream processor state. In 38th IEEE International Conference on Data Engineering, ICDE 2022, pp. 1314–1327
160. Wu, Y., Tan, K.-L.: ChronoStream: Elastic stateful stream computation in the cloud. In ICDE, (2015)
161. Xu, L., Peng, B., Gupta, I.: Stela: enabling stream processing systems to scale-in and scale-out on-demand. In 2016 IEEE International Conference on Cloud Engineering, IC2E 2016, pp. 22–31
162. Xu, L., Venkataraman, S., Gupta, I., Mai, L., Potharaju, R.: Move fast and meet deadlines: fine-grained real-time stream processing with cameo. In 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21), pp. 389–405, (2021)
163. Zaharia, M., Das, T., Li, H., Hunter, T., Shenker, S., Stoica, I.: Discretized streams: fault-tolerant streaming computation at scale. In ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP'13, pp. 423–438
164. Zaharia, M., Das, T., Li, H., Shenker, S., Stoica, I.: Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In USENIX HotCloud, (2012)
165. Zhang, F., Yang, L., Zhang, S., He, B., Lu, W., Du, X.: Finestream: Fine-grained window-based stream processing on cpu-gpu integrated architectures. In: USENIX Annual Technical Conference. USENIX ATC, pp. 633–647 (2020)
166. Zhang, S., He, J., Zhou, A.C., He, B.: Briskstream: scaling data stream processing on shared-memory multicore architectures. In Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, pp. 705–722
167. Zhang, S., Zhang, F., Wu, Y., He, B., Johns, P.: Hardware-conscious stream processing: a survey. SIGMOD Rec. **48**(4), 18–29 (2019)
168. Zhang, Z., Gu, Y., Ye, F., Yang, H., Kim, M., Lei, H., Liu, Z.: A hybrid approach to high availability in stream processing systems. In 2010 International Conference on Distributed Computing Systems, ICDCS 2010, pp. 138–148
169. Zhu, X., Serafini, M., Ma, X., Aboulnaga, A., Chen, W., Feng, G.: LiveGraph: a transactional graph storage system with purely sequential adjacency list scans. Proc. VLDB Endow. **13**(7), 1020–1034 (2020)