A Survey on the Evolution of Stream Processing Systems

Marios Fragkoulis[*] · **Paris Carbone**[†] · **Vasiliki Kalavri**[‡] · **Asterios Katsifodimos**[*]

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 early ('00-'10) and modern ('11-'18) streaming systems, and discuss recent trends and open problems.

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.). For instance, web companies are using stream processing for dynamic car-trip pricing, banks apply it for credit card fraud detection, while traditional industries apply streaming technology for real-time harvesting analytics. At the moment of writing we are witnessing a trend towards using stream processors to build more general event-driven architectures [87], large-scale continuous ETL and analytics, and microservices [83].

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 [127], and was followed by lots of research on stream processing in the early 00s. Fundamental concepts and ideas originated in the database community and were implemented in prototypical systems such as TelegraphCQ [45], Stanford's STREAM, NiagaraCQ [47], Auroral/Borealis [9], and Gigascope [50]. Although these prototypes roughly agreed on the data model, they differed considerably on querying semantics [18, 30]. This research period also introduced various systems challenges, such as sliding window aggregation [19, 95], fault-tolerance and high-availability [26, 120], as well as load balancing and shedding [124]. This first wave of research was highly influential to 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 [54] and the popularization of Cloud Computing. The focus shifted towards distributed, data-parallel processing engines and shared-nothing architectures on commodity hardware. Lacking well-defined semantics and a proper query language, systems like Millwheel [12], Storm [2], Spark Streaming [141], and Apache Flink [34] first exposed primitives for expressing streaming computations as hard-coded dataflow graphs and transparently handled data-parallel execution on distributed clusters. With very high influence, the Google Dataflow model [13] re-introduced older ideas such as out-of-order processing [96] and punctuations [134], 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.

[*] {m.fragkoulis,a.katsifodimos}@tudelft.nl, Delft University of Technology
[†] paris.carbone@ri.se - RISE, parisc@kth.se - KTH EECS
[‡] vkalavri@bu.edu, Boston University

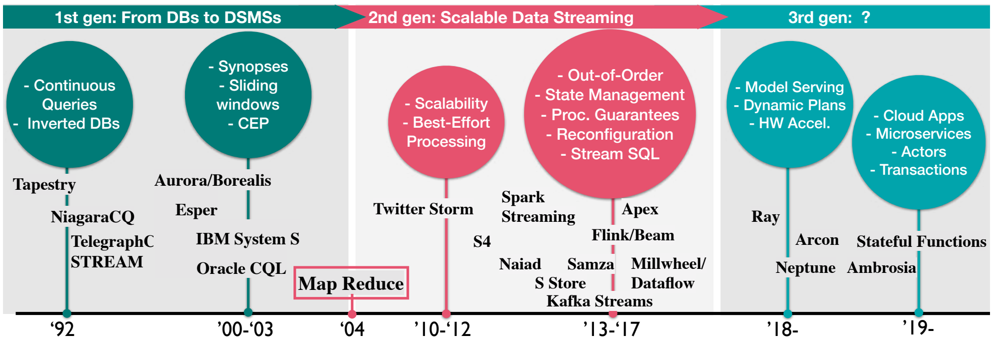


Fig. 1: An overview of the evolution of stream processing and respective domains of focus.

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 yet another paradigm shift towards general event-driven architectures, actor-like programming models and Microservices [11, 31], and a growing use of modern hardware [88, 128, 142, 144].

This survey is the first to focus on the evolution of streaming systems rather than 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

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 important but overlooked 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.

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 or those who seek a comparison between streaming technology and advances from adjacent research communities.

Cugola and Margara [51] 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 [72], 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 [117] 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. [73] 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 [129] survey the concept of state and its applications in big data management systems, covering also aspects of streaming state. Finally, Dayarathna and Perera [53] 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.

Theoretical foundations of streaming data management and streaming algorithms are out of the scope of this survey. A comprehensive collection of influential works on these topics can be found in Garofalakis et al. [61]. The 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 Section 2. Then we elaborate on each of the important functionalities offered by stream processing systems: out-of-order data management (Section 3), state management (Section 4), fault tolerance and high availability (Section 5), and load management, elasticity, and reconfiguration (Section 6). Each one of these sections contains a *Vintage vs. Modern* discussion that compares early to contemporary approaches and a summary of open problems. We summarize our major findings, discuss prospects, and conclude in Section 7.

2 Preliminaries

In this section, we provide necessary background and explain fundamental stream processing concepts the rest of this survey relies on. 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 [61]. Data streams are high-volume, 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 either bear a timestamp or are assigned one on arrival.

Respectively, 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 assumption that state can be reconstructed from associated inputs. To achieve acceptable performance, streaming operators need to leverage incremental computation.

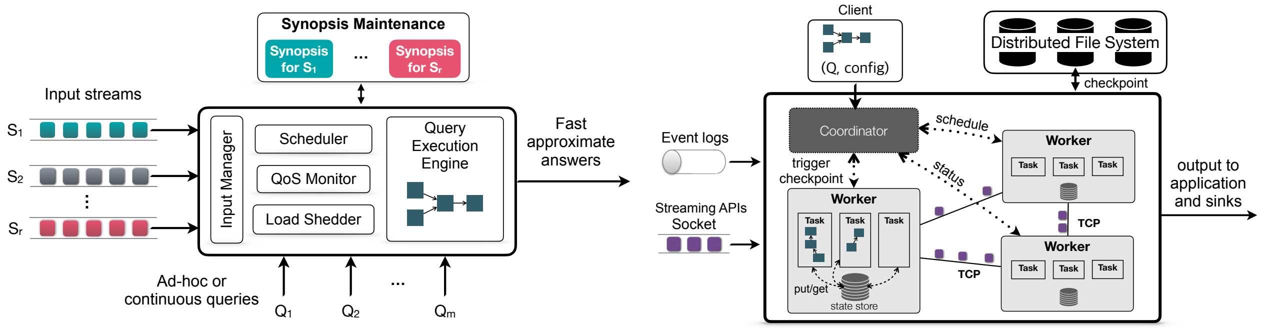
The aforementioned characteristics of data streams and continuous queries provide a set of unique requirements for streaming systems, other than the evident performance ones of low latency and high throughput. Given the lack of control over the input order, a streaming system needs to produce correct results when receiving out-of-order and delayed data (cf. Section 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. Section 4) and guard it against failures (cf. Section 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. Section 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 [61]. 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, a stream is interpreted as describing a changing relation over a common schema. *Base* streams are produced by external sources and update relation tables, while *derived* streams are produced by continuous queries and update materialized views. An operator out-



(a) Data stream management system

(b) Distributed dataflow system

Fig. 2: Architectures of early and modern streaming systems.

puts event streams that describe the changing view computed over the input stream according to the relational semantics of the operator.

STREAM [17] defines streams as bags of tuple-timestamp pairs and relations as time-varying bags of tuples. The implementation unifies both types as sequences of timestamped tuples, where each tuple also carries a flag that denotes whether it is an insertion or a deletion. Input streams consist of insertions only, while relations may also contain deletions. TelegraphCQ [45] uses a similar data model. Aurora [9] models streams as append-only sequences of tuples, where a set of attributes denote the key and the rest of the attributes denote values. Borealis [8] generalizes this model to support insertion, deletion, and replacement messages. Messages may also contain additional fields related to QoS metrics. Gigascope [50] extends the sequence database model. It assumes that stream elements bear one or more timestamps or sequence numbers, which generally increase (or decrease) with the ordinal position of a tuple in a stream. Ordering attributes can be (strictly) monotonically increasing or decreasing, monotone non-repeating, or increasing within a group of records. In CEDR [27], stream elements bear a valid timestamp, V_s, after which they are considered valid and can contribute to the result. Alternatively, events can have validity intervals. The contents of the relation at time t are all events with $V_s \leq t$.

2.2.2 *Dataflow streaming model*

The dataflow streaming model, as implemented by systems of the second generation [13, 34, 141], does not impose any strict schema or semantics to the input stream elements, other than the presence of a timestamp. While some systems, like Naiad [108], require that all stream elements bear a logical timestamp, other systems, such as Flink [34] and Dataflow [13], 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) *processing* time is the time when events are processed in the streaming system, and (iii) *ingestion* time is the time when events arrive at the 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, such as windows.

2.3 Architectures of streaming systems

The general architecture of streaming systems has evolved significantly over the last two decades. Before we delve into the specific approaches to out-of-order management, state, fault tolerance, and load management, we outline some fundamental differences between early and modern streaming systems. Figure 2a shows a typical data stream management system (DSMS) architecture next to a modern dataflow streaming system in Figure 2b.

The architecture of a DSMS follows closely that of a database management systems (DBMS), with the addition of certain components designated to address the requirements of streaming data (cf. Section 2.1). In particular, the input manager is responsible for ingesting streams and possibly buffering and ordering input elements. The scheduler determines the order or operator execution, as well as the number of tuples to process and push to the outputs. Two important additional components are the quality monitor and load shedder which monitor stream input rates and query performance and selectively drop input records to meet target latency requirements. Queries are compiled into a shared query plan which is optimized and submitted to the query execution engine. In the common case, a DSMS supports both ad-hoc and continuous queries. Early architectures are designed with the goal to provide fast, but possibly approximate results to queries.

The next generation distributed dataflow systems are usually deployed on shared-nothing clusters of machines. Dataflow systems employ task and data parallelism, have explicit state management support, and implement advanced fault-tolerance capabilities to provide result guarantees. Distributed workers execute parallel instances of one of more operators (tasks) on disjoint stream partitions. In contrast to DSMSs, queries are independent of each other, maintain their own state, and they are assigned dedicated resources. Every query is configured individually and submitted for execution as a separate job. Input sources are typically assumed to be replayble and state is persisted to embedded or external stores. Modern architectures prioritize high throughput, robustness, and result correctness over low latency.

Despite the evident differences between early and modern streaming systems' architectures, many fundamental aspects have remained unchanged in the past two decades. The following sections examine in detail how streaming systems have evolved in terms of out-of-order processing, state capabilities, fault-tolerance, and load management.

3 Out-of-order data management

A streaming system receives data continuously from one or more input sources. Typically the order of data in a stream is part of the stream's semantics [100]. Depending on the computations to perform, a streaming system may have to process stream tuples in a certain order to provide semantically correct results [121]. However, in the general case, a stream's data tuples arrive out of order [93, 134] for reasons explained in Section 3.1.

Out-of-order data tuples [121, 132] arrive in a streaming system after tuples with later event time timestamps.

In the rest of the paper we use the terms disorder [100] and out-of-order [12, 96] to refer to the disturbance of order in a stream's data tuples. Reasoning about order and managing disorder are fundamental considerations for the operation of streaming systems.

In the following, we highlight the causes of disorder in Section 3.1, clarify the relationship between disorder in a stream's tuples and processing progress in Section 3.2, and outline the two key system architectures for managing out-of-order data in Section 3.3. Then, we describe the consequences of disorder in Section 3.4 and present the mechanisms for managing disorder in Section 3.5. Finally, in Section 3.6, we discuss the differences of out-of-order data management in early and modern systems and we present open problems in Section 3.7.

3.1 Causes of disorder

Disorder in data streams may be owed 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 [89, 121]. Depending on the network's reliability, bandwidth, and load, the routing of some stream tuples can take longer to complete compared to the routing of others, leading to a different arrival order in a streaming system. Even if the order of tuples in an individual stream is preserved, ingestion from multiple sources, such as sensors, typically results in a disordered collection of tuples, unless the sources are carefully coordinated, which is rare.

External factors aside, specific operations on streams break tuple 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 [135] and outputs join results by order of match [68, 82]. Second, windowing based on an attribute different to the ordering attribute reorders the stream [50]. Third, data prioritization [115, 136] 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 tuples of the two input streams interleaving each other in random order [9].

3.2 Disorder and processing progress

In order to manage disorder, streaming systems need to detect processing progress. We discuss how disorder management and progress tracking are intertwined in Sections 3.3 and 3.4.

Progress regards how much the processing of a stream's tuples has advanced over time. Processing progress can be defined and quantified with the aid of an attribute A of a stream's tuples that orders the stream. The processing of the stream progresses when the smallest value of A among the unprocessed tuples increases over time [96]. A then is a *progressing attribute* and the oldest value of A per se, is a measure of progress because it denotes how far in processing tuples the system has reached since the beginning. Beyond this definition, streaming systems often make their own interpretation of progress, which may involve more than one attributes.

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 [9, 18, 50, 121], and (ii) out-of-order processing systems [12, 34, 96, 108].

In-order processing systems manage disorder by fixing a stream's order. As a result, they essentially track progress by monitoring how far the processing of a data stream has advanced. In-order systems buffer and reorder tuples up to a *lateness* bound. Then, they forward the reordered tuples for processing and clear the corresponding buffers.

In out-of-order processing systems, operators or a global authority produce progress information using any of the metrics detailed in Section 3.5, and propagate it to the dataflow graph. The information typically reflects the oldest unprocessed tuple in the system and establishes a lateness bound for admitting out-of-order tuples. In contrast to in-order systems, tuples are processed without delay in their arrival order, as long as they do not exceed the lateness bound.

3.4 Effects of disorder

In unbounded data processing, disorder can impede progress [96] or lead to wrong results if ignored [121].

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 [9, 121]. When operators in in-order systems receive out-of-order tuples, they have to 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. To include late tuples in results, they additionally need to store the processing state up to the lateness bound. As a side-note, order-insensitive operators [9, 96, 121], 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 might lead to incorrect results if the output is computed on partial input only. Thus, a streaming system needs to be capable of processing out-of-order data and incorporate their effect to the computation. However, without knowledge of how late data can be, waiting indefinitely can block output and accumulate large computation state. This concern manifests on all architectures and we discuss how it can be countered with disorder management mechanisms, next.

3.5 Mechanisms for managing disorder

In this section, we elaborate on influential mechanisms for managing disorder in unbounded data, namely slack [9], heartbeats [121], low-watermarks [96], pointstamps [108], and triggers [13]. Heartbeats, low-watermarks, and pointstamps track processing progress and quantify a lateness bound using a metric, such as time. In contrast, slack merely quantifies the lateness bound. If tuples arrive after the lateness bound expires, triggers can be used to update computation results in *revision processing* [8]. We also discuss punctuations [134], a generic mechanism for communicating information across the dataflow graph, that has been heavily used as a vehicle in managing disorder.

Tracking processing progress. *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 tuples intervening between the actual occurrence of an out-of-order tuple 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 tuples.

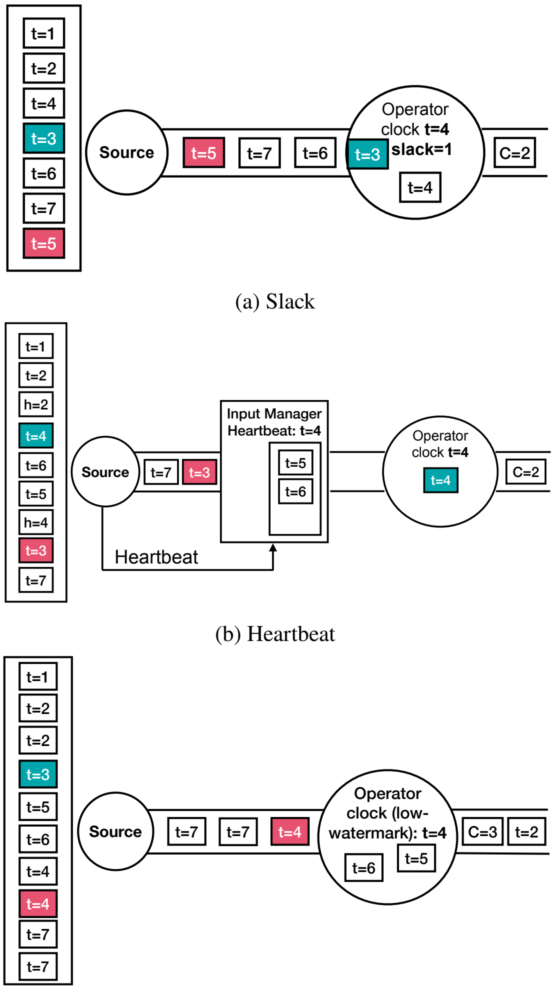
A *heartbeat* is a slack alternative that consists of an external signal carrying progress information about a data stream. It contains a timestamp indicating that all succeeding stream tuples will have a timestamp larger than the heartbeat's timestamp. Heartbeats can either be generated by an input 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 [121].

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 tuples will probabilistically bear a higher value than the current low-watermark for the same attribute. Often, A is a tuple'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.

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 to heartbeats, which is an internal mechanism of a streaming system hidden from users, slack is part of the query specification provided by users [9].

Heartbeats and low-watermarks are similar in terms of progress-tracking logic. However, two important differences set them apart. While heartbeats expose the progress of stream tuple generation at the input sources, the low-watermark extends this to the processing progress of computations within the streaming system by reflecting their oldest pending work. Second, the low-watermark generalizes the concept of the oldest value, which signifies the current progress point, to any progressing attribute of a stream tuple besides timestamps.

In contrast to heartbeats and slack, *punctuations* are metadata annotations embedded in data streams. A punctua-



(a) Slack

(b) Heartbeat

(c) Low-watermark

Fig. 3: Mechanisms for managing disorder.

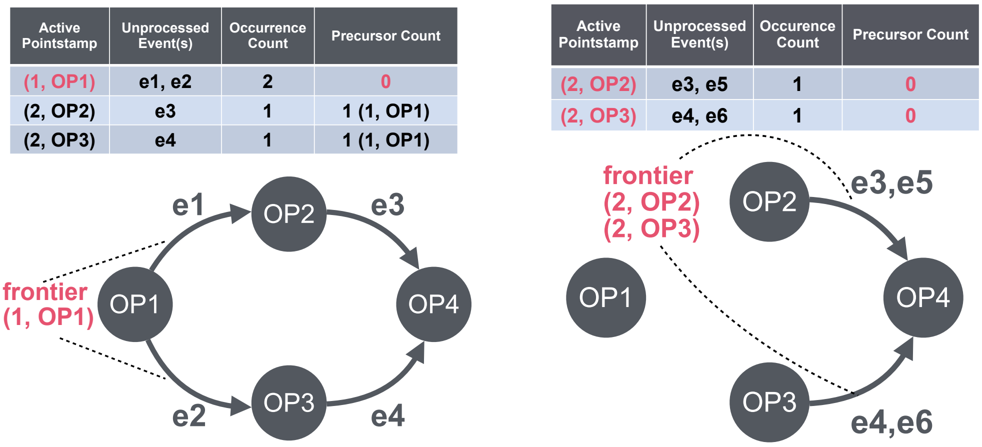
tion is itself a stream tuple, which consists of a set of patterns each identifying an attribute of a stream data tuple. A punctuation is a generic mechanism that communicates information across the dataflow graph. Regarding progress tracking, it provides a channel for communicating progress information such as a tuple attribute's low-watermark produced by an operator [96], event time skew [121], or slack [9]. Thus, punctuations can convey which data cease to appear in an input stream; for instance the data tuples with smaller timestamp 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.

Figure 3 showcases the differences between slack, heartbeats, and low-watermarks. The figure depicts a simple aggregation operator that counts tuples in 4-second event time tumbling windows. 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 tuples containing only a timestamp from t=1 to t=7. The timestamp signifies the event time in seconds that the tuple was produced in the input source. Each tuple contains a different timestamp and all tuples are dispatched from a source in ascending order of timestamp. Due to network latency, the tuples may arrive to the streaming system out of order.

Figure 3a presents the slack mechanism. In order to accommodate out-of-order tuples the operator admits out-of-order tuples up to *slack=1*. Thus, the operator having admitted tuples with t=1 and t=2 not depicted in the figure will receive tuple with t=4. The timestamp of the tuple coincides with the max timestamp of the first window for interval [0, 4). Normally, this tuple 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 tuple. The next tuple t=3 belongs to the first window and is included there. At this point, slack also expires and this event finally triggers the window computation, which outputs C=3 for t=[1, 2, 3]. On the contrary, the operator will not accept t=5 at the tail of input because it arrives two tuples after its natural order and is not covered by the slack value.

Figure 3b depicts the heartbeat mechanism. An input manager buffers and orders the incoming tuples by timestamp. The number of tuples 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 tuples with timestamp less or equal to the timestamp of the heartbeat in ascending order. For instance, when the heartbeat with timestamp t=2 arrives in the input manager (not shown in the figure), the input manager dispatches the tuples with timestamp t=1 and t=2 to the operator. The input manager then receives tuples with t=4, t=6, and t=5 in this order and puts them in the right order. When the heartbeat with timestamp t=4 arrives, the input manager dispatches the tuple with timestamp t=4 to the operator. This tuple triggers the computation of the first window for interval [0, 4). The operator outputs C=2 counting two tuples with t=[1, 2] not depicted in the figure. The input manager ignores the incoming tuple with timestamp t=3 as it is older than the latest heartbeat with timestamp t=4.

Figure 3c 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 tuples with t=1 and t=2, the corresponding low-watermark for t=2 (which is propagated downstream), and tuple t=3, the operator receives tuple t=5. Since this tuple 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



Active Pointstamp	Unprocessed Event(s)	Occurrence Count	Precursor Count
(1, OP1)	e1, e2	2	0
(2, OP2)	e3	1	1 (1, OP1)
(2, OP3)	e4	1	1 (1, OP1)

(a) Pointstamps and frontier

Active Pointstamp	Unprocessed Event(s)	Occurence Count	Precursor Count
(2, OP2)	e3, e5	1	0
(2, OP3)	e4, e6	1	0

(b) Frontier moves forward

Fig. 4: High-level workflow of pointstamps and frontier

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 tuples with t=[5, 6] to the second window with interval [4, 8). The operator will not admit tuple t=4 because it is not greater (more recent) than the current low-watermark value t=4.

Like punctuations, *pointstamps* are embedded in data streams, but a pointstamp is attached to each stream data tuple as opposed to a punctuation, which forms a separate tuple. Pointstamps are pairs of timestamp and location that position data tuples on a vertex or edge of the dataflow graph at a specific point in time. An unprocessed tuple *p* at a specific location *could-result-in* another unprocessed tuple *p'* with timestamp *t'* at another location when *p* can arrive at *p'* before or at timestamp *t'*. Unprocessed tuples p with timestamp t are in the frontier of processing progress when no other unprocessed tuples could-result-in p. Thus, tuples bearing *t* or an earlier timestamp are processed and the frontier moves on. The system enforces that future tuples will bear a greater timestamp than the tuples that generated them. This modeling of processing progress traces the course of data tuples on the dataflow graph with timestamps and tracks the dependencies between unprocessed events in order to compute the current frontier. The concept of a frontier is similar to a low-water mark.

The example shown in Figure 4 showcases how pointstamps and frontiers work. The example in Figure 4a includes three active pointstamps. Poinstamps are active when they correspond to one or more unprocessed events. Pointstamp (1, OP1) is in the frontier of active pointstamps, because its precursor count is 0. The precursor count, specifies the number of active pointstamps that could-result-in that pointstamp. In the frontier, notifications for unprocessed events can be delivered. Thus, 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 data flow graph, it is easy to see that pointstamp (1, OP1) could-result-in pointstamps (2, OP2) and (2, OP3). Therefore, the precursor count of the latter two pointstamps is 1. A bit later as Figure 4b depicts, after 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 count of pointstamps (2, OP2) and (2, OP3) is 0, then the frontier moves on to these active pointstamps. Consequently, all four event notifications can be delivered. Obsolete pointstamps (1, OP1), (2, OP2), and (2, OP3), are removed from their location, since they correspond to no unprocessed events. Although this example is made simple for educational purposes, the progress tracking mechanism, has the power to track the progress of arbitrary iterative and nested computations.

Pointstamps/frontiers track processing progress regardless of the notion of event time. However, it is possible for users to capture out-of-order data with pointstamps/frontiers by establishing a two-dimensional frontier of event time and processing time that is flexibly open on the side of event time.

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 deadlock is inevitable.

Chandramouli et al. [43] 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' timestamp. While the punctuation flows in the loop the operator observes the stream's tuples to validate its guess. When this happens and the speculative punctuation re-enters the operator, it becomes a regular punctuation that carries progress information downstream. Then a new speculative punctuation is generated and is fed in the loop. By combining a dedicated operator, speculative output, and punctuations this work achieves to track progress and tolerate disorder in cyclic streaming queries. The approach works for strongly convergent queries and can be utilized in systems that provide speculative output.

In Naiad [108, 109], the general progress-tracking model features logical 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. Like in Chandramouli et al. [43], 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 timestamp 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 is external to the dataflow. This design defies the associated implementation complexity in favor of a) efficient delivery of notifications that is proportional to dataflow nodes instead of edges and b) incremental computation that avoids redundant work. Although not directly incorporated, the notion of event time can be encapsulated in multidimensional timestamps to account for out-of-order data.

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 [8]. From there on, it has been combined with in-order processing architectures [42, 110], as well as out-of-order processing architectures [13, 14, 27, 89]. In some approaches revision processing works by *storing* incoming data and *revising* computations in face of late, updated, or retracted data [13, 14, 27]. Other approaches *replay* affected data, *revise* computations, and propagate the revision messages to update all affected results until the present [8, 110, 118]. Finally, a third line of approaches maintain multiple *partitions* that capture events with different levels of lateness and *consolidate* partial results [42, 89].

Store and revise. Microsoft's CEDR [27] and StreamInsight [14], and Google's Dataflow [13] 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 [13] 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 overwrite older ones, discarding where new results complement 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 [14].

Replay and revise. **Dynamic revision** [8] and **speculative processing** [110] replay an affected past data subset when a revision tuple is received. An optimization of this scheme relies on two revision processing mechanisms, upstream processing and downstream processing [118]. Both are based on a special-purpose operator, called *connection point*, that intervenes between two regular operators and stores tuples output by the upstream operator. According to the upstream revision processing, an operator downstream from a connection point can ask for a set of tuples 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 tuples related to a received revision tuple. Under circumstances, the operator can calculate correct revisions by incorporating the net effect of the difference between the original tuple 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. It keeps the input message history to 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 can not 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 snapshot recovery if no output has been produced for a disordered input stream. Otherwise, it retracts all produced output in a recursive manner. In speculative processing because revision processing is opportunistic, no history bound is set.

Partition and consolidate. Both **order-independent processing** [89] and **impatience sort** [42] are based on partial processing of independent partitions in parallel and consolidation of partial results. In order-independent processing, when a tuple is received after its corresponding progress indicator a new partition is opened and a new query plan instance processes this partition using standard out-of-order processing techniques. On the contrary, in impatience sort, the latest episode of the vision of CEDR [27], an online sorting operator 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 users. In order-independent processing, tuples that are too old to be considered in their original partition are included in the partition which has the tuple 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 tuple is received that does not belong to one of the ad-hoc partitions, a new ad-hoc partition is introduced. An out-of order tuple 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.

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 tuples. 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 and vice versa. 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 Vintage vs. Modern

The importance of event order in data stream processing became obvious since its early days [22] leading to the first wave of simple intuitive solutions. Early approaches involved buffering and reordering arriving tuples using some measure for adjusting the frequency and lateness of data dispatched to a streaming system in order [9, 45, 121]. A few years later, the introduction of out-of-order processing [96] improved throughput, latency, and scalability for window operations by keeping track of processing progress without ordering tuples. In the meantime, revision processing [8] was proposed as a strategy for dealing with out-of-order data reactively. In the years to come, in-order, out-of-order, and revision processing were extensively explored, often in combination with one another [13,14,27,89,110]. 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 [12] and the Google Dataflow model [13], Flink [34], and Spark [20]. Table 1 presents how both vintage 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 finger the architecture's implementation and maintainance complexity. In addition, revision processing, which is used to reconcile out-of-order tuples 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.

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* [9, 48]	✓			Slack	User config	Number of tuples	—
STREAMS [121]	✓			Heartbeat	Signal to input manager	Timestamp (event time skew, network latency, out-of-order bound)	—
Borealis [8]		✓	✓	History bound	System config	Number of tuples or time units	Replay past data, enter revised values, issue delta output
Gigascope [78]		✓		Low-watermark	Punctuation	Timestamp	—
Timestream [114]	✓			Low-watermark	Punctuation	Timestamp	—
Millwheel [12]		✓		Low-watermark	Signal to central authority	Timestamp	—
Naiad [108]		✓	✓	Pointstamp	Part of data tuple	Multidimensional timestamp	Incremental processing of updated data via structured loops
Trill [41]	✓			Low-watermark	Punctuation	Timestamp	—
Streamscope [97]	✓			Low-watermark	Punctuation	Timestamp; sequence number	—
Samza [112]		✓	✓	—	—	—	Find, roll back, recompute affected input windows
Flink [34]		✓	✓	Low-watermark	Punctuation	Timestamp	Store & Recompute/Revise
Dataflow [13]		✓	✓	Low-watermark	Signal to central authority	Timestamp	Discard and recompute; accumulate and revise; custom
Spark [20]		✓	✓	Slack	User config	Number of seconds	Discard and recompute; accumulate and revise

4 State Management

State is effectively what captures all internal side-effects of a continuous stream computation, which includes for example active windows, buckets of records, partial or incremental aggregates used in an application as well as possibly some user-defined variables created and updated during the execution of a stream pipeline. A careful look into how state is exposed and managed in stream processing systems exposes an interesting trace of trends in computer systems and cloud computing as well as a revelation of prospects on upcoming capabilities in event-based computing. This section provides an overview of known approaches, modern directions and open problems in the context of state management.

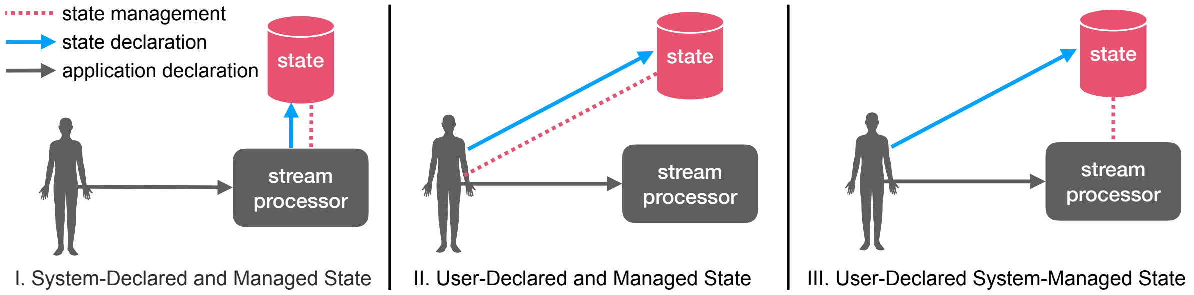
4.1 Topics of Stream State Management

Stream State Management is an active system subject that incorporates different methodologies regarding how state should be declared in a stream application, as well as how it should be scaled and partitioned. Furthermore, it incorporates different methods to make state persistent for infinitely long running applications and defines system guarantees and properties to maintain whenever a change in the system occurs. A system change implies reconfiguration and is the result of a partial process or network failure, or actions that need to be taken to adjust compute and storage capacity. Most of these issues have been introduced in part within the context of pioneering DSMSs such as Aurora and Borealis [38]. The latter system, has set the foundations in formulating many of these problems such as the need for embedded state, persistent store access as well as failure recovery protocols. In Table 2 we categorize known data stream processing systems according to their respective state management approaches, including programmability, scalability and consistency characteristics. The rest of this section offers an overview of each of the topics in stream state management along with past and currently employed approaches, all of which we categorize as follows:

- **Programmability:** State in a programming model can be either implicitly or explicitly declared and used. Different system trends have influenced both how state can been exposed in a data stream programming model as well as how it should be scoped and managed. Section 4.2 discusses different approaches and their trade offs.
- **Scalability and Persistency:** 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. In data stream processing the idea of employing internal and external persistence strategies was uniformly embraced in more recent generations of systems. Section 4.3 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.
- **Consistency:** One of the most foundational transitioning steps in stream technology has been the development and adoption of transactional-level guarantees. Section 4.4 gives an overview of the state of the art and covers the semantics of transactions in data streaming alongside implementation methodologies.

Table 2: State Management Features in Data Streaming Systems

System	Declaration		Management		State Management Architecture				Transactional-Level		
	User	System	User	System	Ephemeral	Embedded	External	Embedded Compute	Action	Epoch	-
Aurora/Borealis [8, 38]		✓		✓	✓						✓
STREAM [16]		✓		✓	✓						✓
TelegraphCQ [45]		✓		✓	✓						✓
S4 [111]	✓		✓		✓						✓
Storm (1.0) [2]	✓		✓		✓						✓
Spark(1.0) [141]	✓			✓		✓				✓	
Trident [3]	✓			✓		✓				✓	
SEEP [107]	✓			✓	✓				✓		
Naiad [108]	✓			✓	✓					✓	
TimeStream [114]	✓			✓							
Millwheel [12]	✓			✓			✓		✓		
Flink [33, 34]	✓			✓		✓				✓	
Kafka-Streams [6]	✓			✓		✓		✓		✓	
Samza [112]	✓			✓		✓				✓	
Streamscope [97]		✓		✓	✓					✓	
S-Store [104]		✓		✓				✓		✓	



I. System-Declared and Managed State
II. User-Declared and Managed State
III. User-Declared System-Managed State

Fig. 5: State Programmability and Management Approaches

4.2 Programmability of State

There are three actors involved in stateful stream processing: the user, the stream processor, and the actual state. In this context, we can observe differences across systems on how the user and system interact with state in a long running stream application. There are two key responsibilities. First, one has to declare and use the state in a stream application but there is also the need for someone to be responsible for managing the state. For both of these, the responsible entity can be either the user or the stream processor. The rest of this section focuses on the three main configurations, which are also depicted in Figure 5 and described below according to adoption order, in the course of the evolution of stream processing.

System-Declared and Managed State. In the early days of data stream management when main memory was scarce, state had a facilitating role, supporting the implementation of user-defined operators, such as CQL's join filter and sort algorithms, as employed in STREAM [16]. 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 [38], attached special synopses to a stream application's dataflow graph supporting different operators, such as a window max, 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. Although, 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.

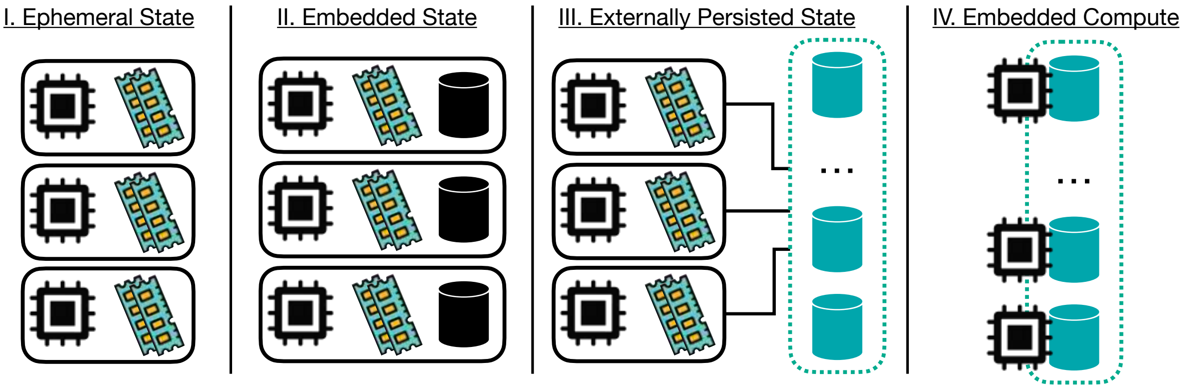


Fig. 6: Scalable Architectures for Stateful Data Streaming

User-Declared and Managed State. At the brink of the second generation of scalable stream processing systems, in the post-MapReduce era, there was a primary focus in compute scalability with systems like Storm [2] allowing the composition of distributed pipelines of tasks. For application flexibility and simplicity, many of these systems did not provide any state management whatsoever, leaving everything regarding state to the hands of the programmer. That included both declaration and management of state. 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, 94]). In summary, application-managed state offers flexibility and gives expert users implementation freedom. However, no state management capabilities are offered from the system's side. As a result, the user has to reason about scalability, processing guarantees, and all necessary third-party storage system dependencies. These are all complex choices to make and require a combination of deep expertise and additional engineering work to integrate stream and storage technologies.

User-Declared System-Managed State. Currently, most stream processing systems allow a level of freedom for user-defined state through a form of a stateful processing API. This enriches stream applications to define their 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 types used, serializers/deserializers and 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 Scalability and Persistence

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 need to facilitate unbounded data stream executions where the space complexity for stream state is linear to the over-increasing input consumed by a stream processor at any point in time. 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.

4.3.1 Types of scalable state

Scalable state takes two forms in a stream application, typically referred to as *task-level* and *key-level* state. Depending on the nature of a specific operator, any or both of these state types can be employed.

Task-Level State. Task-level partitioning maps state to physical compute tasks, allowing one instance per task. This is preferred when there is a need to compute global aggregates, such as top-K stream queries, or when the state does not grow over the course of time in an unbounded execution. Task-level state can also be useful for keeping offset counts in a log consumed by a physical stream source task. It is, however, not the norm in most stream applications, since in most use cases, state needs to scale in a data-parallel manner. Furthermore, task-level state is hard to re-partition, given that it always maps to a physical set of tasks.

Key-Level State. Key-level state is the de facto way to define scalable state in modern streaming systems. Keyed state allows logical-level partitioning to compute tasks, where each task handles a specific range of keys. This is enabled in the API level through an additional operation that is invoked prior to stateful processing which lifts the scope from task- to key-based processing such as “keyBy” in Apache Flink or “groupBy” in Beam and Kafka-Streams.

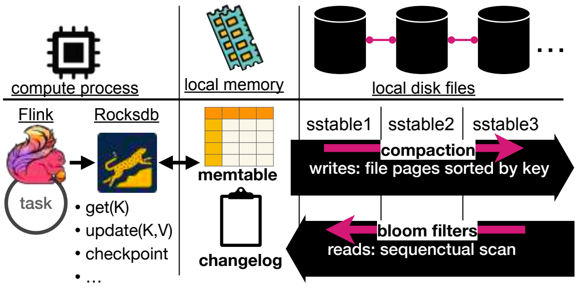


Fig. 7: Embedded State Example (Flink and Rocksdb-LSM)

4.3.2 *Scalable Architectures for Data Streaming*

While data stream ingestion can be unbound, the same does not apply to the number of states that can be kept and maintained by tasks. Relying on limited and ephemeral main memory is not practical, especially when dealing with state growing proportionally to the number of distinct keys in an unbounded stream. In Figure 6, we enumerate four system architectures that have been used to support scalable, "out-of-core" state: I. *ephemeral* (memory-only) state, II. state *locally embedded* to disk, III. *externally persisted* state, and IV. *embedded stream compute*, as feature on top of scalable storage architectures.

Ephemeral State. The ephemeral state is not always an active state management choice. It typically refers to the absence of any form of persistence or the employment of failure recovery and management techniques which have complete reliance on transient state. Systems that belong to the first case are typically first generation of stream processing systems such as Aurora and STREAM, and scalable stream processors without state management capabilities, such as Storm and S4 [111]. Several state management approaches also build completely on transient in-memory state. Those include stream replication/recovery techniques [120] and complete systems, such as SEEP [37].

Embedded State. The embedded state approach is a popular choice among modern data streaming systems. That is mainly due to the fact that local access yields fast reads and reasonably fast writes. On the other hand, since state is coupled to compute tasks, it is more challenging to reconfigure stream processors using this approach, since coordination and data shuffling is needed beforehand.
Example: Figure 7 depicts an example of embedded state with Apache Flink and RocksDB [5] as a state backend. Flink is managing compute tasks and effectively forwarding every read, write or local checkpoint operation to a local RocksDB instance. RocksDB maintains an in-memory table with recent changes in state, so when a write or read arrives it is applied there first. Memtables in RocksDB are periodically flushed to disk in key order according to time or size threshold policies or forcefully during a checkpoint. According to the Log-Structure-Merge (LSM) architecture [113], the on-disk file structures form hierarchical dependencies. Read operations that do not find an entry in the Memtable initiate a scan through the committed files of the LSM tree (SSTables). To avoid lengthy sequential file scans known optimisations include file compaction for space reduction and bloom filtering to skip files with now key matches.

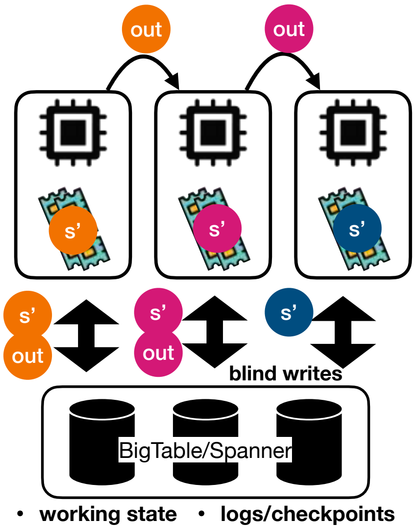


Fig. 8: Externally Persisted State in Google Millwheel

External State. Externally-managed state approaches split state management responsibilities between the stream processor and an external data storage system. This allows for more modular system designs (decoupling) and effective reuse of the properties of other existing systems (e.g., transactions, consistency guarantees, auto-scaling) to support more complex guarantees in the context of data streaming.
Example: When it comes to external state management, Google's Millwheel that serves as the executor of the Google Dataflow service, is a representative example. Millwheel builds on the capabilities of BigTable [46] and Spanner [49] (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 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.

Embedded Compute. Database systems are significantly more mature technology compared to data stream processors. Embedded computing approaches exploit performance characteristics and transactional capabilities of scalable databases to implement and support stateful data streaming on top. This category defers to external state approaches since compute and storage in this case are coupled. Further-

more, it defers to embedded state approaches, since the underlying runtime technology is the actual storage system. Embedded compute is a design direction which depends purely on the capabilities of the storage system that data streaming is implemented on. On one hand, these approaches share the benefits and optimizations of the underlying storage system while also lacking the event-processing performance capabilities of dedicated stream processing systems.
Example: Among embedded computing approaches, S-Store [104] is one of the most representative ideas. S-Store breaks down an unbounded stream computation into a series of transactions that are statically scheduled in H-Store [81], a DBMS provided as an extension of the database. A key characteristic of S-Store is the use of H-Store's ACID transaction properties to also support transactional stream processing as discussed further in subsection 4.4. Kafka Streams [4] is another distinct example of a system that combines embedded compute with embedded state. Stream tasks attach to physical Kafka brokers (physical partitioned logging nodes) and an embedded database instance is allocated per task to support dynamic state.

4.4 Transactional Guarantees and Consistency

Consistent stream processing has for long been an open research issue due to the challenging nature of distributed unbounded processing but also due to the lack of a formal specification of the problem itself. Consistency relates to guarantees a system can make at the face of failure as well as any need for change during its operation. In data streaming, changing or updating a running data stream application is a concept also known as reconfiguration. For example, this includes the case when one needs to apply a software update to a stream application or scale out to more compute nodes without loss of accuracy or computation. The underlying relation between fault tolerance and reconfiguration has been highlighted by several works in the past such as the research behind the SEEP system [37] that considers an integrated approach to scale and recover tasks from failures. Currently, most stream processors are transactional processing systems governed by consistency rules and processing guarantees. This section highlights the types of guarantees offered by different stream processing systems and implementation strategies that materialize them.
Past Challenges and The Lambda Architecture: When large scale computing became mainstream, a design pattern emerged called "lambda architecture" which suggested the separation of systems across different layers according to their specialization and reliability capabilities. Hadoop and transactional databases were reliable in terms of processing guarantees, thus, they could take all critical computation. Whereas, stream processing systems could achieve low latency and scale but they did not offer a clear set of consistency guarantees. For example, in the state-oblivious Storm system the fault-tolerance approach would solely consider which input events have been fully processed or not and which should be replayed on a timeout. Nevertheless, there was no clear picture of what level of consistency can be expected from stream processors. At the same time, databases had formal guarantees. For example, a set of transactions would be processed using ACID guarantees, which includes atomicity across transactions, consistency for the valid states a database can have, isolation in terms of concurrent execution, and durability on what can be recovered after failure. To reason about consistency in the context of data streaming, there had been a need to lay out a set of assumptions (e.g., logged input) and processing granularity for defining a concept related to transactions.

Transactional Data Streaming. 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 and 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". 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 output events created by the sinks. If a specific action is lost or happens twice, then the complete system enters into an erroneous state. Fault tolerance is an integral functional area of streaming systems that significantly impacts their consistency. We analyze the fault tolerance strategies of existing streaming systems in Section 5.1. In addition, due to causal dependencies on state, the order of action execution is also critical. 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.

4.4.1 Action-driven Transactional Streaming

A strict form of transactional 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, as depicted in Figure 8. The act of committing these actions is also called a "strong production" in Millwheel.

Action-driven transactional stream processing is an approach which, seemingly, induces high latency overhead.

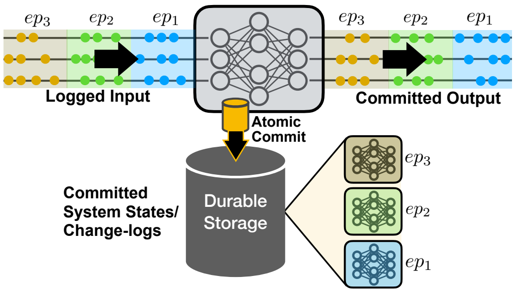


Fig. 9: Transactional Epoch Commits in Data Streaming

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, action-driven transactional processing offers deterministic executions, a property not guaranteed by epoch-based approaches. In addition, the approach has important effects on consistency as perceived by applications that consume the system's output. This follows from the fact that "exactly-once processing" in this context relates to each action being atomically committed, as we will discuss in Section 5.1.1.

4.4.2 Epoch-driven Transactional Streaming

A popular family of transactional data streaming approaches is based on the observation that every stream execution is different but valid if no failures occur. Global changes at the application level can therefore be committed at any time. Thus, instead of adopting action-level transactional processing, epoch-level approaches divide computation into a series of mini-batches, also known as "epochs".

In Figure 9 we depict the overall approach, marking input, system states and outputs with a distinct epoch identifier. Epochs can be defined through clear breakpoints at the logged input of the stream application. Whereas, 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 process happens during the execution of an epoch then the system can roll back to a previously committed epoch and recover its execution consistently. The term "exactly-once processing" in this context relates to each epoch being atomically committed. In Section 5.1 where we present the different levels of processing semantics in streaming we call this flavor *exactly-once processing on state*. The rest of this section focuses on the different known 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 the Phase-2 ensures capturing the state of the system at the end of the computation.

This approach was popularized by Apache Spark [141] through the use of periodic "micro-batching" and it is an effective strategy when batch processing systems are used for unbounded processing. The main downside of this approach is the risk of low task utilization due to synchronous execution, since tasks have to wait for all other tasks to finish their current epoch. Drizzle [137] mitigates this problem by chaining multiple epochs in a single atomic commit. A similar approach was also employed by S-Store [104], 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-executed. Furthermore, it is feasible to achieve the same functionality asynchronously through consistent snapshotting algorithms, known from classic distributed systems literature [32]. Consistent Snapshotting algorithms exhibit beneficial properties such as concurrent execution in par with an event-processing application. Furthermore, they acquire a snapshot of a consistent cut in a distributed execution. In other words, they manage to capture the global states of the system during a "valid" execution. Throughout different implementations we can identify 1. unaligned and 2. aligned snapshotting protocols.

1. Unaligned / Chandy Lamport snapshots provide one of the most efficient methods to obtain a consistent snapshot. This approach is currently supported by several stream processors, such as IBM Streams and Flink (optional support in v1.11.1). The core idea is to make use of 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 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).

2. Aligned Snapshots Aligned snapshots aim to improve performance during recovery and minimize reconfiguration complexity exhibited by unaligned snapshots. The main differentiation is to prioritize input streams that are expected before the snapshot and thus, end up solely with states that reflect a complete computation of an epoch and no events in transit as part of a snapshot. For example, Flink's epoch snapshotting mechanism [33, 35] 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 further. This 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. Furthermore, aligned snapshots yield the lowest reconfiguration footprint as well as setting the basis for live reconfiguration within the alignment phase as exhibited by Chi [99].

4.5 Vintage vs. Modern

State is a concept that has been very 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 DSMS systems [9, 16, 22, 45] (circa 2000-2010) hinted state and its management from the user. They declared and managed internally in inmemory all data structures needed to support a selected set of operations. This type of state, often referred to as “summary” was used to internally materialize continuous processing operators such as those of the time-varying relational model of CQL [18], as seen in STREAM [16].

A decade later, scalable data computing systems based on the MapReduce [54] 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, NewSQL databases). Similarly, a growing number of scalable data stream processing systems [12,13,34,107] married principles of scalable computing with stream semantics and models that were identified in the past (e.g. out-of-order processing [96,121]). This pivoting helped stream management technology to lift all assumptions associated with limited state capacity and thus reach its nearly full potential of executing correctly continuous event-driven applications with arbitrary state.

As of today, modern stream processors can compile and execute graphs of long-running operators with complete, user-defined state yet system-managed that is fault-tolerant and reconfigurable given a clear set of transactional guarantees [12,33,37].

4.6 Open Problems

Data streaming covers many data management needs today that go beyond real-time analytics, the original purpose of the technology. New needs include support for more complex data pipelines with implicit transactional guarantees. Furthermore, modern applications involve Machine Learning, Graph Analysis and Cloud Apps, all of which have a common denominator: complex state and new access patterns. These needs have cultivated novel research directions in the emerging field of stream state management.

The decoupling of state programming from state materialization resembles how database technology has evolved, prior to data streaming. 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 evolving stream processors in terms of performance capabilities. A recent study [79] 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 [44, 146] for local state management with varying capabilities are being adopted by stream processors. This opens new capabilities for optimization and sophisticated, yet user-agnostic state management that can automate the process of selecting the right physical plan and reconfigure that plan while unbound applications are executed.

Complex access patterns such as inter- , intra- and external access to shared state [103] and necessitate new type of guarantees. This requirement gives birth to yet another interesting research system direction such as ensuring state access isolation [33] (e.g. read-committed access), efficient shared state materialization [113], [46] and reliable reconfiguration.

5 Fault Tolerance & High Availability

Fault tolerance is a system’s capacity to continue its operation in spite of failures delivering the expected service as if no failures had happened. It is specially important for streaming systems for two reasons. First, streaming systems conduct stateful computations over potentially unbounded data streams. Without fault tolerance streaming systems would have to redo computations from the beginning given that the state or progress thus far would be lost during a failure. Besides losing processing progress accumulated over an arbitrary time period, recomputation is many times infeasible because the already processed segment of a data stream has permanently vanished.

Second, contemporary streaming systems feature a distributed systems architecture for scalability. In a system deployed on multiple physical machines failures occur commonly. Based on this motivation, a lot of exciting work has

Table 3: Fault-tolerance in streaming systems

System	Processing semantics			Replication			Recovery data			Storage medium			
	Least	Exactly-once		Active	Passive	No	State	Output	No	Resilient store		In-Memory	No
		State	Output							Local	Remote		
Aurora* [48]	✓				✓			✓				✓	
TelegraphCQ [119]		✓		✓			✓	✓		✓			
Borealis [8, 26]	✓			✓			✓	✓				✓	
S4 [111]	✓					✓			✓				✓
Seep [37, 57]			✓		✓		✓	✓			✓		
Naiad [108]		✓			✓		✓	✓			✓		
Timestream [114]			✓		✓		✓	✓			✓		
Millwheel [12]			✓		✓		✓	✓			✓		
Storm [131]	✓					✓			✓				✓
Trident [3]			✓		✓		✓				✓		
S-Store [39, 126]		✓			✓		✓			✓			
Trill [41]		✓			✓		✓			✓			
Heron [90]	✓					✓			✓				✓
Streamscope [97]			✓$_{a,p,r}$	✓$_{a}$	✓$_{p}$	✓$_{r}$	✓$_{p}$		✓$_{a,r}$		✓$_{p}$		✓$_{a,r}$
Streams [77]		✓			✓		✓				✓		
Samza [112]	✓				✓		✓			✓			
Flink [33, 34]		✓			✓		✓				✓		
Spark [20]		✓			✓		✓				✓		

been performed on fault tolerance in streaming systems. We present it in Section 5.1.

In computer systems, availability is defined as the time period that a system accomplishes its service relative to service interruption periods. It is typically quantified as a percentage, 100% being perfect availability [65]. The term high availability has been adopted to denote that a system achieves a very high percentage of availability like 99.999% or higher.

In stream processing where systems are not probed by users as in the case of typical information systems like web applications, what service accomplishment means is open to interpretation. Surprisingly, no definition for high availability is provided in the stream processing literature. Existing research (Section 5.2) quantifies high availability using combinations of three metrics, namely recovery time, performance overhead in terms of throughput and latency, and resource utilization. We highlight the absence of a definition and suitable metric for high availability in the open problems in Section 5.4 where we propose a definition based on processing progress and a proxy for measuring high availability based on end-to-end latency. Before finishing with the open problems, we separate the vintage from the modern in fault tolerance and high availability in Section 5.3.

5.1 Fault-tolerance

Many important challenges in stream processing manifest when we take into account failures. Managing failures in a distributed streaming system entails maintaining snapshots of state, migrating state, and scaling out operators while affecting as least as possible the healthy parts of the system. Table 3 presents the fault-tolerance strategies of eighteen streaming systems arranged in order of publication appearance from past to present. We analyse the strategies across the following four dimensions.

1. Processing semantics conveys how a system's data processing 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, typically, output is 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 [20, 34, 77, 97, 112]. 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.

Exactly-once 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 that exactly-once processing semantics on *state*. While a system can restore its state to a consistent snapshot, the same is not feasible in general to accomplish with the output published by the system. Once the output is out, it is available for consumption by external applications. Thus, a system with exactly-once processing semantics on state will still produce duplicate output. This problem has been termed the output commit problem [56] in the distributed systems literature. Systems that manage to produce the same output under failure as a failure-free execution have exactly-once processing semantics on *output*. In Section 5.1.1 we elaborate how streaming systems treat the output commit problem.

2. Replication regards the use of additional computational resources for recovering an execution. We adopt the terminology of Hwang et al. [76] that classify replication as either *active* where two instances of the same execution run in parallel or *passive* where each running stateful operator that is part of an execution dispatches its checkpointed state to a standby operator.

3. 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. However, we exclude this fact from the table to save space.

4. Storage medium states where recovery data is stored. It can be in a *resilient store* that is *local* to each stateful operator, in a *remote* resilient store, or in the memory space of a stateful operator. *In-memory* means that operators use their memory space as a primary storage medium for recovery data. Systems that cache data for recovery in memory like output tuples do not fall in this category.

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. First, the table contains three more annotations besides the self-explanatory checkmarks. Streamscope [97] presents and evaluates three distinct fault tolerance strategies, an active replication-based strategy (*a*), a passive one (*p*), and a strategy that relies on recomputing state by replaying data from input streams (*r*). Second, 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 [112] or state dependencies [114].

The table reveals four interesting patterns. First, of all columns, two accumulate the majority of checkmarks, passive replication and storing state for recovery. This is perhaps the most visible pattern on the table that 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. 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 Section 5.1.1. Finally, among the various storage media for recovery data a remote resilient store is the clear winner.

5.1.1 The output commit problem

The output commit problem [56] specifies that a system should only publish output to the outside world when it is certain that it can recover the state from where the output was published so that every output is only published once because output cannot be retracted once it is sent. If output is sent twice, then the system manifests inconsistent behavior with respect to the outside world. An important instance of this problem manifests when a system is restoring some previous consistent state due to a failure. In contrast to the system's state, its output cannot be retracted in general. Thus, under failures, systems must be careful not to produce duplicate output.

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. Streaming systems that solve the output commit problem provide output exactly-once. Other terms that refer to the same problem are processing output exactly-once and its paraphrases, as well as precise recovery [76] and strong productions [12].

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 in three categories, transaction-based, progress-based, and lineage-based, and describe each noting the assumptions it involves. Each of the three types of techniques, use a different trait of the input or computation, to identify whether a certain tuple has appeared again. Transaction-based techniques use tuple identity, progress-based techniques use order, while 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 practically, 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 [12] and Trident [3] 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. If a delivered record is retried it is ignored by checking the unique id that it carries. 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 [57] uses timestamp comparison to deliver output exactly-once 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.

A previous version of Seep [37] applies the same process with the difference that a recovered operator rewinds its logical clock to the timestamp of the checkpoint it possesses before emitting records. The system assumes deterministic computations without side-effects and a monotonically increasing logical clock providing timestamps. It further assumes that records in a stream are ordered by their timestamps.

Lineage-based. Timestream [114] and Streamscope [77] 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 made available. Streamscope follows a similar process, but starts from a failed operator's checkpoint snapshot. For each input sequence number in that snapshot 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 a subtly different manner. Timestream computes the input records required by upstream operators in reverse topological order from the final output to the original input and discards those unneeded. Streamscope does the same but instead of computing dependencies, it uses low watermarks per operator and per stream to discard snapshots and records that are behind. 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.

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

System	Assumptions
Millwheel	
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

The time-based and lineage-based solutions are vulnerable to failures of the last operator(s) on the dataflow graph, which produce the final output, since both solutions rely on downstream operators for filtering duplicate records.

Special sink operators. Streams [77] 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 [77], Flink [34], and Spark [20] 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 categorise the solutions provided by special sink operators and external sinks, is as optimistic output techniques, that push output immediately and retract it or update it if needed, and pessimistic output techniques that use a form of write ahead log, to write the output they will publish, if everything goes well until the output is permanently committed [33]. 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.

5.2 High availability

Empirical studies of high availability in stream processing [76] propose an active replication approach [26, 119], a passive replication approach [66, 75, 92], a hybrid active-passive replication approach [71, 122, 145], or model multiple approaches and evaluate them with simulated experiments [40, 76].

Active replication. Flux [119] 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. Contrastingly, Borealis [26] 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. [75] propose that a server in a cluster has another server 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 [92] saves computational resources in another way by checkpointing state asynchronously to a distributed file system. Upon a 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 [66] 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. [145] 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 the provided experiment results, their approach saves 66% recovery time compared to passive replication and produces 80% less message overhead than active replication. Alternatively, Heinze et al. [71] 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. Similarly, Su et al. [122] counter correlated failures by passively replicating processing tasks except for a dynamically selected set that is actively replicated.

Modeling and simulations. In their seminal work Hwang et al. [76] model and evaluate the recovery time and runtime overhead of four recovery approaches, active standby, passive standby, upstream backup, and amnesia, 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 [40], 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 [41], active replication with periodic checkpoints is proved advantageous in many streaming workloads, although no single strategy is appropriate for all of them.

5.3 Vintage vs. Modern

In the early years streaming systems put emphasis on high availability setups with preference towards active replication. Contrastingly modern systems tend to leverage passive replication especially by allocating extra resources on demand that is appropriate for Cloud setups. In addition, past systems provided approximate 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 gains traction is to outsource the deduplication of output to external systems. Finally, while streaming systems used to store their output in order to be able to replay tuples to downstream operators recovering from a failure, now systems rely increasingly on replayable input source for replaying input subsets.

5.4 Open Problems

Many problems wait to be solved in the scope of fault tolerance and high availability in streaming systems. Three of them include 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 like for 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 types 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 what availability

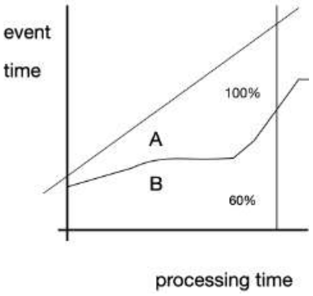


Fig. 10: Measuring availability with the slack between processing time and event time over time

means in the area of stream processing. The generic definition of availability for computer systems by Gray et al. [65] 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 streaming 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, like checkpoints, state migration, garbage collection, and calls to external systems. The common denominator of those factors, is that the system falls behind input. This may not be a problem for other types of systems, like databases which can respond to queries with the historical data they keep, but streaming systems have to continuously catch up processing with the input in order to provide correct results, that is, in order to be available.

Thus, a more specific definition of availability for stream processing can be stated in the following way. *A streaming system is available when it can provide output based on the processing of its current input.* This definition extends to how we measure availability. 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 Figure 10. The area in the plot signifies 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.

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

6 Load management, elasticity, & 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 involving 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 idle when the input load decreases.

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

6.1 Load shedding

Load shedding [24, 123, 124, 133] 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 can be formulated as an optimization problem. Let N be the query network, I the set of input streams with known arrival rates, and C the system processing capacity. Further, consider the *headroom factor*, H, as a conservative estimate of the percentage of resources required by the system at steady state. If $Load(N(I))$ denotes the load as a fraction of the total capacity C that network $N(I)$ presents, and U_{acc} is the aggregate utility, then the load shedder needs to identify a new network N', such that

$$Load(N'(I)) < H * C \quad (1)$$

under the constraint that the utility loss

$$U_{acc}(N(I)) - U_{acc}(N'(I)) \quad (2)$$

is minimized.

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.

6.1.1 Detecting overload

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. A statistics manager module monitors processing and input rates and periodically estimates operator selectivities. It feeds those metrics to a load manager module, which makes load shedding decisions. The load manager assigns a cost, c_i, in cycles per tuple, and a selectivity, s_i, to each operator i. The statistics manager collects metrics and estimates those parameters either continuously or by running the system for a designated period of time, prior to regular query 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 [52, 82, 124]. Other operator-restricted load shedding techniques target window operators [24, 125], or even more specifically, query plans with `SUM` or `COUNT` sliding window aggregates [24].

An alternative, operator-independent approach, is to frame load shedding as a feedback control problem [133]. The load shedder relies on a dynamic model that describes the relationship between average tuple delay (latency) and input rate. To build such a controller, the control signal is the desirable input rate, the output signal is the tuple latency measured by a monitor, arrival rates and processing rates are considered disturbances, and the target output is the desired tuple latency. In this setting, the objective of the feedback control loop is to maintain the average tuple latency under a target value by shedding load when necessary. This model is applicable if the per-tuple processing cost is constant and tuples are consumed by the stream processor in arrival order.

6.1.2 Reacting to overload

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.

Where to shed load. 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 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 [124]. This is a pre-computed table that contains materialized load shedding plans, ordered by the amount of load shedding they will cause. Each row in the LSRM contains a plan with expected cycle savings, locations for drop operations, drop amounts, and, provided that tuples can be associated with a utility metric, QoS effects.

Which tuples to shed. 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 can be employed in the case of sliding window aggregate queries. Approximate results can be provided by inserting random sampling operators in the query plan [24], parametrized with a *sampling rate*. This rate defines the probability to discard a tuple and is computed based on statistics and operator selectivity. The optimization objective is to achieve the highest possible accuracy given the constraint that system throughput matches the data input rate. In the case of known aggregation functions, results can be scaled using approximate query processing techniques, where accuracy is measured in terms of the relative error in the computed query answers. If queries are assembled into a single dataflow plan, the optimization objective entails minimizing the maximum error across all queries.

Window-aware load shedding [125] applies shedding to entire windows instead of individual tuples. When discarding tuples at the sources or another point in a query with multiple window aggregations, it is unclear how shedding will affect the correctness of downstream window operators. This approach preserves window integrity and guarantees that the results under shedding will not be approximations but a subset of the exact answers.

Concept-driven load shedding [84] is a semantic dropping strategy that selects tuples to discard based on the notion of window-based concept drift. The drift is calculated within window boundaries by taking into account common elements and a similarity metric across window contents. Concept-driven load shedding currently supports window group aggregations and equi-joins.

How many tuples to shed. The amount of tuples to discard strongly depends on the decisions of where and which tuples to shed. If input rates and processing capacity are known or easy to measure, estimates can be computed in a straightforward manner. However, conditions can be more complex in practice. Estimations based on static operator selectivities and heuristics are unsuitable for frequent load fluctuations [133]. Naive approaches can lead to system instability or unnecessary load shedding. The feedback control-based method is also challenging to apply, as it requires manual parameter tuning. Pole placement and the damping factor significantly impact the system's performance, in terms of convergence rate and accuracy. Further, accurately measuring the output signal, which is the per-tuple delay time, is not easy to achieve. Measurements happen with a delay of an unknown amount, which is, in fact, the output itself. One way to address this peculiar challenge is to estimate delay indirectly, by counting queue lengths (outstanding tuples) instead.

In window-aware load shedding [125], queries need to define an application-specific maximum tolerance to gaps. This parameter indicates how many consecutive missing results the query can tolerate. Given a shared query plan, the load shedder must respect the maximum gap tolerance of all queries, under the provided load constraint. In the worst case, admission control is employed and expensive queries are chosen for complete shutdown.

Adapting the data rate. An alternative to explicitly dropping tuples is to adapt the streaming data rate to match available bandwidth [143]. This is achieved by a *maybe* operator, similar to load shedding operators, that defines the degradation behavior of computations. Upon encountering network congestion, the adaptation algorithm increases the degradation level to reduce the rate, so that no persistent queue builds up. To recover, it progressively decreases the degradation level after probing for more available bandwidth.

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.

6.2.1 Load-aware scheduling

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 [21, 25]. Alternatively, adaptive scheduling [23, 36] modifies the allocation of resources given a static query plan.

In its simplest form, adaptive scheduling assumes a fixed memory budget and a single thread that executes all operators and makes scheduling decisions. Existing methods are applicable to acyclic plans with a restrictive set of operators, such as selections, projections, filters, joins with stored relations, and per-record sliding window joins. The state size per operator is considered bounded and the runtime engine utilizes a common system memory pool that all operators share for storing their input tuples.

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. To compute these priorities, the scheduler builds a *progress map* of the query plan which represents a mapping of the operator paths to a 2-dimensional plane of tuple size (intermediate results size) over time (execution cost). Operators with a high potential of reducing the intermediate results' size quickly are mapped to deeper slopes on this plane. These operators are assigned higher priorities.

Figure 11 shows an example progress map as introduced by the Chain scheduling approach [23]. This map indicates the size of intermediate data as tuples move along the dataflow graph. The scheduler then computes the chart's *lower envelope* which is the sequence of the steepest slope segments. Those segments reveal the operators that have the higher potential to reduce intermediate data in the shortest amount of time. The scheduler groups operators in *chains* (highlighted with different colors in the figure) corresponding to segments in the lower envelope of the progress chart. It then assigns priorities to operator chains based on the fraction of tuples they are likely to eliminate per unit of time.

6.2.2 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

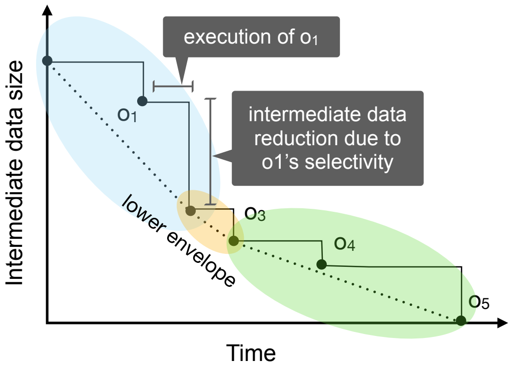


Fig. 11: An example progress map built by Chain scheduling.

sources. To ensure no data loss, a persistent input message queue, such as Apache Kafka, and adequate storage space are required.

Buffer-based flow control. 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 12a demonstrates buffer-based flow control when the producer and the consumer run on the same machine. We assume that each produced and consumed stream has managed buffer pools with bounded capacity. A buffer pool is simply a set of buffers which are recycled after they have been consumed and can be re-used. 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 will be 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.

Figure 12 demonstrates the case when the producer and the consumer are deployed on different machines. In this case, results are transferred via the network, often via a TCP connection. If no buffer is available on the consumer side, the TCP connection will be interrupted. 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.

Buffer-based flow control is a simple mechanism where the buffer occupancy controls the data rate automatically. However, when parallel tasks are connected via virtual channels multiplexed over TCP connections, the presence of data skew might overload a single channel and affect the entire dataflow. The technique we discuss next addresses this issue.

Credit-based flow control. (CFC) [91] is a link-by-link, per virtual channel congestion control technique used in ATM network switches. To exchange data through an ATM network, each pair of endpoints first needs to establish a virtual circuit (VC) or connection. CFC maximizes network utilization and prevents faults caused by high congestion. In the presence of bursty traffic, CFC causes backpressure to build up fast and propagate along congested VCs to their sources which can be throttled. Essentially, CFC allows blocking excess traffic outside the network to protect it. In a nutshell, CFC uses a credit system to signal the availability of buffer space from receivers to senders. Senders maintain a credit balance for all their receivers and receivers regularly send notifications upstream containing their number of available credits. One credit corresponds to some amount of buffer space so that a sender can know how much data they can afford to forward downstream.

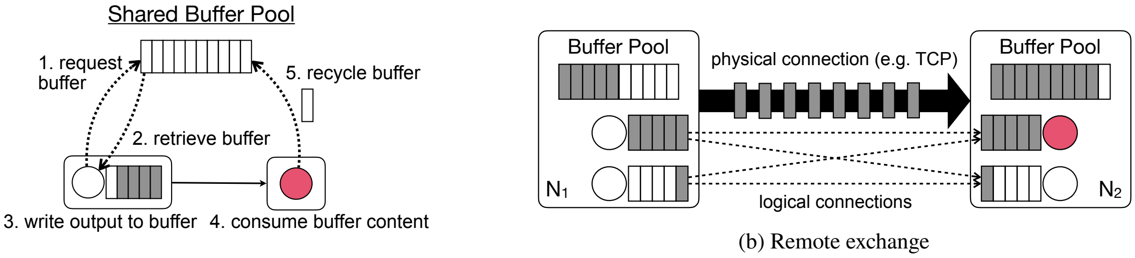
This classic networking technique turns out to be very useful for load management in modern, highly-parallel stream processors and is implemented in Apache Flink [1]. Figure 13 shows how the scheme works for a hypothetical dataflow. Parallel tasks are connected via virtual channels multiplexed over TCP 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), backpressure appears on its virtual channel.

An important advantage of this per-channel flow control mechanism is that bakcpressure is inflicted on pairs of communicating tasks only and does not interfere with other tasks sharing the same TCP connection. This 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. In fact, their models rely on the assumption that a fixed set of resources has been allocated to the stream processor and that computing and memory capacity are predetermined and will not change. However, as stream processor architectures started shifting to shared-nothing, distributed clusters and cloud deployments, load management could be addressed by less rigid approaches.

Stream processors deployed on cloud environments or clusters have access to 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-



(a) Local exchange

(b) Remote exchange

Fig. 12: Buffer-based flow control.

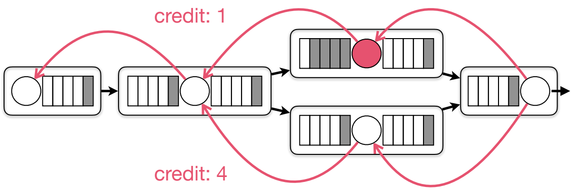


Fig. 13: Credit-based flow control in a dataflow graph. Receivers regularly announce their credit upstream (gray and white squares indicate full and free buffers, respectively).

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 individual decisions. First, it needs to detect the symptoms of an unhealthy computation and decide whether scaling is necessary. Symptom detection is a well-understood problem and can be addressed using conventional monitoring tools. Second, 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 support automatic scaling include Google Cloud Dataflow [86], Heron [90], and IBM System S [63], while DS2 [80], Seep [37] and StreamCloud [67] are recent research prototypes.

We categorize policies into *heuristic* and *predictive*. Heuristic policies rely on sets of empirically predefined rules and are often triggered by thresholds and observed conditions. On the other hand, predictive policies make scaling decisions guided by analytical performance models.

Heuristic policies. 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 [116]. StreamCloud [67] and Seep [37] 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 [86] relies on CPU utilization for scale-down decisions only but still suffers false negatives. Dhalion [58] and IBM Streams [63] also use congestion and back-pressure signals to identify bottlenecks. These metrics are helpful for identifying bottlenecks but they cannot detect resource over-provisioning.

Heuristic scaling policies are expressed by a set of rules, using predefined thresholds and conditions, e.g. *if CPU utilization > 50% and back-pressure* $\implies$ *scale up*. Careful and continuous threshold tuning is a cumbersome yet necessary process. Slightly misconfigured thresholds might cause incorrect scaling decisions even when relying on fine-grained metrics. For lack of an analytical performance model, scaling actions are *speculative*, as the system explores the effects of reconfiguration. Most policies configure a single operator at a time, requiring many iterations to find a good configuration. More aggressive strategies test alternative configurations and blacklist them if they end up degrading performance.

Predictive policies. 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 [59, 59, 98, 130], control theory [15, 85, 102], and instrumentation-driven linear performance models [80]. Thanks to their closed-form analytical formulation, predictive policies are capable of making multi-operator decisions in one step.

Selecting an appropriate queuing network model to represent streaming computations is challenging. Simple mod-

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 [8]	✓		✓	✓	n/a			n/a	
StreamCloud [67]	✓			✓		✓		✓	
Seep [37]	✓		✓	✓		✓		✓	
IBM Streams [63]	✓			✓		✓		✓	
FUGU [69, 70]	✓			✓		✓		✓	
Nephele [98]		✓	✓						
DRS [59]		✓	✓						
MPC [102]		✓	✓			✓		✓	
CometCloud [130]		✓	✓				✓	n/a	
Chronostream [138]	n/a		n/a				✓	✓	
ACES [15]		✓	✓	✓	n/a			n/a	
Stella [139]	✓			✓					
Google Dataflow [86]	✓		✓	✓					
Dhalion [58]	✓			✓	✓			✓	
DS2 [80]		✓		✓	✓			✓	
Spark Streaming [20, 140]	✓			✓	✓			✓	
Megaphone [74]						✓			✓
Turbine [105]	✓			✓	✓			✓	
Rhino [55]	n/a		n/a			✓		✓	

els with closed-form solutions make strong assumptions about their inputs, the arrival distribution, the queue properties, and their outputs. Such models are incapable of accurately capturing the behavior of large streaming dataflows, as they expect every element arriving at an operator to be processed and leave the queue. This assumption is at odds with the semantics of custom window operators, joins, and special event processing, such as watermark propagation. Complex queuing network models, on the other hand, either have no closed-form analytical solutions or only numerical solutions are known. Most queuing theory controllers adopt models where the probability distributions of data item inter-arrival and service times are generally unknown. To model a computation with multiple operators, each parallel task of a dataflow node can be represented as a single-server GI/G/1 queuing system. The model predicts the actual queue waiting time per task when its parallelism changes using the currently measured queue waiting time for the current degree of parallelism. The single-server model can be extended to a generalized Jackson network where each operator is represented by a stochastically independent GI/G/k service node.

Control-theory based approaches either consider the entire dataflow graph as a black-box system or regard each operator as a separate feedback-loop system. Feedback mechanisms require measuring an input and output signal, which correspond to the stream input rate and the tuple delay (latency), respectively. However, modeling the delay as the output signal in turn indicates that the feedback system is approximating a variable whose measurement can be arbitrarily delayed. Another challenge with control-theoretic approaches is accurate parameter estimation. Poles placement, sampling period, and damping are instrumental to the controller's performance and must be determined offline. For instance, an unnecessarily high damping can cause instability while too low value slows down convergence. Viewing the computation as a black box is also problematic as the controller cannot identify individual bottlenecks.

Both queuing and control theoretic approaches require fine-grained metrics collection, often at the granularity of a single tuple. Example metrics include task latency, mean and variance of a task's service time (how long a task is busy with a data item), mean and variance of a task's inter-arrival time, channel latency (mean time between a data item being emitted from its producer and being processed by its consumer), and output batch latency (mean time data items wait due to batching before actually being shipped). Further, they might include measuring the total time spent on processing each tuple and intermediate results derived from it, as well as the total time a tuple and its derived tuples wait in queues. Random sampling or window-based metrics collection is often used to avoid the overhead of dense monitoring.

Instrumentation-based elasticity [80] combines a general performance model of streaming dataflows with lightweight instrumentation to estimate the true processing and output rates of individual dataflow operators. As opposed to observed elapsed time, *useful time* is defined as the time spent by an operator instance in deserialization, processing, and serialization activities. Essentially, useful time amounts to the time an operator instance runs for if executed in an *ideal* setting where it never has to wait to obtain input or push output. Using this notion, the true processing (resp. output) rate corresponds to how many records an operator instance can process (resp. output) per unit of useful time. True rates denote the maximum processing and output rate the instance could sustain for the current workload. Using lightweight instrumentation to periodically measure the true processing and output rates of individual operators, the controller builds a continuously updated linear performance model of streaming computations and makes scaling accurate decisions with negligible overhead.

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 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 locally managed state, as reconfiguration mechanisms are significantly simplified when state is external.

State redistribution. A straight-forward approach to state redistribution is to have all new tasks load the entire state from a checkpoint and filter out keys they are not responsible for. Despite the advantage of fast sequential reads, such a strategy would overload the file system and cause tasks to read mostly unnecessary state. Another simple strategy is to track the state location for each key in the checkpoint, so that tasks can locate and read matching keys only. This strategy would avoid reading irrelevant data, but it would incur a large amount of random I/O. Further, it would require a materialized index for all keys, which could potentially grow very large. State redistribution must preserve key semantics, so that existing state for a particular key and all 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. On average M/N partitions are moved when the N^{th} worker is inserted or removed from a system with M partitions. Apache Flink [34] 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* [62] 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 [101] in a sliding window setting to estimate heavy hitters, as keeping exact counts would be impractical for large key domains.

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 very common in modern systems.

Partial pause and restart, introduced by FLUX [120], is a less disruptive strategy that only blocks the affected dataflow subgraph temporarily. The affected subgraph contains the operator to be scaled, as well as upstream channels and upstream operators. Figure 14 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.

The *pro-active replication* strategy maintains state backup copies in multiple nodes so that reconfiguration can be performed in a nearly live manner when needed. The state is organized into smaller partitions, each of which can be transferred independently. Each node has a set of pri-

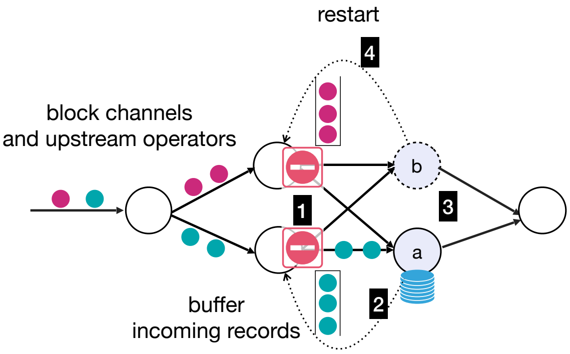


Fig. 14: An example of the partial-pause-and-restart protocol. To move state from operator *a* to *b*, the mechanism executes the following steps: (1) Pause *a*'s upstream operators, (2) extract state from *a*, (3) load state into *b*, and (4) send a restart signal from *b* to upstream operators.

mary state slices and a set of secondary state slices. Figure 15 shows an example of the protocol as implemented by ChronoStream [138]. The migration is coordinated by a leader. To move slice #1 from N_{src} to N_{dest}, (1) the leader instructs N_{dest} to load slice #1 and upgrade it from a backup slice to a primary slice. (2) The destination node loads slice #1 and sends an acknowledgement to the leader together with the slice's progress. In the meantime, the source node keeps processing events destined for slice #1. (3) The leader notifies upstream operators to replay events according to the progress metric provided by N_{dest}. (4) Upstream nodes receive the message and start rerouting events to N_{dest}. For some time interval, both N_{src} and N_{dest} process events for slice #1. As a result, downstream operators need to implement a de-duplication mechanism. (5) Once the leader notifies N_{src} that the transfer is complete, N_{src} consumes any remaining data and moves slice #1 to the backup group.

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 [74] 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 Vintage vs. Modern

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 results quality anymore. Another important difference is that load management approaches in vintage systems used to affect the execution of multiple queries as they formed a shared dataflow plan (cf. Section 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.

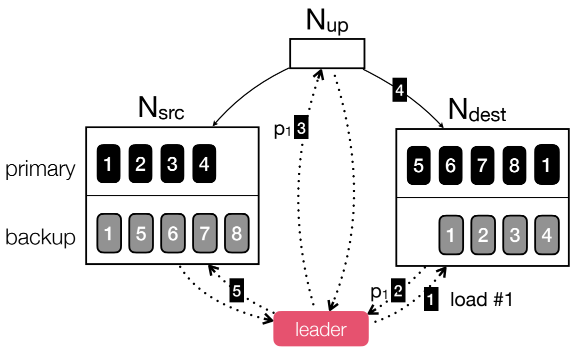


Fig. 15: An example of the proactive replication protocol. To move slice #1 from N_{src} to N_{dest}, the mechanism executes the following steps: (1) the leader instructs N_{dest} to load slice #1, (2) N_{dest} loads slice #1 and sends ack to the leader, (3) the leader notifies upstream operators to 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.

6.5 Open Problems

Adaptive scheduling methods have so far been studied 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 allocation but other elements of their runtime as well. Elasticity, the ability of

Table 6: Evolution of streaming systems

	Vintage (1st generation)	**Modern (2nd-3rd generation)**
Results	approximate or exact	exact
Language	SQL extensions, CQL	Java, Scala, Python, SQL-like
Query plans	global, optimized, with pre-defined operators	independent, with custom operators
Execution	centralized	distributed
Parallelism	pipeline	data, pipeline, task
Time & 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

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 Conclusion

While early streaming systems strove to extend relational execution engines with time-based window processing, modern systems have evolved significantly in terms of architecture and capabilities. Table 6 summarizes the evolution of major streaming system aspects over the last three decades.

While approximate results were mainstream in early systems, modern systems have primarily focused on results correctness and have largely rejected the notion of approximation. In terms of languages, modern systems favor general-purpose programming languages, however, we recently witness a trend to return to extensions for streaming SQL [28]. Over the years, execution has also gradually transitioned from mainly centralized to mainly distributed, exploiting data, pipeline, and task parallelism. At the same time, most modern systems construct independent execution plans per query and apply little optimization and sharing.

Regarding time, order, and progress, many of the inventions of the past proved to be vintage at the test of time, since they continue to hold a place in modern streaming systems. Especially 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. As a result, fault tolerance and high availability also shifted towards passive replication and exactly-once processing. Finally, load management approaches have transitioned from load shedding and scheduling methods to elasticity and backpressure coupled with persistent inputs.

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. Yet, today's systems have invested thoroughly in providing transactional guarantees that are in par with those modern database management systems can offer today. Transactional stream processing has pivoted data streaming beyond the use for data analytics 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 allows for better optimizations, wider interoperability with storage technologies as well as novel semantics for shared and external state having stream processors as the backbone of modern continuous applications and live scalable data services.

We believe the road ahead is still long for streaming systems. Emerging streaming applications in the areas of Cloud services [11, 64], machine learning [60, 106], and streaming graph analytics [10, 29] present new requirements and are already shaping the key characteristics of the future generation of data stream technology. We expect systems to evolve further and exploit next-generation hardware [142, 144], focus on transactions and iteration support, improve their re-configuration capabilities, and take state management a step further by leveraging workload-aware backends [79], shared state and versioning.

References

1. Apache Flink. `http://flink.apache.org/`. Last access: July 2020.
2. Apache Storm. http://storm.apache.org/. Last access: July 2020.
3. The Trident Stream Processing Programming Model. `http://storm.apache.org/releases/0.10.0/Trident-tutorial.html`. Last access: July 2020.
4. Introduction to Kafka Streams. `http://www.confluent.io/blog/introducing-kafka-streams-stream-processing-made-simple`, 2017.
5. Rocksdb. `http://rocksdb.org/`, 2017.

6. Kafka-Streams Documentation. `https://kafka.apache.org/documentation/streams/`, 2020.
7. Redis. `https://redis.io/`, 2020.
8. D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, et al. The design of the Borealis stream processing engine. In *CIDR*, 2005.
9. D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A new model and architecture for data stream management. *VLDBJ*, 12(2):120–139, 2003.
10. Z. Abbas, V. Kalavri, P. Carbone, and V. Vlassov. Streaming graph partitioning: an experimental study. *In VLDB*, 2018.
11. A. Akhter, M. Fragkoulis, and A. Katsifodimos. Stateful functions as a service in action. *In VLDB*, 2019.
12. T. Akidau, A. Balikov, K. Bekiroglu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. MillWheel: Fault-tolerant stream processing at internet scale. In *VLDB*, 2013.
13. T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, 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.
14. M. Ali, B. Chandramouli, J. Goldstein, and R. Schindlauer. The extensibility framework in Microsoft Streaminsight. In *ICDE*, 2011.
15. L. Amini, N. Jain, A. Sehgal, J. Silber, and O. Verscheure. Adaptive control of extreme-scale stream processing systems. *In ICDCS*, 2006.
16. A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom. Stream: The stanford data stream management system. *Book chapter in "Data Stream Management: Processing High-Speed Data Streams"*, 2004.
17. A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, I. Nishizawa, J. Rosenstein, and J. Widom. STREAM: the stanford stream data manager. In *SIGMOD*, 2003.
18. A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: Semantic foundations and query execution. *VLDBJ*, 15(2):121–142, 2006.
19. A. Arasu and J. Widom. Resource sharing in continuous sliding-window aggregates. In *VLDB*, 2004.
20. M. Armbrust, T. Das, J. Torres, B. Yavuz, S. Zhu, R. Xin, A. Ghodsi, I. Stoica, and M. Zaharia. Structured streaming: A declarative API for real-time applications in Apache Spark. In *SIGMOD*, 2018.
21. R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. *SIGMOD Record*, 29(2):261–272, 2000.
22. B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *PODS*, 2002.
23. B. Babcock, S. Babu, R. Motwani, and M. Datar. Chain: Operator scheduling for memory minimization in data stream systems. In *SIGMOD*, 2003.
24. B. Babcock, M. Datar, and R. Motwani. Load shedding for aggregation queries over data streams. In *ICDE*, 2004.
25. S. Babu, R. Motwani, K. Munagala, I. Nishizawa, and J. Widom. Adaptive ordering of pipelined stream filters. In *SIGMOD*, 2004.
26. M. Balazinska, H. Balakrishnan, S. R. Madden, and M. Stonebraker. Fault-tolerance in the Borealis distributed stream processing system. *ACM TODS*, 33(1):44, 2008.
27. R. S. Barga, J. Goldstein, M. H. Ali, and M. Hong. Consistent streaming through time: A vision for event stream processing. In *CIDR*, 2007.
28. E. Begoli, T. Akidau, F. Hueske, J. Hyde, K. Knight, and K. Knowles. One SQL to rule them all - an efficient and syntactically idiomatic approach to management of streams and tables. In *SIGMOD*, 2019.
29. M. Besta, M. Fischer, V. Kalavri, M. Kapralov, and T. Hoefler. Practice of Streaming and Dynamic Graphs: Concepts, Models, Systems, and Parallelism. *CoRR*, abs/1912.12740, 2020.
30. I. Botan, R. Derakhshan, N. Dindar, L. Haas, R. J. Miller, and N. Tatbul. Secret: A model for analysis of the execution semantics of stream processing systems. *In VLDB*, 2010.
31. S. Bykov, A. Geller, G. Kliot, J. R. Larus, R. Pandya, and J. Thelin. Orleans: Cloud computing for everyone. In *ACM Symposium on Cloud Computing*, 2011.
32. P. Carbone. *Scalable and Reliable Data Stream Processing*. PhD thesis, KTH Royal Institute of Technology, 2018.
33. P. Carbone, S. Ewen, G. Fóra, S. Haridi, S. Richter, and K. Tzoumas. State management in Apache Flink: Consistent stateful distributed stream processing. *In VLDB*, 2017.
34. P. Carbone, S. Ewen, S. Haridi, A. Katsifodimos, V. Markl, and K. Tzoumas. Apache Flink: Stream and batch processing in a single engine. *IEEE Data Engineering Bulletin*, 38, 2015.
35. P. Carbone, G. Fóra, S. Ewen, S. Haridi, and K. Tzoumas. Lightweight asynchronous snapshots for distributed dataflows. *arXiv preprint arXiv:1506.08603*, 2015.
36. D. Carney, U. Çetintemel, A. Rasin, S. Zdonik, M. Cherniack, and M. Stonebraker. Operator scheduling in a data stream manager. In *VLDB*, 2003.
37. R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *SIGMOD*, 2013.
38. U. Çetintemel, D. Abadi, Y. Ahmad, H. Balakrishnan, M. Balazinska, M. Cherniack, J.-H. Hwang, S. Madden, A. Maskey, A. Rasin, et al. The aurora and borealis stream processing engines. In *Data Stream Management*, pages 337–359. Springer, 2016.
39. U. Cetintemel, J. Du, T. Kraska, S. Madden, D. Maier, J. Meehan, A. Pavlo, M. Stonebraker, E. Sutherland, N. Tatbul, K. Tufte, H. Wang, and S. Zdonik. S-Store: A streaming NewSQL system for big velocity applications. *In VLDB*, 2014.
40. B. Chandramouli and J. Goldstein. Shrink: Prescribing resiliency solutions for streaming. *In VLDB*, 2017.
41. B. Chandramouli, J. Goldstein, M. Barnett, R. DeLine, D. Fisher, J. C. Platt, J. F. Terwilliger, and J. Wernsing. Trill: A high-performance incremental query processor for diverse analytics. *In VLDB*, 2014.
42. B. Chandramouli, J. Goldstein, and Y. Li. Impatience is a virtue: Revisiting disorder in high-performance log analytics. In *ICDE*, 2018.
43. B. Chandramouli, J. Goldstein, and D. Maier. On-the-fly progress detection in iterative stream queries. *In VLDB*, 2009.
44. B. Chandramouli, G. Prasaad, D. Kossmann, J. Levandoski, J. Hunter, and M. Barnett. Faster: A concurrent key-value store with in-place updates. In *SIGMOD*, 2018.
45. S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah. TelegraphCQ: Continuous dataflow processing. In *SIGMOD*, 2003.
46. F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM TOCS*, 26(2):26, 2008.
47. J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. *In SIGMOD*, 2000.
48. M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. B. Zdonik. Scalable distributed stream processing. In *CIDR*, 2003.
49. J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Google's globally distributed database. *ACM TOCS*, 31(3):22, 2013.

50. C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk. Gigascope: A stream database for network applications. In *SIGMOD*, 2003.
51. G. Cugola and A. Margara. Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.*, 44(3):61, 2012.
52. A. Das, J. Gehrke, and M. Riedewald. Approximate join processing over data streams. In *SIGMOD*, 2003.
53. M. Dayarathna and S. Perera. Recent advancements in event processing. *ACM Comput. Surv.*, 51(2):35, 2018.
54. J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.
55. B. Del Monte, S. Zeuch, T. Rabl, and V. Markl. Rhino: Efficient management of very large distributed state for stream processing engines. In *SIGMOD*, 2020.
56. E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):34, 2002.
57. R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. Making state explicit for imperative big data processing. In *USENIX ATC*, 2014.
58. A. Floratou, A. Agrawal, B. Graham, S. Rao, and K. Ramasamy. Dhalion: Self-regulating stream processing in Heron. *In VLDB*, 2017.
59. T. Z. J. Fu, J. Ding, R. T. B. Ma, M. Winslett, Y. Yang, and Z. Zhang. DRS: auto-scaling for real-time stream analytics. *IEEE/ACM Trans. Netw.*, 25(6):15, 2017.
60. P. Garefalakis, K. Karanasos, and P. Pietzuch. Neptune: Scheduling suspendable tasks for unified stream/batch applications. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 233–245. ACM, 2019.
61. M. Garofalakis, J. Gehrke, and R. Rastogi. *Data stream management: processing high-speed data streams*. Springer, 2007.
62. B. Gedik. Partitioning functions for stateful data parallelism in stream processing. *VLDBJ*, 23(4):517—539, 2014.
63. B. Gedik, S. Schneider, M. Hirzel, and K. L. Wu. Elastic scaling for data stream processing. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):17, 2014.
64. J. Goldstein, A. Abdelhamid, M. Barnett, S. Burckhardt, B. Chandramouli, D. Gehring, N. Lebeck, C. Meiklejohn, U. F. Minhas, R. Newton, R. G. Peshawaria, T. Zaccai, and I. Zhang. A.M.B.R.O.S.I.A: Providing performant virtual resiliency for distributed applications. *In VLDB*, page 588–601, Jan. 2020.
65. J. Gray and D. P. Siewiorek. High-availability computer systems. *Computer*, 24(9):10, 1991.
66. Y. Gu, Z. Zhang, F. Ye, H. Yang, M. Kim, H. Lei, and Z. Liu. An empirical study of high availability in stream processing systems. In *Middleware*, 2009.
67. V. Gulisano, R. Jiménez-Peris, M. Patiño-Martínez, C. Soriente, and P. Valduriez. StreamCloud: An elastic and scalable data streaming system. *IEEE Transactions on Parallel and Distributed Systems*, 23(12):15, 2012.
68. M. A. Hammad, M. J. Franklin, W. G. Aref, and A. K. Elmagarmid. Scheduling for shared window joins over data streams. In *VLDB*, 2003.
69. T. Heinze, Z. Jerzak, G. Hackenbroich, and C. Fetzer. Latency-aware elastic scaling for distributed data stream processing systems. In *DEBS*, 2014.
70. T. Heinze, V. Pappalardo, Z. Jerzak, and C. Fetzer. Auto-scaling techniques for elastic data stream processing. In *ICDE Workshops*, 2014.
71. T. Heinze, M. Zia, R. Krahn, Z. Jerzak, and C. Fetzer. An adaptive replication scheme for elastic data stream processing systems. In *DEBS*, 2015.
72. M. Hirzel, G. Baudart, A. Bonifati, E. Della Valle, S. Sakr, and A. Akrivi Vlachou. Stream processing languages in the big data era. *SIGMOD Record*, 47(2), 2018.
73. M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm. A catalog of stream processing optimizations. *ACM Comput. Surv.*, 46(4):34, 2014.
74. M. Hoffmann, A. Lattuada, F. McSherry, V. Kalavri, J. Liagouris, and T. Roscoe. Megaphone: Latency-conscious state migration for distributed streaming dataflows. *In VLDB*, 2019.
75. J. Hwang, Y. Xing, U. Cetintemel, and S. Zdonik. A cooperative, self-configuring high-availability solution for stream processing. In *ICDE*, 2007.
76. J.-H. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik. High-availability algorithms for distributed stream processing. In *ICDE*, 2005.
77. G. Jacques-Silva, F. Zheng, D. Debrunner, K.-L. Wu, V. Dogaru, E. Johnson, M. Spicer, and A. E. Sariyüce. Consistent regions: guaranteed tuple processing in IBM Streams. *In VLDB*, 2016.
78. T. Johnson, S. Muthukrishnan, V. Shkapenyuk, and O. Spatscheck. A heartbeat mechanism and its application in Gigascope. In *VLDB*, 2005.
79. V. Kalavri and J. Liagouris. In support of workload-aware streaming state management. In *12th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*, 2020.
80. V. Kalavri, J. Liagouris, M. Hoffmann, D. Dimitrova, M. Forshaw, and T. Roscoe. Three steps is all you need: Fast, accurate, automatic scaling decisions for distributed streaming dataflows. In *OSDI*, 2018.
81. R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. Jones, S. Madden, M. Stonebraker, Y. Zhang, et al. H-store: a high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment*, 1(2):1496–1499, 2008.
82. J. Kang, J. F. Naughton, and S. D. Viglas. Evaluating window joins over unbounded streams. In *ICDE*, 2003.
83. A. Katsifodimos and M. Fragkoulis. Operational stream processing: Towards scalable and consistent event-driven applications. In *EDBT*, 2019.
84. N. R. Katsipoulakis, A. Labrinidis, and P. K. Chrysanthis. Concept-driven load shedding: Reducing size and error of voluminous and variable data streams. In *Big Data*, 2018.
85. A. Khoshkbarforoushha, A. Khosravian, and R. Ranjan. Elasticity management of streaming data analytics flows on clouds. *J. Comput. Syst. Sci.*, 89:24–40, 2017.
86. E. Kirpichov and M. Denielou. No shard left behind: dynamic work rebalancing in Google Cloud Dataflow (accessed: June 2020). https://cloud.google.com/blog/big-data/2016/05/no-shard-left-behind-dynamic-work-rebalancing-in-google-cloud-dataflow.
87. M. Kleppmann, A. R. Beresford, and B. Svingen. Online event processing: Achieving consistency where distributed transactions have failed. *ACM Queue*, 2019.
88. A. Koliousis, M. Weidlich, R. Castro Fernandez, A. L. Wolf, P. Costa, and P. Pietzuch. Saber: Window-based hybrid stream processing for heterogeneous architectures. In *SIGMOD*, 2016.
89. S. Krishnamurthy, M. J. Franklin, J. Davis, D. Farina, P. Golovko, A. Li, and N. Thombre. Continuous analytics over discontinuous streams. In *SIGMOD*, 2010.
90. S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja. Twitter Heron: Stream processing at scale. In *SIGMOD*, 2015.
91. H. T. Kung, T. Blackwell, and A. Chapman. Credit-based flow control for ATM networks: Credit update protocol, adaptive credit allocation and statistical multiplexing. In *SIGCOMM*, 1994.
92. Y. Kwon, M. Balazinska, and A. Greenberg. Fault-tolerant stream processing using a distributed, replicated file system. *In VLDB*, 2008.
93. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

94. J. Leibiusky, G. Eisbruch, and D. Simonassi. *Getting started with Storm.* " O'Reilly Media, Inc.", 2012.
95. J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker. No pane, no gain: Efficient evaluation of sliding-window aggregates over data streams. *SIGMOD Record*, 34(1):39–44, 2005.
96. J. Li, K. Tufte, V. Shkapenyuk, V. Papadimos, T. Johnson, and D. Maier. Out-of-order processing: A new architecture for high-performance stream systems. *In VLDB*, 2008.
97. W. Lin, H. Fan, Z. Qian, J. Xu, S. Yang, J. Zhou, and L. Zhou. STREAMSCOPE: Continuous reliable distributed processing of big data streams. In *NSDI*, 2016.
98. B. Lohrmann, P. Janacik, and O. Kao. Elastic stream processing with latency guarantees. *In ICDCS*, 2015.
99. L. Mai, K. Zeng, R. Potharaju, L. Xu, S. Suh, S. Venkataraman, P. Costa, T. Kim, S. Muthukrishnan, V. Kuppa, et al. Chi: a scalable and programmable control plane for distributed stream processing systems. *Proceedings of the VLDB Endowment*, 11(10):1303–1316, 2018.
100. D. Maier, J. Li, P. Tucker, K. Tufte, and V. Papadimos. Semantics of data streams and operators. In *ICDT*, 2005.
101. G. S. Manku and R. Motwani. Approximate frequency counts over data streams. In *VLDB*, 2002.
102. T. D. Matteis and G. Mencagli. Elastic scaling for distributed latency-sensitive data stream operators. In *PDP*, 2017.
103. F. McSherry, A. Lattuada, M. Schwarzkopf, and T. Roscoe. Shared arrangements: practical inter-query sharing for streaming dataflows. *arXiv*, pages arXiv–1812, 2018.
104. J. Meehan, N. Tatbul, S. Zdonik, C. Aslantas, U. Cetintemel, J. Du, T. Kraska, S. Madden, D. Maier, A. Pavlo, et al. S-Store: Streaming meets transaction processing. *In VLDB*, 2015.
105. Y. Mei, L. Cheng, V. Talwar, M. Y. Levin, G. Jacques-Silva, N. Simha, A. Banerjee, B. Smith, T. Williamson, S. Yilmaz, et al. Turbine: Facebook's service management platform for stream processing. *Traffic*, 40:80.
106. M. Meldrum, K. Segeljakt, L. Kroll, P. Carbone, C. Schulte, and S. Haridi. Arcon: Continuous and deep data stream analytics. In *Proceedings of Real-Time Business Intelligence and Analytics*, page 3. ACM, 2019.
107. M. Migliavacca, D. Eyers, J. Bacon, Y. Papagiannis, B. Shand, and P. Pietzuch. SEEP: Scalable and Elastic Event Processing. In *Middleware*, 2010.
108. D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataflow system. In *SOSP*, 2013.
109. D. G. Murray, F. McSherry, M. Isard, R. Isaacs, P. Barham, and M. Abadi. Incremental, iterative data processing with timely dataflow. *Communications of the ACM*, 59(10):75–83, 2016.
110. C. Mutschler and M. Philippsen. Reliable speculative processing of out-of-order event streams in generic publish/subscribe middlewares. In *DEBS*, 2013.
111. L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *ICDMW*, 2010.
112. S. A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, and R. H. Campbell. Samza: Stateful scalable stream processing at Linkedin. *In VLDB*, 2017.
113. P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.
114. Z. Qian, Y. He, C. Su, Z. Wu, H. Zhu, T. Zhang, L. Zhou, Y. Yu, and Z. Zhang. TimeStream: Reliable stream computation in the cloud. In *EuroSys*, 2013.
115. V. Raman, B. Raman, and J. M. Hellerstein. Online dynamic reordering for interactive data processing. In *VLDB*, 1999.
116. N. Rameshan, Y. Liu, L. Navarro, and V. Vlassov. Hubbub-Scale: Towards Reliable Elastic Scaling under Multi-Tenancy. In *CC-Grid*, 2016.
117. H. Röger and R. Mayer. A comprehensive survey on parallelization and elasticity in stream processing. *ACM Comput. Surv.*, 52(2):1–37, 2019.
118. E. Ryvkina, A. S. Maskey, M. Cherniack, and S. Zdonik. Revision processing in a stream processing engine: A high-level design. In *ICDE*, 2006.
119. M. A. Shah, J. M. Hellerstein, and E. Brewer. Highly available, fault-tolerant, parallel dataflows. In *SIGMOD*, 2004.
120. M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *ICDE*, 2003.
121. U. Srivastava and J. Widom. Flexible time management in data stream systems. In *PODS*, 2004.
122. L. Su and Y. Zhou. Tolerating correlated failures in massively parallel stream processing engines. In *ICDE*, 2016.
123. N. Tatbul, U. Çetintemel, and S. Zdonik. Staying FIT: Efficient load shedding techniques for distributed stream processing. In *VLDB*, 2007.
124. N. Tatbul, U. Çetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker. Load shedding in a data stream manager. In *VLDB*, 2003.
125. N. Tatbul and S. Zdonik. Window-aware load shedding for aggregation queries over data streams. In *VLDB*, 2006.
126. N. Tatbul, S. B. Zdonik, J. Meehan, C. Aslantas, M. Stonebraker, K. Tufte, C. Giossi, and H. Quach. Handling shared, mutable state in stream processing with correctness guarantees. *IEEE Data Eng. Bull.*, 38:94–104, 2015.
127. D. Terry, D. Goldberg, D. Nichols, and B. Oki. Continuous queries over append-only databases. *SIGMOD Record*, 1992.
128. J. Thomas, P. Hanrahan, and M. Zaharia. Fleet: A framework for massively parallel streaming on FPGAs. In *ASPLOS*, 2020.
129. Q.-C. To, J. Soto, and V. Markl. A survey of state management in big data processing systems. *VLDBJ*, 27(6):847–872, 2018.
130. R. Tolosana-Calasanz, J. D. Montes, O. F. Rana, and M. Parashar. Feedback-control & queueing theory-based resource management for streaming applications. *IEEE Transactions on Parallel and Distributed Systems*, 28:1061–1075, 2017.
131. A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, et al. Storm @ Twitter. In *SIGMOD*, 2014.
132. J. Traub, P. Grulich, A. R. Cuéllar, S. Breß, A. Katsifodimos, T. Rabl, and V. Markl. Efficient window aggregation with general stream slicing. In *EDBT*, 2019.
133. Y.-C. Tu, S. Liu, S. Prabhakar, and B. Yao. Load shedding in stream databases: A control-based approach. In *VLDB*, 2006.
134. P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting punctuation semantics in continuous data streams. *IEEE TKDE*, 2003.
135. T. Urhan and M. J. Franklin. Xjoin: A reactively-scheduled pipelined join operator. *IEEE Data Eng. Bull.*, 23, 2000.
136. T. Urhan and M. J. Franklin. Dynamic pipeline scheduling for improving interactive query performance. In *VLDB*, 2001.
137. S. Venkataraman, A. Panda, K. Ousterhout, A. Ghodsi, M. J. Franklin, B. Recht, and I. Stoica. Drizzle: Fast and adaptable stream processing at scale. In *SOSP*, 2017.
138. Y. Wu and K.-L. Tan. ChronoStream: Elastic stateful stream computation in the cloud. In *ICDE*, 2015.
139. L. Xu, B. Peng, and I. Gupta. Stela: Enabling stream processing systems to scale-in and scale-out on-demand. In *IC2E*, 2016.
140. M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *SOSP*, 2013.
141. M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters. In *USENIX HotCloud*, 2012.
142. S. Zeuch, B. D. Monte, J. Karimov, C. Lutz, M. Renz, J. Traub, S. Breß, T. Rabl, and V. Markl. Analyzing efficient stream processing on modern hardware. *In VLDB*, 2019.
143. B. Zhang, X. Jin, S. Ratnasamy, J. Wawrzynek, and E. A. Lee. AWStream: Adaptive wide-area streaming analytics. In *SIGCOMM*, 2018.

144. S. Zhang, F. Zhang, Y. Wu, B. He, and P. Johns. Hardware-conscious stream processing: A survey. *SIGMOD Record*, 2020.
145. Z. Zhang, Y. Gu, F. Ye, H. Yang, M. Kim, H. Lei, and Z. Liu. A hybrid approach to high availability in stream processing systems. In *ICDCS*, 2010.
146. X. Zhu, G. Feng, M. Serafini, X. Ma, J. Yu, L. Xie, A. Aboulnaga, and W. Chen. Livegraph: A transactional graph storage system with purely sequential adjacency list scans. *arXiv preprint arXiv:1910.05773*, 2019.