



A unified semantic framework for window operators under late and out-of-order data

Ahmed Awad^{1,2}

Received: 22 October 2025 / Revised: 14 September 2026 / Accepted: 3 October 2026
© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2026

Abstract

The growing adoption of stream processing engines (SPEs) has exposed critical inconsistencies in how window operators handle late and out-of-order events, undermining the portability, correctness, and maintainability of stream-based applications. We present a unified semantic framework centred on a window-operator lifecycle model that distinguishes single-buffer and multi-buffer maintenance and semantic variations for event inclusion, recomputation, and result reporting. We instantiate the model on five widely used distributed SPEs, Apache Beam, Flink, Spark, Kafka Streams, and Storm, by mapping it to their APIs and execution behaviour, and run an identical windowed query under total-, partial-, and out-of-order arrival. Even for one query over identical data, the engines diverge: under a total order, all differences reduce to two conventions (window-boundary inclusion and session-end labelling); a partial order adds order-dependent boundary counts that no fixed interval can justify (a window reported as 3 where only 2 or 4 are consistent); and out-of-order arrival splits the engines into context-aware versus context-agnostic session handling. We distil these findings into a portability checklist and a mapping of engine configuration knobs, giving developers a principled basis for configuring, testing, and porting window operators across SPEs.

Keywords Big stream processing · Stream processing semantics · Window operators · Consistency models · Event-time processing · Out-of-order data handling · Software architecture

1 Introduction

Modern software systems increasingly rely on continuous data streams, from financial analytics and vehicular monitoring [1] to intelligent transportation systems [2], healthcare, and IoT deployments [3]. These data originate from heterogeneous hardware (e.g., mobile devices, IoT sensors) and software services [4], typically arriving as continuous streams [5]. Stream processing has therefore become a first-class citizen in the big data ecosystem, providing the foundation for context-aware applications, real-time analytics, and dependable data-driven services.

Practical use cases illustrate this demand clearly: financial services require immediate detection of anomalies such

as stock market fluctuations or credit card fraud [6, 7]; vehicular networks demand real-time monitoring of traffic conditions [8]; and healthcare applications rely on continuous patient data analysis [9, 10]. In all such domains, data arrive continuously and must be processed with minimal latency. A stream processing pipeline is commonly represented as a directed acyclic graph (DAG) of operators, where each operator transforms inputs to outputs. Since streams are unbounded, stateful computations cannot be directly expressed. To address this, stream processing engines (SPEs) provide *window operators*, which slice unbounded streams into bounded subsets suitable for stateful operations such as aggregation.

Over the years, researchers have proposed numerous window operator designs to meet diverse application requirements. Abadi et al. introduced time-based and tuple-based sliding windows, which became de facto standards [11], though their semantics depend on system design choices [12]. Later, data-driven windows emerged [13]. For example, Akidau et al. [14] proposed session and landmark windows for use cases with unknown durations, while

✉ Ahmed Awad
ahmed.awad@buid.ac.ae

¹ Engineering & IT, The British University in Dubai, Dubai, UAE

² Cairo University, Giza, Egypt

Grossniklaus et al. [15] introduced Frames, which divide input streams based on aggregate functions. In all cases, event-time processing is critical: a data element has both an *event time* (when it is generated at the source) and a *processing time* (when the SPE observes it). Two elements may arrive in the reverse of their event-time order, so the one generated earlier is observed later. Because event-time progress is external to the SPE, mechanisms such as watermarks [14, 16] are employed. A watermark indicates that no further data with earlier event times are expected, and any element with a timestamp earlier than the watermark is considered *late*. Correctly handling late events is thus a fundamental challenge for SPEs.

Importantly, major SPEs differ in the way they implement window operators, and even a simple query exposes it. Consider a stream with event-time timestamps 1, 1, 2, 2, 5, 13, 14, 15, 19, 23, 26 sliced by a sliding time window of size 10 and slide 2, where each window counts its elements. When the elements arrive in timestamp order (Fig. 1(a)), Kafka Streams [17] reports each window incrementally as elements arrive, whereas Flink [18] and Storm [19] emit a single result per window; the engines nonetheless disagree on the counts of windows they share, for instance at the window boundaries. When the same elements arrive out-of-order, e.g., as 13, 15, 26, 23, 1, 2, 5, 2, 1, 14, 19 (Fig. 1(b)), the divergence widens: Kafka Streams recomputes and re-reports the affected windows as late elements arrive; Flink issues repeated reports, and Storm drops late elements and may even leave a window under-counted. The colours and markers in Fig. 1 flag these phenomena, which we analyse in Sects. 3 and 4. These inconsistencies underscore the lack of uniform semantics across SPEs, complicating application portability, correctness reasoning, and software maintenance.

Several models exist that formally describe window operators [11, 20–22]. Akidau et al. [14] further advanced this line of work by introducing the *Dataflow model*, which explicitly distinguishes between event-time and processing-time, incorporates watermarks as progress indicators, and defines triggers to refine results when late data arrive. Despite these advances, existing models still fall short of capturing the full richness of newer window types (e.g., data-driven windows) and do not provide a unifying abstraction for the diverse ways in which contemporary SPEs handle out-of-order data [12]. This gap is particularly problematic for software engineers who require consistent abstractions to ensure correctness and portability across platforms.

In this paper, we address this gap with a unified semantic framework centred on a window-operator lifecycle model for window operators under out-of-order arrivals, and we validate the model through an empirical study of five stream processors: Apache Beam, Apache Flink, Spark Structured

Streaming, Kafka Streams, and Apache Storm. We present an architectural view that distinguishes *single-buffer* and *multi-buffer* maintenance, define the phases of the window lifecycle, and formalise reporting mechanisms that govern (re)computation when late events occur. Using controlled input traces for sliding and session windows, we experimentally characterise each system's event-acceptance predicates and reporting behaviour, exposing concrete inconsistencies across engines. The combined formalisation and evidence yield principled guidelines for engineering dependable, portable, and verifiable stream-processing applications.

The remainder of this paper is organised as follows. Section 2 introduces preliminaries. Section 3 presents our proposed model. Section 4 evaluates five large-scale SPEs in terms of lifecycle support. Related work is discussed in Section 5, and conclusions are drawn in Section 6.

2 Preliminaries

2.1 Data streams

First of all, it is crucial to understand how SPEs represent the time domain. Two notions of time are common in stream processing, event time, i.e., the instant when the data was generated from the source, and processing time, i.e., the instant when the data enters the SPE [14, 18]. Both notions are monotonic, i.e., they progress in a non-decreasing manner. From now on, we will refer to these notions of time as $\mathcal{T}_e$ and $\mathcal{T}_p$ respectively.

The progress of processing time is controlled by the SPE while the progress of event time is controlled by the source. If the SPE is answering continuous queries that rely on the $\mathcal{T}_p$ a total order on the incoming data is ensured. On the other hand, when the continuous query relies on the $\mathcal{T}_e$ there are no guarantees that data will arrive in the same order they are produced, i.e., out-of-order arrivals are possible.

It is now convenient to define what data streams are.

Definition 1 (Stream) A stream $\mathcal{S}$ is a countably infinite sequence of elements. A stream element $s \in \mathcal{S}$ is a tuple $\langle d, t_e, t_p \rangle$ where, d is the payload, t_e is the event time timestamp¹ ($t_e \in \mathcal{T}_e$), t_p is the processing time timestamp assigned by the SPE ($t_p \in \mathcal{T}_p$). Notably, t_p defines a total order on stream elements whereas t_e define a partial order as two or more stream element can have the same event time value.

In general, stream processing pipelines are formed from a chain of operators forming a directed acyclic graph (DAG).

¹ t_e is part of *data* but it is represented explicitly to ease further discussion

SPE	Results on the form (window start, window end, count)
Kafka Streams	<u>(0,10,1)</u> , <u>(0,10,2)</u> , <u>(0,10,3)</u> , (2,12,1), <u>(0,10,4)</u> , (2,12,2), <u>(0,10,5)</u> , (2,12,3) , (4,14,1), (4,14,2) , (6,16,1), (8,18,1), (10,20,1), (12,22,1), (6,16,2), (8,18,2), (10,20,2), (12,22,2), (14,24,1), (6,16,3), (8,18,3), (10,20,3), (12,22,3), (14,24,2), (10,20,4), (12,22,4), (14,24,3), (16,26,1), (18,28,1), (14,24,4), (16,26,2), (18,28,2), (20,30,1), (22,32,1), (18,28,3), (20,30,2), (22,32,2), (24,34,1), (26,36,1)
Flink	..., (0,10,5), (2,12,3) , (4,14,2) , (6,16,3), (8,18,3), (10,20,4), (12,22,4), (14,24,4), (16,26,2), (18,28,3), (20,30,2), (22,32,2), (24,34,1), (26,36,1)
Storm	..., (0,10,5), (2,12,1) , (4,14,3) , (6,16,3), (8,18,3), (10,20,4), (12,22,4), (14,24,3), (16,26,3), (18,28,3), (20,30,2), (22,32,2), (24,34,1)

(a) In order arrival. **Bold**: a window that every engine produces but with a different count (the engines include the window boundaries differently), e.g. (2, 12) and (4, 14); blue, underlined: Kafka Streams re-reports the same window as each new element arrives; orange: a window that Kafka Streams and Flink report but Storm does not; . . . : earlier windows with negative start omitted (Kafka Streams starts its windows at 0).

SPE	Results on the form (window start, window end, count)
Kafka Streams	(4,14,1), (6,16,1), (8,18,1), (10,20,1), (12,22,1), (6,16,2), (8,18,2), (10,20,2), (12,22,2), (14,24,1), (18,28,1), (20,30,1), (22,32,1), (24,34,1), (26,36,1) , (14,24,2), (16,26,1), (18,28,2), (20,30,2), (22,32,2), <u>(0,10,1)</u> , <u>(0,10,2)</u> , (2,12,1), <u>(0,10,3)</u> , (2,12,2), (4,14,2) , <u>(0,10,4)</u> , (2,12,3), <u>(0,10,5)⁺</u> , (6,16,3), (8,18,3), (10,20,3), (12,22,3), (14,24,3), (10,20,4) , (12,22,4), (14,24,4), (16,26,2), (18,28,3)
Flink	(4,14,1), (6,16,2), (8,18,2), (10,20,2), (2,12,1), (4,14,2) , (2,12,2), <i>(12,22,2)</i> , <i>(12,22,3)</i> , (10,20,3), (8,18,3), (6,16,3), <i>(12,22,4)</i> , (10,20,4) , (14,24,4), (16,26,2), (18,28,3), (20,30,2), (22,32,2), (24,34,1), (26,36,1)
Storm	(4,14,1) , (6,16,2), (8,18,2), (10,20,2) , (12,22,2), (14,24,1)[†] , (16,26,1)[†] , (18,28,2), (20,30,2), (22,32,2), (24,34,1)

(b) Out of order arrival. **Bold**: a window that every engine produces but with a different count; blue, underlined: Kafka Streams re-reports the same window as late elements arrive, and ⁺ marks a window that only Kafka Streams produces; *green, italic*: Flink re-reports the same window when late elements arrive; orange: a window that Kafka Streams and Flink report but Storm does not; **red[†]**: a Storm window whose count no fixed window can justify, an element counted in some of the windows that contain it but not in others

Fig. 1 Results of sliding time window that count window contents

The operators are either stateless, like `map` or `filter`, or stateful like `aggregate` or `join`. Stateful operators for stream processing are basically related to window operators. A window operator is responsible for enforcing the logic of slicing the infinite stream into finite chunks according to some user-defined logic. Obviously, stateless operators are not affected by the out-of-order arrival of the data. On the other hand, window operators results may vary w.r.t out-of-order-arrival and the employed handling policy.

2.2 Progress indicator for event time processing

As indicated above, when process data streams based on their time of arrival to the SPE, a.k.a processing time, the SPE can use its internal clock to assign a timestamp to the data and thus controls the time progress. On the other hand, when processing data streams based on event-time, SPEs require an external progress indicator. The role of the progress indicator is to determine which window instances can be marked as *completed* and the operator can apply the user-defined computation on their contents. A progress indicator, thus, has to be monotonic. In general, the progress indicator has to relate to a timestamp that is part of the data elements. Progress indicators can be supplied from an external source (e.g. the case of punctuation [23]) or generated internally by the SPE based on the event timestamp of stream tuples [14, 24]. Without loss of generality, we refer to each occurrence of a progress as a watermark.

Out-of-order arrival of stream tuples is not an issue as long as their event time is greater than the most recent watermark. Whenever a tuple arrives at SPE where its event time is less than or equal to the most recent watermark, it is considered as late arrival. Only then, depending on architecture of the window operator (we discuss in Sect. 3), the response of the operator to late arrivals will vary.

2.3 Out-of-order and late arrival

In practice, data may arrive out-of-order (OOO), among other reasons, because of network delays. For instance, two elements e_1 and e_2 where $e_1.t_e \leq e_2.t_e$ are said to arrive out-of-order if $e_2.t_p < e_1.t_p$. We define the *delay* of an element e as the divergence between its processing and event time, $\delta(e) = e.t_p - e.t_e \geq 0$, the larger $\delta(e)$, the later e is observed relative to when it was generated. Unfortunately, most stream processing use cases require exploiting event time. Therefore, SPEs need to support one or more OOO techniques. Generally, these techniques can be classified as buffer-based, punctuation-based, or speculation-based [25].

Buffer-based techniques *postpone* the processing of stream elements for a slack period T . After that, they sort the elements and pass them to the window operator for

processing. Punctuation-based techniques depend on a special stream element that indicates the progress. The difference between buffer-based and punctuation-based techniques is that the buffering period is fixed in the former, whereas it depends on the data in the latter. In both cases, elements whose delay exceeds the buffering period T (i.e., $\delta(e) > T$) are ignored.

Speculation-based out-of-order processing is optimistic. That is, it assumes no out-of-order events will occur and that the window operator can emit the window result upon window completion. If late elements arrive, the emitted result is invalidated, and an updated result that accounts for the late element is emitted.

3 The proposed window operator model

Available models, e.g. [11, 21, 22, 26], that formally describe the windowing behaviour fall short of covering the recent enrichment in both new window types (e.g. data-driven windows) as well as the handling of out-of-order data [12]. As shown in Sect. 1, the behaviour of SPEs in response to data arriving out-of-order or late varies significantly. We argue that this variation relates to two aspects: 1) the ability to recompute a window function upon the arrival of an element that belongs to a *completed* window, and 2) how the updated result is communicated to downstream operators in the processing pipeline. We further argue that this variation stems from the window operator's architecture, namely whether it maintains stream elements in a *single-buffer* or in separate buffers (*multi-buffer*). In the former case, elements remain buffered by the operator; upon a trigger (discussed later), they are sorted by their $e.t_e$, grouped according to the window specification logic, and the results are computed and emitted, after which some elements are *evicted* to make room for new arrivals. Because this sorting and eviction proceed as the (monotonic) watermark advances, a single-buffer operator reports each window once and in non-decreasing window order and never revisits an already-reported window, as Storm does in Fig. 1, whose output is globally ordered by start time under both arrival scenarios. Multi-buffer operators, by contrast, assign each element upon its arrival to one or more window instances (buffers) based on the window specification logic, and, upon a trigger, each eligible instance immediately reports its content; results are therefore emitted in the order windows are *updated* during processing, i.e. in arrival order rather than in window-start order. This is visible for Kafka Streams and Flink in Fig. 1: in Fig. 1(a) Kafka Streams already interleaves re-reports of (0, 10) after (2, 12), and in Fig. 1(b) both engines emit an earlier-start window after a later one (e.g. Flink reports (2, 12) after (10, 20)). The reordering is

intrinsic to multi-buffer maintenance rather than a consequence of out-of-order input.

Before we characterize single and multi-buffer operators and reporting policies in sects. 3.2, 3.3, and 3.4, we describe what we mean by a window specification and a window instance in Sect. 3.1 together with a running example.

3.1 Window specification and instances

Windowing is a core abstraction for developing stream processing applications [27]. In particular, windows turn streaming data into a snapshot of static data suitable for computation, known as a window function. In this paper, we distinguish between *window specification* and *window instance*.

Definition 2 (*Window Specification*) A window Specification refers to (i) a set of parameters w_o , namely window type, that determine how to divide a stream S into finite portions (window instances), (ii) a function f to be evaluated on these finite portions, (iii) a default trigger that can be derived from w_o , (iv) an optional evictor, (v) an optional retention period p to keep window instances, and (vi) a reporting policy for relating results of multiple window function invocations for the same window instance and sending them downstream.

Based on a window specification, one or more window instances can be generated when the specification is passed to the window operator in a processing DAG.

Definition 3 (*Window instance*) A window instance w over stream S is a finite subset of S ; *start* and *end* are the boundaries of w . Given a window specification W , we call $\mathcal{W}$ the set of all window instances derived from W . We define

$\text{content} : \mathcal{W} \rightarrow 2^S$ as a function that returns the content of stream tuples of a window instance.

Example window specifications are sliding time or sliding tuple windows [11]. This specification has two parameters ω the window width and β the window slide. These generate window instances of fixed width ω and trigger the window function every β units. Whether the domains of ω and β refer to the time domain or to integers, this controls whether it is a sliding time or tuple window. If $\beta < \omega$, window instances will overlap and one or more stream elements can belong to more than one window instance. When $\omega = \beta$, this is a tumbling window specification where windows do not overlap, and each stream element is assigned to exactly one window. Finally, if $\omega > \beta$, this is a hopping window specification and a stream element may not be assigned to any window [28]. Sliding window specifications are data agnostic. That is, the slicing of the stream does not depend on the data payload d of a stream element (Definition 1). Instead, it depends on timestamps or tuple numbers. Other data-agnostic window specifications are Landmark windows [28]. In this case, the specification fixes one boundary and lets the other flow freely.

Another category is data-driven specifications. Examples are session windows [14], threshold, delta, aggregate and boundary frames [13, 15]. These specifications usually refer to a condition on either a tuple's timestamp (as in session windows) or one or more attributes of the tuple's payload d (e.g., threshold windows). Assuming in-order arrival, contiguous tuples that satisfy the condition form a window instance.

Figure 2 compares the different window specifications and visualises sample instances thereof. The top three are the common data-agnostic window specifications. They have a fixed width as per their specification. The lower

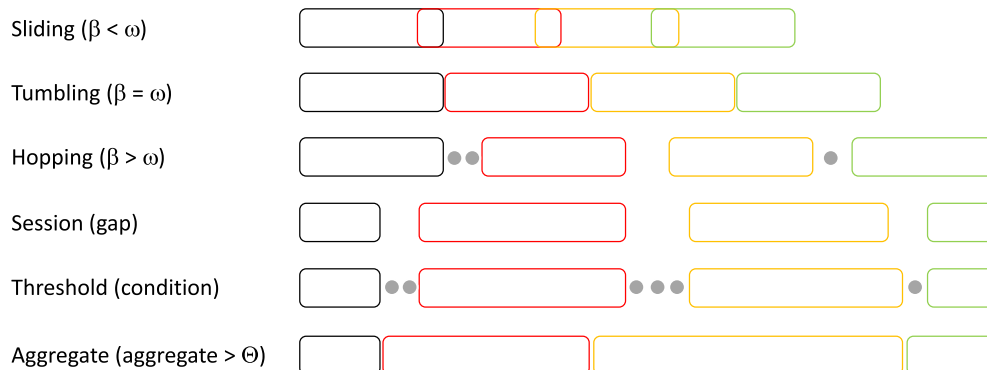


Fig. 2 Window specifications and sample instances over a single stream (time flows left to right). Each rounded box is one window instance; consecutive instances are drawn in alternating colours *only* to separate them. Grey dots mark stream elements assigned to no instance. The top three specifications are *data-agnostic*: their boundaries follow from the window parameters, independently of the payload, sliding ($\beta < \omega$

) instances overlap, tumbling ($\beta = \omega$) instances partition the stream, and hopping ($\beta > \omega$) instances leave gaps. The bottom three are *data-driven*: their boundaries follow from the data, session (an inactivity gap), threshold (a predicate on the payload), and aggregate (an accumulated measure exceeding Θ)

three examples represent data-driven specifications. As can be noticed, only sliding window specifications have overlapping instances. All other specifications have disjoint instances. This means that, considering in-order arrival, at any time there is at most one active window instance. We will discuss later how out-of-order arrival can disrupt that tuple-to-window assignment.

We can draw an analogy between fixed- and variable-width specifications. For example, aggregate and tumbling window instances fully partition the stream. That means that once a window instance ends, another starts immediately. Hopping, session and threshold windows can have temporal gaps between their instances. Moreover, some stream elements may not be assigned to any window instances.

To assign tuples to window instances of data-agnostic window specifications, the window operator depends solely on the specification parameters and the newly arriving tuples to figure out to which window instance(s) to assign the tuple. In contrast, for data-driven window instances, the operator must examine the available window instances to determine where to assign the tuple.

Consider the example shown in Fig. 3. We consider processing the eight elements at the top of the figure. The subscript represents the processing time, whereas the value after the = sign is the event time of the element, e.g. $e_2.t_e = 3$ and $e_2.t_p = 2$. Our pipeline counts elements in a sliding time window of size 5 milliseconds, with a slide of 2 milliseconds. Thus, every 2 milliseconds, we report the count of elements from the past 5 milliseconds.

Our window operator model distinguishes between single-buffer and multi-buffer window instance maintenance. We describe primitives for handling tuple arrivals, triggering, and policies for handling out-of-order and late tuple arrivals. We support our discussions with examples on sliding windows (as an example of data-agnostic windows) and threshold windows (as an example of data-driven windows)

3.2 Single-buffer window operator (Lazy)

A single-buffer window operator remains *lazy* with respect to assigning stream tuples to window instances. When a query containing a window operator is deployed, a single

buffer is created. The operator first admits incoming elements and updates progress (e.g., the watermark).

The operator exposes only two external events: *add element* and an abstract *trigger*. Internally, it has two transient micro-states, EVICTING and COMPUTING, which run to completion and immediately return control to BUFFERING. The ordering between these micro-steps is policy-driven (engine/configuration dependent):

- On *add element*, the operator may (i) execute EVICTING followed by COMPUTING, (ii) execute EVICTING only, or (iii) remain in BUFFERING (admit-only). In all cases, after the micro-steps complete, the just-arrived element is admitted into the buffer if it is not late (otherwise it is dropped or side-output).
- On *trigger*, the operator executes either EVICTING → COMPUTING, COMPUTING → EVICTING, EVICTING only, or COMPUTING only. The *trigger* is intentionally abstract (time-, count-, or delta-based); the concrete engine chooses the order.

Window instances are determined *at compute time* by applying the window specification (Definition 2) to the currently buffered (non-stale) tuples; for each window instance, the window function f is invoked and results are emitted. EVICTING removes stale tuples (e.g., those older than the progress bound) either before or after computation, according to the selected order. After the micro-steps finish, the operator returns to BUFFERING and continues to accept incoming tuples until the next event.

Figure 4 depicts this lifecycle with BUFFERING as the sole stable state, and EVICTING/COMPUTING as transient steps chained by run-to-completion transitions.

We define the following basic operations for a single-buffer window operator.

- $\text{addElement}(s : \mathcal{S}, t : \mathcal{T}_e)$: The first parameter is the newly arriving stream tuple s ; the second parameter t is the minimum timestamp that can be accepted, not necessarily the watermark. Basically, the tuple is added to the buffer if and only if $s.t_e > t$.

Fig. 3 Handling of late elements in single versus multiple buffering based SPEs

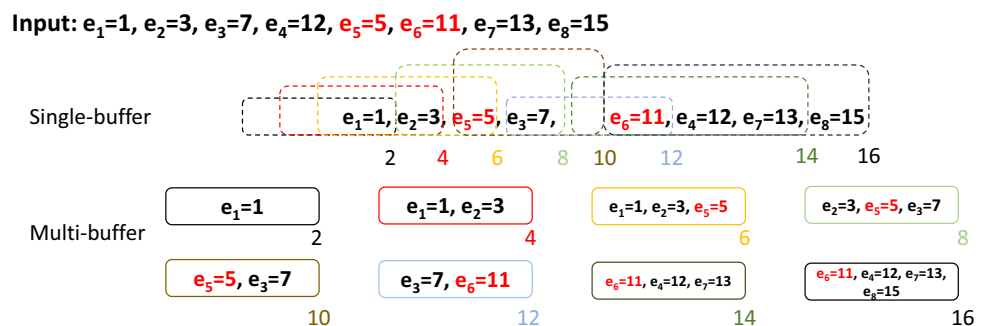
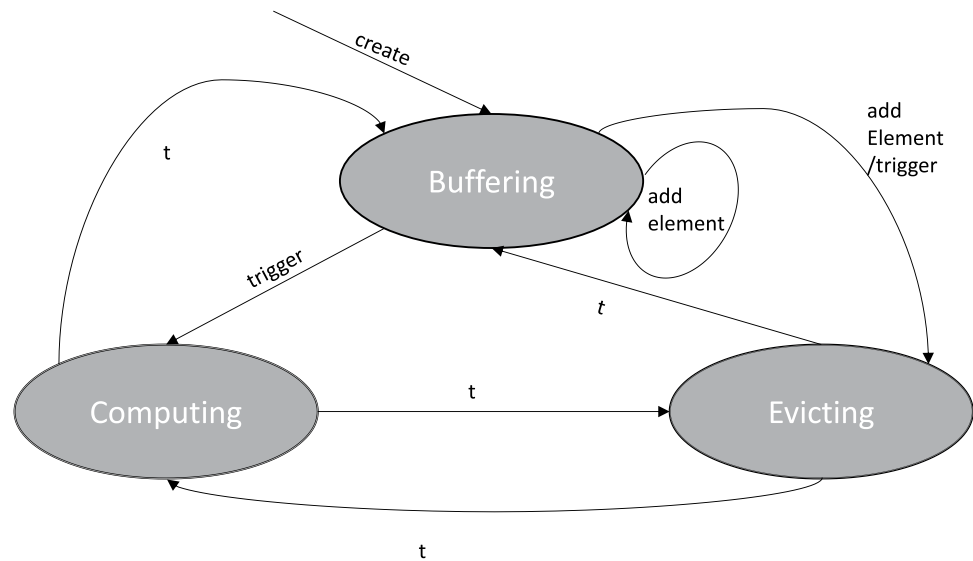


Fig. 4 Lifecycle of the single-buffer window instance



- $\text{trigger}(t : \text{Trigger})$: a trigger t makes the window operator start applying the given window specification on the set of buffered stream tuples. A trigger may occur due to the arrival of a watermark, may be a periodic process within the SPE or may be caused by reaching a specified number of elements in the buffer or any other reason [11].
- $\text{apply}(wo : \text{Window})$: applies the windowing logic. That is, it defines how the buffered tuples should be grouped as per the specified windowing logic. In some cases, the tuples in the buffer must be sorted by timestamp first (an example follows shortly). At this stage, the operator creates window instances. The window function $wo.f$ is then applied to each instance, and results are reported to the downstream operator.
- $\text{evict}(e : \text{Evictor})$: the evictor e is the criterion to be used to determine the expired tuples that have to be removed from the buffer [11].

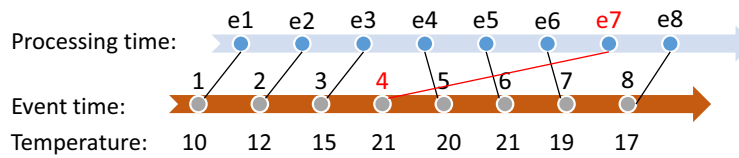
Example 1 (Sliding window) Let us get back to the sequence of input tuples given in Fig. 3. We also consider a sliding time window with $\omega = 5$ and $\beta = 2$. Without loss of generality, assume a triggering watermark occurred at processing time 3 with value 7. At that time, the buffer contains $e_1 = 1, e_2 = 3, e_3 = 7$. Under the sliding specification, the operator reports a window every 2 milliseconds that includes tuples whose timestamps are at most 5 milliseconds old [11]. Thus, the operator creates window instances $[-3, 2) = \{e_1 = 1\}$, $[-1, 4) = \{e_1 = 1, e_2 = 3\}$ and $[1, 6) = \{e_1 = 1, e_2 = 3, e_3 = 7\}$. The operator cannot proceed to the next window instance $[3, 8)$ as its upper bound is greater than 7. At that stage, the minimum acceptable timestamp is $t = 3$. This is because it is the lower bound of the next window instance to be computed with the next trigger. After computing for this trigger, the operator evicts all

elements with timestamps less than 3 from the buffer. This results in dropping $e_1 = 1$. The tuple $e_4 = 12$ is added to the buffer as $12 \geq 3$. The same applies to tuple $e_5 = 5$, which is accepted even though it arrived late. However, it should have been added to window instance $[1, 6)$ if it had arrived before the trigger. In this case, the operator cannot recompute that window instance because it has already advanced past it and evicted older elements belonging to it.

Example 1 has shown the limitations of single-buffer operators with the common sliding window specification w.r.t. handling late elements. One could argue that eviction criteria can be overwritten to retain tuples longer. While this might seem like a solution, it would require the operator to reset window progress each time it receives a trigger, since it cannot distinguish which window instances need recomputation. This would send several identical results downstream (since window instances' contents did not change). An alternative is to left-shift the watermark value by some lag l on a time scale. In this case, it is possible to account for late elements as long as their delay does not exceed l . However, this incurs higher latency, as all window instances that end after $w - l$ but before w must wait for a future trigger to apply their window functions and emit results.

The tuple-to-window instance assignment would require one pass over the tuples. In Example 1, tuples need not be sorted. In other cases, tuples must be sorted by timestamp before applying the windowing logic, which adds to processing latency. We describe that in Example 2.

Example 2 (Threshold window) Consider a threshold window specification [15]. The specification refers to a condition on an attribute in the tuple $s.d$ (Definition 1). Window instances consist of sequences of tuples that satisfy the condition. A tuple that does not satisfy the condition marks the



Trigger (watermark=6)

- (1) Sorted buffer: $e_1=10, e_2=12, e_3=15, e_4=20, e_5=21$
- (2) Window instances: $e_1=10, e_2=12, e_3=15, e_4=20, e_5=21$
- (3) All tuples are evicted

Trigger (watermark=8)

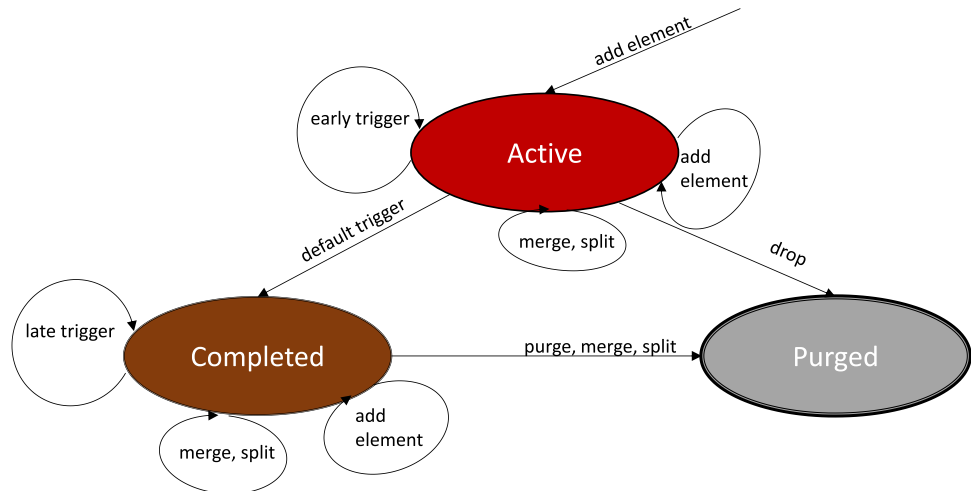
- (1) Sorted buffer: $e_6=19, e_8=17$
- (2) Window instances: $e_6=19, e_8=17$
- (3) All tuples are evicted

Trigger (watermark=8)

- (1) Sorted buffer: $e_1=10, e_2=12, e_3=15, e_7=21, e_4=20, e_5=21, e_6=19, e_8=17$
- (2) Window instances: $e_1=10, e_2=12, e_3=15, e_7=21, e_4=20, e_5=21, e_6=19, e_8=17$
- (3) All tuples are evicted

Fig. 5 Handling of a threshold window by a single buffer operator

Fig. 6 Lifecycle of the multi-buffer window instance



end of the current window instance, if any. For our example, assume a sequence of temperature reading events as shown at the top of Fig. 5. The threshold window condition is that $temperature \leq 20$. The operator shall reset the buffer (i.e. evict all the tuples) after handling a trigger. At the bottom left of the figure, we assume that two triggers occurred with watermarks $w = 6$ and $w = 8$, respectively. On the bottom right, we assume that only one trigger occurred with $w = 8$. In the former case, only tuples e_1 – e_5 are received. Tuples are sorted by timestamp, and one threshold window instance is found $[1, 6]$ ² and reported. The minimum acceptable timestamp is now $t = 7$. The buffer is emptied. As the operator receives the next tuples, it rejects e_7 because its timestamp 4 is less than t . At the next watermark $w = 8$, the same

procedure is repeated and another window instance $[7, 9]$ is reported. In the other case, when only one trigger occurs with $w = 8$, all tuples have been added to the buffer and sorted (bottom right of Fig. 5). After sorting the tuples, we identify three window instances: $[1, 4]$, $[5, 6]$, and $[7, 9]$.

3.3 Multi-buffer window operator (Eager)

A multi-buffer window operator is eager w.r.t. tuple-to-window instance assignment. Window instances are maintained explicitly and are first-class citizens. Moreover, each window instance has a lifecycle (Fig. 6) according to which the operational status of the window instance determines how it is possible to respond to an out-of-order/late tuple arrival. In the beginning, a window instance is in the operational status *active* upon its creation by the window operator. A window

² Note that the subscript refers to processing time.

instance is created when it gets assigned its first element. When a trigger occurs, e.g., a watermark, indicating window instance closure, the window instance content can be passed to the window function for computation. The window instance's operational state is updated to *completed*. An out-of-order arrival while a window instance is *active* may cause it to split or merge with another instance. If the arrival occurs after the instance has been *completed*, the same split or merge may occur, but the window function is re-triggered, and an updated result is sent downstream. According to the retention period defined in the window specification (Definition 2), completed window instances will be kept by the window operator. Afterwards, the window instance is purged. At that stage, the operator drops any late arrivals.

Definition 4 (*Window instance operational state*) Let w_0 be a window specification. Let $\mathcal{W}$ be the set of window instances in the SPE according to w_0 . We define the function operational state $opState : \mathcal{W} \rightarrow \{active, completed, purged\}$ that returns the operational status of a window instance $w \in \mathcal{W}$.

We define the following basic operations for a multi-buffer window operator:

- $addElement(s : S, w_0 : Window, t : \mathcal{T}_e)$: assigns the stream element to one or more window instances. This assignment is defined by the window specification w_0 given to the window operator. Because of this assignment, the window operator may create a window instance when it receives the first element belonging to that window. Moreover, the window operator has to keep information about purged windows or a progress metric t where s is considered for assignment if $s.t_e > t$.
- $trigger(t : Trigger)$: a trigger t allows the application of the window function $w_0.f$ on the content of the window instances. Unlike the single-buffer operator, the content of each window instance is already known by the trigger time, and the operator does not need to sort or apply the windowing logic. In addition, a multi-buffer window operator supports a richer triggering model. That is, there could optionally be an early trigger (Fig. 6). In this case, the window instance content is passed to the window function even before the window completes. Additionally, late arrivals on completed windows send updates (corrections) of the window function computations downstream.
- $merge(w_1 : \mathcal{W}, w_2 : \mathcal{W})$: After adding some stream element s to a window instance w_1 , it might be the case that the window boundaries of w_1 overlap with another window instance w_2 whereas the window specification w_0 does not allow for overlapping windows.

In this case, the window instances have to be merged, resulting in a new window instance w_3 . In this case $content(w_3) = content(w_1) \cup content(w_2)$. w_3 is active if either w_1 or w_2 is active. Otherwise, it is set to completed. If either w_1 or w_2 is completed, an update should be sent downstream.

- $split(w : \mathcal{W})$: Window instance w may be split into two window instances w_1 and w_2 due to adding an element to the window, which breaks the assignment logic in w_0 . In this case, the content of w is divided between w_1 and w_2 based on the logic in w_0 . Both w_1 and w_2 inherit the operational status of w . If w was completed, the split triggers a late (re)computation: the previously reported result for w is superseded and updated results for w_1 and w_2 are sent downstream (a $1 : 2$, and in general $1 : M$, update; see Sect. 3.4).
- $purge(w : \mathcal{W}, p : Int, t : \mathcal{T}_e)$: a purge operation removes the content of window instance w from the memory of the window operator. It, however, may keep meta-data about the window instances. The retention period p and the progress indicator t control purging. Whenever $w.end + p \leq t$, the window is purged.

Example 3 (*Sliding window*) Here, we discuss how a multi-buffer operator handles a sliding time-window specification. Referring to Fig. 3, we have a sliding window specification with $\omega = 5$ and $\beta = 2$. We also define the retention period as 10. When receiving the first element $e_1 = 1$, the operator will create three window instances $w_1 = [-3, 2)$, $w_2 = [-1, 4)$ and $w_3 = [1, 6)$ and assign a copy of $e_1 = 1$ to each of them. When it receives $e_2 = 3$, it assigns it to both w_2 and w_3 . Additionally, the window operator creates a new window instance $w_4 = [3, 8)$ and assigns a copy of $e_2 = 3$ to it. Again, by receiving $e_3 = 7$, it gets assigned to w_4 , and the window operator creates two new window instances $w_5 = [5, 10)$ and $w_6 = [7, 12)$, where each gets a copy of $e_3 = 7$. Similar to Example 1, assume that a triggering watermark occurred at processing time 3 with timestamp 7. At that time, all window instances are active. Window instances w_1, w_2 , and w_3 can change status to completed because their end timestamps are before the watermark. Once completed, the window function can process their content, and the results are sent downstream. When $e_4 = 12$ arrives, it gets assigned to new window instances $w_7 = [9, 14)$ and $w_8 = [11, 16)$. Next, element $e_5 = 5$ arrives late. It gets assigned to window instances w_3, w_4 and w_5 . w_3 is already completed. Thus, assigning $e_5 = 5$ to it will cause it to recompute (late trigger) and send the update downstream. Meanwhile, window instances w_4 and w_5 are still active. Element $e_6 = 11$ arrives out of order but not late, because the latest watermark is 6. So, it gets assigned to the respective window instances. Now, assume that another triggering watermark occurs with

timestamp 12. In this case, window instances w_4, w_5 and w_6 will be completed, and the results of their window functions can be sent downstream. Moreover, window instance w_1 will be purged, as its retention has now expired. Now, the minimum acceptable timestamp is 2, which is the end of w_1 .

Example 4 (Threshold window) Let us consider how the threshold window specification given in Example 2 will be handled by a multi-buffer window operator. Considering the sequence of arrival shown at the top of Fig. 7³, a multi-buffer operator will group stream elements that are temporally contiguous and that satisfy the condition $temperature \leq 20$. We also follow the same settings for watermark triggers. In the bottom left of Fig. 7, we show the window instances maintained by the operator at watermarks of 6 and 8. At watermark 6, we have one window instance [1, 6), where the member elements are shown in the figure. Note that $e_5 = 21$ does not belong to the window as it does not satisfy the condition. At the watermark trigger time, this window instance is marked as completed, and the result of applying the window function is reported downstream. Next, stream element $e_6 = 19$ is received and is assigned to a new window instance [7, 8). This is shown as the status before receiving $e_7 = 21$. By receiving $e_7 = 21$, whose timestamp is 4, the operator inserts this element into window instance [1, 6). This will cause the window instance to split, as the element does not satisfy the threshold condition. This results in a total of three window instances [1, 4), [5, 6) and [7, 8) as shown with the status “After Receiving $e_7 = 21$ ”. As the

window instance [1, 6) was already completed before splitting, the arrival of $e_7 = 21$ causes a late trigger, where the results of applying window functions over [1, 4) and [5, 6) are sent downstream, invalidating the old results sent earlier for [1, 6). On the bottom right of Fig. 7, the case when only one watermark trigger with timestamp 8 is shown. The main difference from the former case is that after receiving $e_7 = 21$, we have three window instances; however, all of them are still *active*. So, the split will not cause any updates to be sent downstream. In both cases, the downstream operator receives the same results when the watermark triggers at timestamp 8.

3.4 Report

As we illustrated in Sect. 3.3, late arrivals of stream elements to a window instance can cause a series of operations that end up recomputing the window function. In this section, we discuss the Reporting step that follows the (re) computation of the window function.

Definition 5 (Report) The Report of a window operator w.r.t. f propagates the results downstream according to a reporting policy.

In Definition 5, we listed two central building blocks for reporting: the result produced by evaluating the window function f and the reporting policy. The evaluation of the

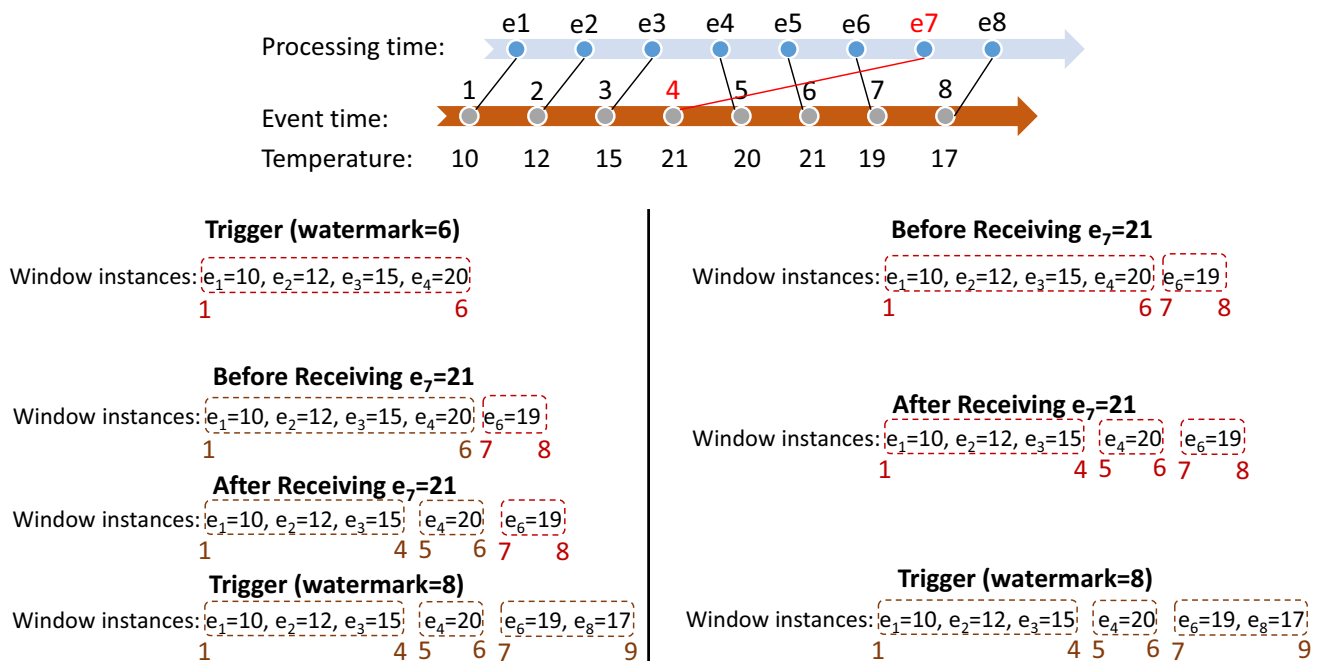


Fig. 7 Handling of a threshold window by a multi-buffer operator

³ We have repeated the same input as shown in Fig. 5 for convenience

window function is achieved by applying the logic of f to the content of a window instance (Definition 3).

To define reporting policies, we need the notion of an instantaneous result.

Definition 6 (*Instantaneous Result*) An instantaneous result is a tuple $\langle rid, r, t_e, t_p \rangle$ where r is the computed result, i.e. $r = f(w)$ where $w \in \mathcal{W}$ is one of the window instances eligible to trigger at processing time t_p and event time t_e , and rid is a unique identifier of the result. The combination $\langle w.start, w.end \rangle$ uniquely identifies a result tuple. We call this identifier rid . Notably, the result structure resembles that of the window function. The instantaneous result inherits the timestamps of the trigger that produced it: t_p is the processing time at which the trigger fires, and t_e is the event time carried by that trigger (e.g., the watermark value). In general, $t_e \neq t_p$; a late trigger, for instance, fires at a large t_p while carrying the earlier t_e of the progress indicator.

Last but not least, complementary to Definition 6, we define Instantaneous Retraction as follows.

Definition 7 (*Instantaneous Retraction*) An instantaneous retraction is an instantaneous result where the result data r is null.

Downstream operators receive the produced results. Three reporting policies can be defined around the notions of rid and the guarantees downstream operators require.

Definition 8 (*Stateless Reporting*) Stateless Reporting policy implies that the SPE does not keep track of the emitted results over time, i.e., the report does not take into account the relation between instantaneous results.

This policy can be represented by the following function that calculates the result ID $rid: \mathcal{T}_p \rightarrow N+$. The rid depends on the processing time only and, thus, result IDs are always increasing.

Definition 9 (*Refinable Reporting*) The Refinable Reporting policy implies that instantaneous results can be refined. The SPE tracks emitted results and can emit updates when it receives new data.

This policy can be represented by the following function that calculates the result id $rid: \mathcal{W} \rightarrow N+$, where $\mathcal{W}$ is the set of window instances. The rid depends on a window instance; therefore, results deriving from the late update maintain the same rid and can be correctly interpreted downstream.

Definition 10 (*Retractable Reporting*) The Retractable Reporting policy implies that instantaneous results can be retracted upon receiving new data, e.g., late arrivals. The SPE tracks emitted results and, upon the reception of new data, emits atomically⁴ a retraction of the previous results and the new instantaneous result.

This policy can be represented by the following function that calculates the result id $rid: \mathcal{T}_p \times \mathcal{W} \rightarrow N+$. The rid depends on both processing time and a window instance. Therefore, result IDs are always increasing as old rid s are explicitly retracted.

Because a single-buffer window operator can compute a window instance only once, it supports only stateless reporting (Definition 8). Similarly, a multi-buffer window operator supports stateless reporting only if it purges the window instance immediately after its first trigger. However, a multi-buffer window operator can support the other reporting policies. Repeated reporting occurs in response to (late) arrivals of tuples assigned to the respective window instance.

Referring to Example 3, upon the arrival of late tuple $e_5 = 5$, it is assigned to window instances w_3 , w_4 , and w_5 . At this stage, w_3 has already been triggered once. Using a stateless reporting policy, the results of the first trigger, upon closure, and the second trigger, due to the late arrival of a tuple, look like $\langle rid = (1, 6), r = 2, t_e = 7, t_p = 3 \rangle$ at the watermark occurrence, and $\langle rid = (1, 6), r = 3, t_e = 7, t_p = 5 \rangle$, upon the arrival of the late tuple, respectively. As this a stateless reporting policy, downstream operator has to maintain its own state of results to reason about the change of the result. This is possible because the result has a unique identifier, $rid = (1, 6)$.

Following a refinable reporting policy (Definition 9, the results of triggering w_3 will be $\langle rid = (1, 6), r = 2, t_e = 7, t_p = 3 \rangle$ and $\langle rid = (1, 6), r = 1, t_e = 7, t_p = 5 \rangle$. Upon the reception of the refined result by the downstream operator, it shall add 1 to the older result sent earlier and thus, the new result is now 3. In the former case of stateless reporting, the downstream operator cannot decide whether the new value 3 in the second report is the new total value or an increment to the previously reported value.

Adopting a retractable reporting policy, the reporting for w_3 will be $\langle rid = (1, 6), r = 2, t_e = 7, t_p = 3 \rangle$ at the watermark trigger. Then, upon arrival of e_5 , the old result will be retracted as $\langle rid = (1, 6), r = null, t_e = 7, t_p = 5 \rangle$ and the new result will be $\langle rid = (1, 6), r = 3, t_e = 7, t_p = 5 \rangle$. Compared with the stateless reporting policy, the downstream operator can clearly see the new result of the computation. Moreover, compared to the refinable policy, the downstream

⁴ The event timestamp of the retraction and the event timestamp of the new instantaneous result are the same.

operator does not have to remember the latest result sent from the upstream window operator.

These policies are not merely hypothetical; the engines in Fig. 1 instantiate them. Storm, a single-buffer operator, reports each window instance exactly once and never revisits it; this is stateless reporting (Definition 8). Kafka Streams and Flink, both multi-buffer, instead re-report the *same* window instance (identified by $rid = (start, end)$) with an updated result as new or late elements are assigned to it: in Fig. 1(a) Kafka Streams reports (0, 10) successively as 1, 2, ..., 5, and in Fig. 1(b) both engines re-report a window such as (12, 22) after a late arrival. This repeated, window-keyed reporting is stateful rather than stateless (refinable or retractable, Definitions 9 and 10); whether a given engine refines, retracts, or replaces the earlier result, and under which configuration, is engine-specific, and Sect. 4 determines it for all five engines. The reporting policies above are therefore the ones realised by production systems, the earlier examples serving only to isolate each policy in a controlled setting.

Reporting for data-driven windows is more challenging, as window durations might change due to a split or merge. Consider Example 4. Upon the arrival of $e_7 = 21$, where $e_7.t_e = 4$, the window $[1, 6)$ which was triggered earlier will have to be split into two windows $[1, 4)$ and $[5, 6)$. Stateless reporting will generate $\langle rid = (1, 6), r = 4, t_e = 6, t_p = 10 \rangle$ at the first trigger and $\langle rid = (1, 4), r = 3, t_e = 6, t_p = 15 \rangle$, $\langle rid = (5, 6), r = 1, t_e = 6, t_p = 15 \rangle$ for the later trigger. In this setup, the downstream operator cannot correlate results by result ID. This is because the window boundaries have changed.

Following a refinable reporting policy, the first trigger will be similar to stateless reporting. However, for the second trigger, the refinement of the old result for the window $[1, 6)$ will be $\langle rid = (1, 6), r = -4, t_e = 6, t_p = 15 \rangle$. The result $r = -4$ indicates that the older result should be refined to 0. For the two new windows, the window operator reports $\langle rid = (1, 4), r = 3, t_e = 6, t_p = 15 \rangle$ and $\langle rid = (5, 6), r = 1, t_e = 6, t_p = 15 \rangle$. Moving to retractable reporting, the first report is similar to the two cases above. For the second reporting, the operator retracts the old result by sending out $\langle rid = (1, 6), r = null, t_e = 6, t_p = 15 \rangle$. Then it sends the new updates similar to the two new windows. With data-driven windows, we can notice that refinable and retractable reporting is more complex as there can be 1 : M relationship between the old and the new results, due to window split. $M : 1$ relationship is also possible in the case of window merging.

3.5 Discussion

In sects. 3.2 and 3.3, we showed how single-buffer and multi-buffer operators would handle window specifications, with examples of standard sliding window specification and another data-driven threshold window specification. In both cases, the limitations of the single-buffer operator are obvious w.r.t. handling late data. For a single-buffer operator to improve accuracy (less tuples to be dropped), watermarks (or other external progress indicators) must occur less often. However, this increases latency. One way to overcome this inherent inflexibility of a single-buffer operator is to spawn new instances of the pipeline (window operator) when a late arrival is observed [29]. However, this is a system- or application-level solution, rather than being an operator-level solution. Referring to a window specification in general (Definition 2), we can see that a single-buffer operator cannot retain window instances. That is, $p = 0$ as the operator constructs window instances on the fly and evicts outdated stream elements after computation is complete. A single-buffer operator supports stateless reporting only (Definition 8). This is because it can only trigger a computation on a window instance once.

A multi-buffer operator provides more flexibility in balancing accuracy and latency. First, a (re)computation can occur several times. This is because multiple triggers can apply to the same window instance. Besides the default trigger, e.g. due to a watermark, a computation can be triggered early, e.g. every n time units or tuples before window completion. Moreover, late arrivals to a completed window instance cause late triggers. Multiple triggering is possible as long as the operator retains window instances. The flexibility of the multi-buffer operator comes with a price. The price is that window specification (windowing logic) is applied eagerly on each arriving element. A multi-buffer operator can support any of the reporting policies discussed in Sect. 3.4, as it is aware of both processing time and window instances, in response to multiple triggers.

Referring to out-of-order handling techniques discussed in Sect. 2.3, we can see that a single-buffer operator naturally supports buffer-based OOO handling. It can also support punctuation-based OOO handling techniques, as it can still keep buffering elements until a punctuation is received. A single-buffer operator cannot support speculation-based techniques, as accuracy will suffer significantly. On the other hand, multi-buffer operators naturally support speculation-based techniques because they can recompute the window function over a window instance. They can also support buffer- and punctuation-based techniques.

Table 1 summarises the points discussed in this section. Downstream complexity relates to the window operator's reporting capabilities. Beyond stateless reporting,

Table 1 Comparison of single versus multi-buffer window maintenance

Window maintenance	Latency	Memory overhead	OOO handl.	Trigger	Retention	Report	Downstream Complexity
Single-buffer	High	Low	Buff., Punc.	Once	No	Stateless	Simple
Multi-buffer	Low	High	All	Multiple	Yes	All	Complex

Table 2 Window maintenance and window types supported by the evaluated systems

System	Window maintenance	Window types	Watermark progress
Storm (2.6.0)	Single-buffer	Sliding, tumbling	Implicit
Spark (3.5.1)	Multi-buffer	Sliding, tumbling, session	Implicit
Kafka (3.9.1)	Multi-buffer	Sliding, tumbling, session	N/A
Beam (2.68.0)	Both	Sliding, tumbling, session, custom	Explicit
Flink (1.18.1)	Both	Sliding, tumbling, session, custom	Explicit

the downstream operator needs to apply complex logic to resolve the update received from the upstream window operator. This must also be reflected in downstream operators through the end of the DAG.

4 Assessment of SPEs against our window lifecycle model

In this section, we evaluate the windowing operators of five widely used stream processing systems against our model. We selected them using three criteria that make the study both principled and inspectable. First, each system is *open-source*, so we can examine its public windowing API and, where necessary, its internal implementation, and thus *explain* the observed behaviour rather than treat the engine as a black box. Second, each processes data in *event time* using an *external progress indicator* (e.g., watermarks, either explicit or implicit) to decide window completeness and lateness, precisely the setting in which the single- versus multi-buffer maintenance and the reporting policies of Sect. 3 diverge. Third, each is a *distributed* stream processor for production workloads, not a single-node library or a complex-event-processing engine. Several open-source engines still satisfy all three; we study the five most widely adopted and architecturally distinct, and we discuss the remaining candidates in Sect. 4.8. The five we study already span the design space our model characterises: Storm exemplifies single-buffer maintenance, whereas Beam, Flink, Spark Structured Streaming, and Kafka Streams exemplify multi-buffer maintenance under different reporting policies. Concretely, we evaluated Apache Beam (executed on its default/direct runner), Apache Flink, Apache Spark

Structured Streaming, Kafka Streams, and Apache Storm core.

Our evaluation used both static analysis of the source code and experimental evaluation. The code for reproducing the experimental evaluation is available in our GitHub repo⁵.

Table 2 describes whether the window operator of the respective system is a single- or multi-buffer operator. We also list the window specifications each system supports and whether it allows explicit watermark definition.

To instantiate our model for each engine and compare results, we run a simple pipeline with the respective window specification and feed it a small amount of data. To guarantee a fair comparison, we have unified the watermark policy for Beam and Flink to update the watermark after each incoming record to the maximum timestamp seen. For Spark, as it works in micro-batches, we worked around this limitation by watching a folder where we drop a single-record file at a time to observe the output. We disabled any allowed lateness and received the window output after the window closure. The test data was deliberately tiny so we could easily analyse it, as our objective is to understand the lifecycle of elements and windows rather than measure operational metrics such as latency and throughput.

Concretely, we use 14 records: $r_1 = (13, A, 22)$, $r_2 = (15, A, 23)$, $r_3 = (26, A, 15)$, $r_4 = (27, W, 0)$, $r_5 = (23, A, 7)$, $r_6 = (1, A, 24)$, $r_7 = (2, A, 23)$, $r_8 = (5, A, 23)$, $r_9 = (30, W, 0)$, $r_{10} = (2, A, 22)$, $r_{11} = (1, A, 22)$, $r_{12} = (14, A, 28)$, $r_{13} = (500, W, 0)$, $r_{14} = (19, A, 30)$, where each record is a timestamp, key, and value, and rows with key W (value 0) are watermark-advancing tuples that carry no countable data and only advance event-time progress. Note that these records are *not* totally ordered by event time: two records share timestamp 1 (r_6, r_{11}) and two share timestamp 2 (r_7, r_{10}). We feed these records following three arrival regimes that let us separate distinct sources of cross-engine divergence:

- *Total order*: The records have unique timestamps: the two ties above are broken so the first four records (by event time) carry the strictly increasing timestamps 1, 2, 3, 4 (payloads unchanged). This regime is the baseline in which no two events share an event time.

⁵ <https://github.com/AhmedAwad/WindowAssessor/>

- *Partial order*: The records are sorted by their timestamp, keeping the ties. Because event time defines only a partial order (Definition 1), two records remain at timestamp 1 and two at timestamp 2. We call this the in-order trace, but it is not a total order. Contrasting it with the total-order regime isolates the effect of equal-timestamp ties at window boundaries (Sect. 4.9).
- *Out-of-order*: The records in the arrival order listed above, in which arrival order differs from event-time order (e.g., r_1 has timestamp 13 but arrives before r_6 with timestamp 1).

In our experiments, we create windows, a sliding time window and a session window, and count the elements per key in each window⁶. In all cases, the engine reports (*start*, *end*, *key*, *count*), i.e., the window defined by its boundaries and the count of items it holds.

4.1 Sliding time windows

We define a sliding-time window with width $\omega = 10$ and a slide of $\beta = 2$ milliseconds. We use milliseconds to match the default time granularity in all systems. For instance, event $r_1 = (13, A, 22)$ will actually have the Unix epoch timestamp 1970-01-01T00:00:00.013Z.

All five engines assign an event to the same $\frac{\omega}{\beta}$ windows, computed identically. For an event e , the start of the highest-start enclosing window is given by Eq. 1:

$$t_{\text{start}} = \left\lfloor \frac{e.t_e}{\beta} \right\rfloor \times \beta, \quad (1)$$

aligned to the Unix epoch (offset 0); the operator then works backwards in steps of β to the remaining windows. The engines differ in only two conventions: the boundary-inclusion rule, $[start, end)$ for Spark Structured Streaming, Kafka Streams, Beam, and Flink versus $(start, end]$ for Storm; and whether windows with a negative start (before the epoch) are emitted, which Kafka Streams suppresses. Because the per-engine sliding outputs are long, we present and compare them within each engine's subsection, while we compare the more compact session outputs inline in the next subsection.

4.2 Session windows

We define a session window as a 10 *milliseconds* gap. We chose this value deliberately: it exceeds the largest inter-arrival gap in the key- A trace, so that under in-order arrival all eleven records fall into a single session, while under out-of-order arrival the same gap is still small enough to expose

session merges and the engines' differing acceptance of late elements. More generally, the gap is the sole knob controlling sessionisation granularity: a larger gap merges elements into fewer, longer sessions, while a smaller one yields more, shorter sessions, down to per-element windows as the gap tends to zero, so its effect scales with the stream's temporal density.

With ordered data, and, identically, with total-order data, since a single session spanning all key- A records is unaffected by the ties, all engines supporting session windows agree on the session *content*: 11 elements for key A . They differ, however, in how they *label* the session boundaries. Spark, Beam, and Flink report (1, 36, A , 11), taking the session end as the timestamp of the last record plus the gap, whereas Kafka Streams reports (1, 26, A , 11), taking the end as the timestamp of the last record in the session. The final result does not reveal the internals of window merges. To expose them, we forced the engines to report after each processing step: this is the default behaviour in Kafka Streams; we forced it in Flink by defining a custom trigger, and in Spark we used the `complete` output mode. Because `complete` retains all window state without eviction, it is a testing device for observing the transient merges rather than a reporting mode, so the reported Spark session results remain the `append` output.

With unordered data, the reported results diverge further. Flink reports (13, 25, A , 2), (14, 36, A , 3) and Kafka Streams reports (13, 15, A , 2), (14, 26, A , 3), whereas Spark and Beam report (13, 25, A , 2), (23, 36, A , 2). The four systems agree on the content of the first session, two elements, at timestamps 13 and 15, up to the same end-labelling convention seen with ordered data, namely that Kafka Streams ends a session at the timestamp of the last record while Spark, Beam, and Flink end it at the last timestamp plus the gap. They disagree on the second session: Flink and Kafka Streams admit more elements into it (three) than Spark and Beam (two). This indicates a difference in the criterion for accepting new elements between Flink and Kafka Streams on the one hand, and Spark and Beam on the other, with respect to how event-time progress is used when adding elements from a source (cf. Sect. 3.3). We elaborate on this below for each system.

Before turning to individual engines, we delimit which parts of our model (Sect. 3) the evaluated systems exercise, so that the per-engine discussions map uniformly onto it. All five expose `addElement`, `trigger`, `eviction/retention (purge)`, and `reporting`; the multi-buffer engines additionally exercise `merge` through session windows, whose instances grow and coalesce as elements arrive. None of the engines, however, exercises `split`: a split arises only for data-driven specifications with a non-monotonic membership predicate (e.g., threshold windows, Example 4),

⁶ Storm does not have native support for session windows; therefore, it is excluded from the respective evaluation.

and, as Table 2 shows, the evaluated engines support only data-agnostic (sliding, tumbling) and session specifications; none offers threshold or comparable data-driven windows. The split-driven reporting of Sect. 3.4 therefore cannot be observed here; we note this once rather than per engine. Table 3 summarises how each engine instantiates the lifecycle; the subsections below add only what is distinctive and then report results.

4.3 Apache storm

Storm uses single-buffer maintenance with configurable time- and count-based triggers and evictors (Table 3); it does not support data-driven windows. It computes a watermark periodically as the maximum event time seen minus a configurable `lag`, and this value is also the minimum acceptable timestamp, so any older (late) tuple is dropped. Concretely, on each arrival, Storm follows an evict-then-compute routine: it advances the watermark, evicts stale tuples, computes and emits every closed window, then admits the new element if it is not late. As a single-buffer engine, Storm supports stateless reporting only (cf. Definition 8).

Results for sliding-time windows

For ordered (partial-order) data, we observed the following output from Storm: $(-8, 2, A, 3)$, $(-6, 4, A, 4)$, $(-4, 6, A, 5)$, $(-2, 8, A, 5)$, $(0, 10, A, 5)$, $(2, 12, A, 1)$, $(4, 14, A, 3)$, $(6, 16, A, 3)$, $(8, 18, A, 3)$, $(10, 20, A, 4)$, $(12, 22, A, 4)$, $(14, 24, A, 3)$, $(16, 26, A, 3)$, $(18, 28, A, 3)$, $(20, 30, A, 2)$, $(22, 32, A, 2)$, $(24, 34, A, 1)$. Under Storm's $(start, end]$ convention, the earliest window $(-8, 2]$ should contain the first four tuples, whose timestamps are 1, 1, 2, and 2. Storm, however, reports a count of 3, not 4. This is a direct consequence of the single-buffer, evict-then-compute-on-arrival

lifecycle (Sect. 3.2) interacting with a co-timestamped tie on the window boundary: the first tuple with timestamp 2 advances the watermark to 2, which closes window $(-8, 2]$ and computes its result over the elements seen so far, $\{1, 1, 2\}$; the second tuple with timestamp 2 arrives afterward, once $(-8, 2]$ has already been computed and its elements evicted, so it is not counted in that window (it is still admitted into later, still-open windows). The resulting count of 3 cannot be produced by any consistent interval convention on these elements; $[start, end)$ would give 2 and $(start, end]$ would give 4, so it is an artefact of the partial order. It disappears under a total order, as we show next and analyse in Sect. 4.9.

For total-order data (timestamps made unique), Storm reports $(-8, 2, A, 2)$, $(-6, 4, A, 4)$, $(-4, 6, A, 5)$, $(-2, 8, A, 5)$, $(0, 10, A, 5)$, $(2, 12, A, 3)$, $(4, 14, A, 3)$, $(6, 16, A, 3)$, $(8, 18, A, 3)$, $(10, 20, A, 4)$, $(12, 22, A, 4)$, $(14, 24, A, 3)$, $(16, 26, A, 3)$, $(18, 28, A, 3)$, $(20, 30, A, 2)$, $(22, 32, A, 2)$, $(24, 34, A, 1)$. The earliest window is now $(-8, 2, A, 2)$, the clean $(start, end]$ count over $\{1, 2\}$, with no anomalous value.

For out-of-order data, we forced Storm to progress the watermark by every tuple to ensure a fair comparison with the other systems. Details are in our GitHub repo. We observed the following output from Storm: $(4, 14, A, 1)$, $(6, 16, A, 2)$, $(8, 18, A, 2)$, $(10, 20, A, 2)$, $(12, 22, A, 2)$, $(14, 24, A, 1)$, $(16, 26, A, 1)$, $(18, 28, A, 1)$, $(20, 30, A, 1)$, $(22, 32, A, 1)$, $(24, 34, A, 1)$. The first window $(4, 14, A, 1)$, for instance, contains only the tuple $(13, A, 22)$. The window was closed by the arrival of the next tuple $(15, A, 23)$. With the fourth tuple, $(27, W, 0)$, the watermark is set to 27. Therefore, Storm drops all subsequent tuples for key A . As a result, the window, e.g., $(14, 24, A, 1)$, contains only one element, $(15, A, 23)$, whereas other elements that would have been included in the in-order case are dropped, i.e., $(19, A, 30)$ and $(23, A, 7)$. Storm drops any tuple if

Table 3 Per-engine instantiation of the window lifecycle: maintenance, event-time progress, late-acceptance knob, retention, and natively supported reporting policies

Engine	Buffer	Event-time progress	Late-acceptance knob	Retention	Native reporting policies
Storm	Single	periodic watermark ($\max t_e - \text{lag}$)	lag	evict on advance ($p=0$)	Stateless only
Spark SSS	Multi	implicit watermark	withWatermark delay	watermark-based purge	Stateless (append), Refinable (update/complete); sessions append-only
Kafka Streams	Multi	stream-time ($\max t_e$)	grace	window-store retention	Refinable (default, fixed-boundary), Retractable (sessions), Stateless (suppress)
Beam	Multi (Global: single)	watermark (at source)	allowedLateness	GC at end + lateness	Stateless (DISCARDING), Refinable (ACCUMULATING)
Flink	Multi (Global: single)	watermark (configurable)	allowedLateness	purge at end + lateness	Refinable (default), Stateless (emulated), Retractable (Table/SQL)

none of the windows it belongs to are still open. That is, $\nexists t_{end} : t_{end} - \beta < e.t_p \leq t_{end} \wedge t_{end} > watermark$.

4.4 Spark structured streaming

Apache Spark unifies batch and stream processing; from version 2, Spark Structured Streaming (SSS) [30] added event-time processing with multi-buffer maintenance and watermarks [31]. The watermark governs retention (purging), not triggering; a batch or processing-time trigger re-evaluates the retained windows, and the output mode (append, update, complete) selects the reporting policy (Table 3), with session windows restricted to append. Sessions of the same key merge as elements arrive, finalising once the watermark passes the session end.

Results for sliding-time windows

For total-order data, SSS reports $(-8, 2, A, 1)$, $(-6, 4, A, 3)$, $(-4, 6, A, 5)$, $(-2, 8, A, 5)$, $(0, 10, A, 5)$, $(2, 12, A, 4)$, $(4, 14, A, 3)$, $(6, 16, A, 3)$, $(8, 18, A, 3)$, $(10, 20, A, 4)$, $(12, 22, A, 4)$, $(14, 24, A, 4)$, $(16, 26, A, 2)$, $(18, 28, A, 3)$, $(20, 30, A, 2)$, $(22, 32, A, 2)$, $(24, 34, A, 1)$, $(26, 36, A, 1)$. In this regime, the only difference between SSS and Storm is the interval convention: on windows not affected by a boundary-aligned timestamp, the two coincide (e.g., $(0, 10, A, 5)$, $(4, 14, A, 3)$, $(18, 28, A, 3)$), while they differ exactly where $[start, end]$ excludes and $(start, end]$ includes the boundary event (e.g., SSS $(-8, 2, A, 1)$ versus Storm $(-8, 2, A, 2)$, and SSS's extra $(26, 36, A, 1)$ with no Storm counterpart).

The observed output for the in-order (partial-order) data is $(-8, 2, A, 2)$, $(-6, 4, A, 4)$, $(-4, 6, A, 5)$, $(2, 12, A, 3)$, $(0, 10, A, 5)$, $(-2, 8, A, 5)$, $(4, 14, A, 2)$, $(8, 18, A, 3)$, $(6, 16, A, 3)$, $(12, 22, A, 4)$, $(10, 20, A, 4)$, $(16, 26, A, 2)$, $(14, 24, A, 4)$, $(18, 28, A, 3)$, $(20, 30, A, 2)$, $(22, 32, A, 2)$, $(26, 36, A, 1)$, $(24, 34, A, 1)$. Comparing the earliest window from Spark to the one from Storm, we observe $spark = (-8, 2, A, 2)$ against $storm = (-8, 2, A, 3)$: Spark's $[start, end]$ interval excludes the boundary timestamp 2, whereas Storm's value additionally reflects the partial-order tie artefact discussed above (a consistent $(start, end]$ count would be 4). The $[start, end]$ convention also produces $spark = (26, 36, A, 1)$, which has no counterpart in Storm. Moreover, the output windows above are not sorted by window start timestamp. This indicates SSS's multi-buffer nature: the SSS window operator processes window buffers arbitrarily.

For the out-of-order data, we observe the following windows: $(4, 14, A, 1)$, $(12, 22, A, 2)$, $(8, 18, A, 2)$, $(10, 20, A, 2)$, $(6, 16, A, 2)$, $(14, 24, A, 1)$, $(18, 28, A, 2)$, $(20, 30, A, 2)$, $(22, 32, A, 2)$, $(26, 36, A, 1)$, $(24, 34, A, 1)$. We observe agreement with the output from Storm on the windows reported,

except that SSS does not output window $(16, 26, A, _)$ because of the nature of the $[start, end)$ inclusion. This also affects the element counts in each window. Spark drops any tuple if none of the windows it belongs to are still open. That is, $\nexists t_{end} : t_{end} - \beta \leq e.t_e < t_{end} \wedge t_{end} > watermark$.

Results for session windows

For ordered data, no key- A tuple is dropped, as expected; to expose the merges behind the single reported session (Sect. 4.2), we re-ran under complete output mode.

For out-of-order arrival, the two sessions arise as follows. When $(26, A, 15)$ arrives it advances the watermark to 26, closing the first session and creating $[26, 36)$; the next key- A tuple creates $[23, 33)$, which overlaps and is merged into $[23, 36)$. The very late tuples $(1, 2, 5, 2, 1)$ for key A create windows whose ends are already behind the watermark and do not overlap the open session, so they are dropped. When $(14, A, 28)$ arrives, its window $[14, 24)$ ends behind the watermark (30) and does not overlap the open session, so SSS drops it, as it does the last tuple $(19, A, 30)$. We therefore deduce Spark's acceptance criterion (cf. `addElement`, Sect. 3.3): a tuple e is admitted iff $e.t_e + gap \geq watermark$.

4.5 Kafka streams

Kafka Streams supports sliding and session windows with multi-buffer maintenance. Progress is driven by *stream-time* (the maximum observed event timestamp) rather than an explicit watermark, and late acceptance is governed by a per-window *grace* interval. Processing is per record: each arriving element updates every window instance it belongs to and, by default, immediately forwards the updated result downstream as a changelog, while `suppress` defers forwarding until the window closes (Table 3). How that update is reported depends on whether the window's identity changes. For a fixed-boundary window (e.g., sliding), the window key is stable, so Kafka Streams forwards the new aggregate under the same key, which is refinable reporting (Definition 9). For a session window, a merge changes the boundaries, so Kafka Streams first emits a retraction (a *null*-valued record) for each superseded session and then the merged result, which is retractable reporting (Definition 10). A window closes once stream-time exceeds its end plus *grace*, after which its records are dropped; sessions extend or merge across inactivity gaps.

Results for sliding-time windows

In all three regimes, every window Kafka Streams emits (those with $start \geq 0$) has the same content as the corresponding SSS window, so it differs from Storm only

by the $[start, end)$ inclusion. Kafka Streams drops a tuple e when no window containing it is still open, i.e., $\nexists t_{end} : t_{end} - \beta \leq e.t_e < t_{end} \wedge t_{end} > watermark$.

Results for session windows

For ordered data, no key- A tuple is dropped; we confirmed the session's incremental expansion by running in the default report-on-every-tuple mode.

For out-of-order data, the divergence from Spark stems from tuple acceptance in `addElement`. Kafka Streams is context-aware: on receiving e it first checks whether e 's singleton window $[e.t_e, e.t_e + gap)$ is mergeable with an open session, rather than only comparing it against the watermark. When $(14, A, 28)$ arrives, stream-time is 30, and the open session is $[23, 26]$; unlike Spark, which drops the singleton $[14, 24)$, Kafka Streams finds it mergeable (their gap is $9 \leq 10$) and merges into $[14, 26]$, giving three elements $\{14, 23, 26\}$. The last tuple $(19, A, 30)$ arrives after stream-time has reached 500, by which point the session $[14, 26]$ is closed, so it is dropped. Kafka Streams labels the session end at the last timestamp, unlike Spark's last-timestamp-plus-gap (Sect. 4.2).

4.6 Apache beam

Apache Beam [32] is a unified programming model executed by runners (e.g., Flink, Spark, Dataflow), not an SPE itself; we use its default (direct) runner. Its window operator is multi-buffer (`GlobalWindow` is single-buffer). The default event-time trigger fires when the watermark passes a window's end, optional early and late triggers fire before and after within `withAllowedLateness`, and the accumulation mode selects the reporting policy (Table 3). Sessions are mergeable.

Results for sliding-time windows

Beam produces the same results as SSS for the sliding-time window in all three regimes. Consequently, it overlaps with the Kafka Streams windows whose start time is 0 or greater, and we can derive the same tuple-acceptance criterion as SSS.

Results for session windows

Beam produces the same results as SSS for the session window, matching both window bounds and content across ordered, out-of-order, and total-order data. We therefore classify the Beam session operator as context-agnostic when accepting tuples, like SSS.

4.7 Apache flink

Apache Flink [18] is a stream-first SPE with native event-time processing and stateful, multi-buffer windowing (`GlobalWindow` is single-buffer). The default event-time trigger closes a window when the watermark passes its end, `allowedLateness` admits later firings, and optional evictors run before or after the window function. The trigger and window-function settings select the reporting policy (Table 3); Flink's Table/SQL API also emits explicit retractions.

Results for sliding-time windows

Flink follows $[start, end)$ intervals, so on data without co-timestamped ties or out-of-order arrivals it agrees with SSS and Beam. Under partial-order and out-of-order regimes, however, its per-record watermark, together with disabled allowed lateness, closes a boundary window one element early, producing a single divergent window in each case. For total-order data, Flink's output matches SSS and Beam; with unique timestamps and in-order arrival, there is no boundary element to lose.

For the in-order (partial-order) data, Flink's output is identical to SSS except for the earliest window: Flink reports $(-8, 2, A, 1)$ whereas SSS and Beam report $(-8, 2, A, 2)$. Because we advance the watermark to the maximum event time after every record and disable allowed lateness, the first record with timestamp 1 raises the watermark to 1, which reaches the maximum timestamp of window $[-8, 2)$ and fires it with a single element; the second record with timestamp 1 then finds $[-8, 2)$ already closed and is not admitted to it (it is still admitted to the later windows $[-6, 4) \dots [0, 10)$, which is why those match SSS).

For the out-of-order data, Flink matches SSS except window $(18, 28)$: Flink reports $(18, 28, A, 1)$ against SSS's $(18, 28, A, 2)$. The watermark reaches 27 when the tuple $(27, W, 0)$ arrives, firing window $[18, 28)$ (whose maximum timestamp is 27) with only the element $(26, A, 15)$ seen so far. The out-of-order tuple $(23, A, 7)$ arrives afterwards and is dropped from $[18, 28)$, whereas SSS keeps $[18, 28)$ open until the watermark passes 28 and therefore still admits it.

Results for session windows

Flink is context-aware like Kafka Streams: it merges the singleton window of $(14, A, 28)$ into the still-open session even though that singleton's end is behind the watermark, so its second session holds three elements $\{14, 23, 26\}$. Unlike Kafka Streams, Flink labels the session end as the last timestamp plus the gap (as SSS and Beam do), which is why its boundaries read $(13, 25)$ and $(14, 36)$ rather than Kafka

Streams' (13, 15) and (14, 26) (Sect. 4.2). Context-aware engines (Flink, Kafka Streams) thus admit relatively more tuples than context-agnostic ones (SSS, Beam), giving better resilience to out-of-order arrival.

4.8 Other distributed systems

We also considered three other open-source engines, each of which reduces to a design already covered. Apache Heron⁷ follows Storm, so its window operator is single-buffer and inherits Storm's capabilities and limitations. Apache Samza [33] and Apache Apex⁸ follow Apache Beam's event-time model, Samza only through Beam's Samza runner (its native APIs lack explicit event-time support), Apex through Beam-inspired APIs of its own, so both are multi-buffer, though Samza supports only tumbling and session windows and Apex disallows custom windowing logic. None, therefore, adds a new point in our design space.

4.9 Total, partial, and out-of-order ingestion

Ingesting an identical set of records under the three arrival regimes isolates the sources of cross-engine divergence and lets us attribute each observed discrepancy to a distinct underlying cause.

Total order.

When every event time is unique, and records arrive in event-time order, all observed differences reduce to two conventions, each a fixed and predictable property of the engine:

- *Sliding boundary inclusion.* Storm includes an event in the $(start, end]$ window, whereas SSS, Beam, Flink, and Kafka Streams use $[start, end)$. Consequently, a boundary-aligned event is counted one window earlier or later; e.g., Storm reports $(-8, 2, A, 2)$ where the others report $(-8, 2, A, 1)$, and the $[start, end)$ engines emit a trailing $(26, 36, A, 1)$ that is absent from Storm. Away from boundary-aligned events, all five engines agree.
- *Session end labelling.* All engines agree on session *content*, but Kafka Streams labels a session's end as the last event's timestamp, whereas SSS, Beam, and Flink use the last timestamp plus the gap; hence $(1, 26, A, 11)$ versus $(1, 36, A, 11)$.

Kafka Streams also suppresses windows with a negative start (before the Unix epoch), yet produces identical content

to SSS for every window it emits. Thus, under a total order, the engines are semantically interchangeable up to these two conventions.

Partial order.

Real event time is only a partial order (Definition 1): distinct records may share a timestamp. When such a tie falls on a window boundary, engines that fire a window on watermark *advance* can close it *between* two co-timestamped records, so the boundary window's count depends on the processing-time interleaving of those records rather than on their event-time content. Two instances appear in our data:

- Storm's earliest window is $(-8, 2, A, 3)$, a value no fixed interval can yield ($[start, end)$ gives 2, $(start, end]$ gives 4). The first record at timestamp 2 advances the watermark to 2, closing and computing $(-8, 2]$ over $\{1, 1, 2\}$; the second record at timestamp 2 arrives afterwards and is evicted from that window.
- Flink reports $(-8, 2, A, 1)$ where SSS and Beam report $(-8, 2, A, 2)$: the first record at timestamp 1 fires $[-8, 2)$ before the co-timestamped second record is admitted.

Both anomalies disappear under the total-order regime, confirming that they are artefacts of co-timestamped ties at a boundary, not of the query or of the multiset of event times. In other words, the result of a boundary window is *undetermined* by the query together with the event-time content alone.

Out-of-order.

Out-of-order arrival adds a third axis: late records whose windows have already closed are dropped, governed by each engine's progress and retention policy. Flink reports $(18, 28, A, 1)$ where SSS reports $(18, 28, A, 2)$, because Flink's per-record watermark fires $[18, 28)$ as soon as the watermark reaches 27 (set by the $(27, W, 0)$ tuple), before the out-of-order $(23, A, 7)$ arrives, whereas SSS keeps the window open until the watermark passes 28. For session windows, the acceptance criterion diverges further: context-aware engines (Flink, Kafka Streams) merge a late singleton session into an open session even when the singleton's end is behind the watermark, admitting more elements than context-agnostic engines (SSS, Beam).

Context beyond window boundaries.

Taken together, the partial-order and out-of-order regimes reveal a dimension of window context that existing classifications do not capture. Traub et al. [34], adopting the classes

⁷ <https://apache.github.io/incubator-heron/>

⁸ <https://apex.apache.org/>

introduced by Li et al. [22], distinguish *context-free* (CF), *forward-context-free* (FCF), and *forward-context-aware* (FCA) windows according to how many tuples an operator must process to know where windows start and end; sliding and tumbling time windows are context-free, since all window boundaries can be computed a priori from the window parameters, independently of the data. That classification concerns *where* windows begin and end. Our results expose a second, orthogonal dimension: even for a context-free window, whose boundaries are fixed a priori, the *content*, and even the *existence*, of a given window instance can depend on the arrival order and on the operator's progress state at firing time. Under a partial order, the content of the fixed-boundary window varies with the interleaving of co-timestamped elements (Storm reports 3, Flink 1, and SSS/Beam 2); any window whose boundary coincides with a tied timestamp is likewise affected. Under out-of-order arrival, entire fixed-boundary windows may be absent from the output because all of their would-be members arrive late and are dropped; for instance, the early windows present in Storm's in-order output do not appear in its out-of-order output. A sliding time window is thus *boundary-context-free* but neither *content-* nor *existence-context-free* once event time is a partial order or arrivals are out of order: the reported result is determined not by timestamps alone, but by the operator's progress state when the window fires.

This progress state is, in turn, shaped by a *global* watermark. In all five systems, a single, stream-wide watermark tracks event-time progress; consequently, the record that advances the watermark and prematurely closes a boundary window need not even belong to the same key as the window it closes. In a multi-key stream, ties or out-of-order arrivals in one key can thus curtail unrelated keys' windows. A per-key watermark, such as the keyed-watermark operator of [16], which reconstructs Flink's centralised watermark as a distributed operator that maintains event-time progress independently per key, localises this effect to the key that actually carries the ties or late data, leaving the windows of all other keys intact. Our single-key trace isolates the phenomenon in its purest form; in the general multi-key case, keyed watermarks would prevent its propagation across keys.

Implications for portability and testing.

These results give software engineers a concrete checklist for configuring, testing, and porting a windowed application across engines. First, the *same* query over the *same* event-time data can yield different boundary-window counts and different session boundaries; a downstream component that thresholds on a window count, or that keys on a session's reported end, is not portable without adjustment; porting a

Table 4 Knobs that align window behaviour across the multi-buffer engines

Concept	Flink	Spark SSS	Kafka streams	Beam
<i>Tunable knobs, can be matched to align behaviour</i>				
Event-time progress	watermark strategy	implicit (with Watermark)	stream-time= max ts	source watermark
Late-acceptance horizon	allowed Late-ness	with Watermark delay	grace	with Allowed Late-ness
Defer vs. early report	trigger	trigger+output Mode	default; suppress	triggers
Stateless report	emulate (delta)	append	suppress	DIS-CARDING
Refinable report	default	update/complete	default	ACCUMULATING
Retractable report	Table/SQL changelog	not native	native for sessions	not native
<i>Fixed per engine, residual divergence, not tunable</i>				
Session end label	last ts + gap	last ts + gap	last ts	last ts + gap
Session acceptance	context-aware	context-agnostic	context-aware	context-agnostic

The lower group is fixed per engine (not tunable) and accounts for the residual divergence of Sect. 4.9.

sliding-count alert from Spark to Storm shifts every boundary-aligned count by the interval convention, and porting a sessionisation job from Kafka Streams to Spark changes both the reported session end (last timestamp versus last timestamp plus gap) and, under out-of-order arrival, which late elements are admitted. Second, when the input can contain co-timestamped events, tests must exercise ties *on window boundaries*, since these expose order-dependent behaviour that a merely in-order (but total-order) trace would hide. Third, the divergence a ported application will experience is predictable from the operator's position in our model, and this induces two portability clusters. A pipeline written for Storm ports to any multi-buffer engine with a single systematic change: the boundary-inclusion convention ($[start, end] \rightarrow [start, end)$), since the multi-buffer engines subsume Storm's stateless, buffer/punctuation-based behaviour; the reverse is *lossy*, because Storm retains no window instances ($p = 0$) and reports only statelessly, so refinable/retractable reporting and late re-firing cannot be reproduced on it. Among the multi-buffer engines, portability reduces to matching three *tunable* knobs: event-time progress, the late-acceptance horizon, and the reporting policy (Table 4).

Two coordinates, however, are *fixed per engine and cannot be reconciled by configuration*: the boundary-inclusion convention and, for session windows, the end labelling and **context-awareness** of tuple acceptance. Context-aware engines (Flink, Kafka Streams) merge a late singleton into an open session, whereas context-agnostic ones (Spark, Beam) drop it. These non-tunable residuals bound how closely any two engines can agree.

4.10 Threats to validity

Our findings are configuration-sensitive: knobs such as `allowedLateness` (Flink, Beam), `grace` and `suppress` (Kafka Streams), and watermark settings (Spark) determine whether a late element is re-fired or dropped, and our Beam results use the default runner, whose trigger and accumulation semantics other runners may realise differently. For a fair comparison, we advanced the watermark to the maximum event time after every record; a coarser or periodic watermark would shift the boundary-firing effects of Sect. 4.9. Because the outcome of a co-timestamped tie on a window boundary depends on the processing-time interleaving of the tied records, our partial- and out-of-order results reflect the specific ingestion order in our repository; a different but equally valid interleaving could yield different boundary-window contents, an order dependence that is itself one of our findings. Finally, our analysis reflects the engine versions in Table 2; future releases may evolve their APIs or defaults.

5 Related work

The earliest description and formalisation of sliding time and landmark windows dates to 2001 [35], when single-pass correlated aggregates were computed. TelegraphCQ [36] and CQL [37] have extended those window types with a sliding tuple window. The formalisation focuses only on defining the scope of the different window types. A comprehensive formalisation appears in [28]. CQL interestingly allows the user to receive updates not only upon the arrival of new stream elements, but also upon their removal from a window. Out-of-order arrival was not in scope.

SECRET [21] was the first model that addresses the lifecycle of a window instance defined under sliding time or tuple windows. Beyond being limited to sliding windows, the model can describe SPE behaviour only when data arrive in order. Affetti et al. [12] analysed the execution semantics of Flink, Storm, Spark discretised streaming, Azure Stream Analytics and Google Cloud Dataflow based on the SECRET model. They acknowledge the shortcomings of SECRET we discussed earlier.

The Dataflow model [14] represents a major step toward formalising multi-buffer style window management. It explicitly distinguishes event-time from processing-time, introduces watermarks as progress indicators, and allows early, on-time, and late triggers so that the same window instance can fire multiple times. A key contribution of the model is recognising the need to merge windows (e.g., in session windows) when late data extend existing boundaries. However, the model does not address the complementary need to split windows when out-of-order arrivals invalidate earlier assignments. As a result, while Dataflow provides a powerful foundation and underpins systems such as Apache Beam and Flink, its semantics remain incomplete for handling the full range of out-of-order scenarios. Our work complements this by generalising both merge and split behaviours within a unified lifecycle model and mapping them systematically across contemporary SPEs.

Grossniklaus et al. [15] motivated the need for data-driven windows, called frames, by presenting four types of windows: *threshold*, *delta*, *aggregate* and *boundary* frames. The idea is that the user does not need to define a fixed time or tuple window size. Rather, for example, for threshold frames, the system shall report the start and end time of the frame in which the value of the attribute of interest is above/below the defined threshold. In their original work, the authors assumed a single-buffer window operator and only a buffering technique to handle out-of-order data. In our work, we elaborate on how single- and multi-buffer operators would, in general, handle such window specifications and show examples of threshold windows.

Gabriele et al. [38] introduced two new models for parallel processing of sliding windows. The focus is on in-order processing of elements on sliding tuple and time windows. They adopt the notions of triggers and evictors as in [11]. Compared to our work, we discuss more window types and contribute an out-of-order-aware window lifecycle model where the application developer can enable multiple triggers throughout a window instance lifecycle.

Sun and Hwang [39] propose an approach to handle out-of-order and late arrivals using two pipelines: one for data that arrive in order and another for data that arrive late. The results from the normal flow are kept in a state after the windowing operator, where they are updated by results from the late pipeline. The approach is similar to [29]. The paper provides no details on how updates are merged. Moreover, the approach seems limited to fixed-size windows, where an element's assignment to a window depends only on its timestamp.

Verwiebe et al. [40] provide a comprehensive survey of window types for stream processing systems research and commercial systems. The survey covers a rich set of

window types and briefly discusses the effects of out-of-order arrivals and their support.

6 Conclusion

This paper introduced a unified semantic framework for reasoning about window operators under late and out-of-order data, centred on a window-operator lifecycle model. Distinguishing single-buffer from multi-buffer maintenance and formalising their lifecycle phases, event inclusion, recomputation, and reporting policies, we showed how these choices govern correctness, latency, and reporting, and why seemingly similar windowing abstractions in Apache Beam, Flink, Spark Structured Streaming, Kafka Streams, and Storm diverge in practice. Placing an operator in the model lets an engineer predict its late-data handling (refine vs. drop), its message amplification, and the feasibility of data-driven boundary changes (merge and split), turning these into system-aware decisions when configuring, testing, and porting stream-processing applications across engines.

Our model generalises beyond the systems we studied: it places any windowed operator by its maintenance strategy, boundary-inclusion convention, progress and retention policy, and reporting policy, and its lifecycle primitives, including merge and split, describe data-driven specifications such as threshold and aggregate windows. Mainstream engines, however, expose only data-agnostic and session windows, so the split behaviour the model captures cannot yet be exercised on them; as engines add such specifications, or adopt finer-grained progress such as per-key watermarks, the model applies without change. The model thus provides a durable basis for reasoning about window semantics as stream-processing systems evolve.

Acknowledgements The author thanks Riccardo Tommasini for fruitful, insightful discussions that significantly shaped the ideas presented in this work.

Author contributions As a single author, I did everything, conceptualisation, writing, experimentation, and validation.

Funding Not applicable.

Data availability All codes and used data can be found in this GitHub repo, <https://github.com/AhmedAwad/WindowAssessor/tree/master>

Declarations

Competing interests The authors declare no competing interests.

Ethical approval Not applicable.

References

- Chen, M.: Real-time compliance monitoring engine based on ebpf: Privacy-enabled data tracking for edge computing. *Proced. Comput. Sci.* **281**, 216–225 (2026)
- Jan, M.A., Adil, M., Brik, B., et al.: Making sense of big data in intelligent transportation systems: current trends, challenges and future directions. *ACM Comput. Surv.* **57**(8), 197:1–197:43 (2025). <https://doi.org/10.1145/3716371>
- Bakirci, M.: Internet of things-enabled unmanned aerial vehicles for real-time traffic mobility analysis in smart cities. *Comput. Electr. Eng.* **123**, 110313 (2025)
- Raychoudhury, V., Cao, J., Kumar, M., et al.: Middleware for pervasive computing: a survey. *Pervasive Mob. Comput.* **9**(2), 177–200 (2013). <https://doi.org/10.1016/j.pmcj.2012.08.006>
- Medeiros, D.S.V., Neto, H.N.C., Lopez, M.A., et al.: A survey on data analysis on large-scale wireless networks: online stream processing, trends, and challenges. *J. Internet Serv Appl* **11**(1), 6 (2020). <https://doi.org/10.1186/s13174-020-00127-2>
- Daksa, R.P., Kemala, A.P.: A comparative study on real time data streaming for fraud detection using kafka with apache flink and apache spark. *Proced. Comput. Sci.* **269**, 192–199 (2025). <https://doi.org/10.1016/j.procs.2025.08.272>
- Khodabandehlou, S., Golpayegani, A.H.: Fiffraud: unsupervised financial fraud detection in dynamic graph streams. *ACM Trans. Knowl. Discov. Data* **18**(5), 1–29 (2024). <https://doi.org/10.1145/3641857>
- Jurzenia, K., Rak, J.: A survey of vehicular network systems for road traffic management. *IEEE Access* **10**, 42365–42385 (2022). <https://doi.org/10.1109/ACCESS.2022.3168354>
- Chakilam, C.: Leveraging ai, ml, and big data for precision patient care in modern healthcare systems. *Euro. J. Anal. Artif. Intell. (EJAAI)* p-ISSN 3050-9556 en e-ISSN 3050-9564 **2**(1) (2024)
- Tiwari, N., Biswal, P.K., Ranjan, P.: Blockchain-enhanced electronic health records: a kafka-based approach for iomt integration. *J. Netw. Syst. Manag.* **34**(2), 56 (2026). <https://doi.org/10.1007/s10922-025-10025-2>
- Gedik, B.: Generic windowing support for extensible stream processing systems. *Soft. Pract. Exp.* **44**(9), 1105–1128 (2013)
- Affetti, L., Tommasini, R., Margara, A., et al.: Defining the execution semantics of stream processing engines. *J. Big Data* **4**(1), 12 (2017)
- Awad, A., Tommasini, R., Langhi, S., et al.: D²ia: User-defined interval analytics on distributed streams. *Inf. Syst.* **104**, 101679 (2022). <https://doi.org/10.1016/j.is.2020.101679>
- Akidau, T., Bradshaw, R., Chambers, C., et al.: The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *VLDB Endowment* **8**(12), 1792–1803 (2015)
- Grossniklaus, M., Maier, D., Miller, J.: Frames: data-driven Windows, In: DEBS. ACM, pp. 13–24 (2016)
- Yasser, T., Arafa, T., ElHelw, M., et al.: Keyed watermarks: a fine-grained watermark generation for apache flink. *Future Gener. Comput. Syst.* **169**, 107796 (2025). <https://doi.org/10.1016/j.future.2025.107796>
- Kafka Streams (2016). <https://kafka.apache.org/documentation/streams/>
- Carbone, P., Katsifodimos, A., Ewen, S.: Apache flink™: stream and batch processing in a single engine. *IEEE Data Eng. Bull.* **38**(4), 28–38 (2015)
- Allen, S.T., Jankowski, M., Pathirana, P.: Storm Applied: Strategies for Real-time Event Processing, 1st edn Manning Publications Co., Greenwich, CT, USA (2015)

20. De Matteis, T., Mencagli, G.: Parallel patterns for window-based stateful operators on data streams: an algorithmic skeleton approach. *Int. J. Parall. Prog.* **45**(2), 382–401 (2017)
21. Dindar, N., Tatbul, N., Miller, R.J.: Modeling the execution semantics of stream processing engines with secret. *The VLDB Journal* **22**(4), 421–446 (2013)
22. Li, J., Maier, D., Tufte, K.: Semantics and evaluation techniques for window aggregates in data streams. In: *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, pp. 311–322. ACM, New York, NY, USA (2005). <https://doi.org/10.1145/1066157.1066193>. (SIGMOD '05)
23. Tucker, P.A., Maier, D., Sheard, T., et al.: Exploiting punctuation semantics in continuous data streams. *IEEE Trans. Knowl. Data Eng.* **15**(3), 555–568 (2003)
24. Awad, A., Traub, J., Sakr, S.: Adaptive watermarks: a concept drift-based approach for predicting event-time progress in data streams. In: *Advances in Database Technology - 22nd International Conference on Extending Database Technology, EDBT 2019, Lisbon, Portugal, March 26-29, 2019. OpenProceedings.org*, pp 622–625 (2019)
25. Dayarathna, M., Perera, S.: Recent advancements in event processing. *ACM Comput. Surv.* **51**(2), 33:1–33:36 (2018). <https://doi.org/10.1145/3170432>
26. De Matteis, T., Mencagli, G.: Parallel patterns for window-based stateful operators on data streams: an algorithmic skeleton approach. *Int. J. Parall. Prog.* **45**(2), 382–401 (2017)
27. Rabl, T., Sakr, S., Hirzel, M.: Big stream processing systems (dagstuhl seminar 17441). *Dagstuhl Reports* **7**(10), 111–138 (2017)
28. Patroumpas, K., Sellis, T.K.: Window specification over data streams. In: *EDBT Workshops*, pp. 445–464. (2006)
29. Krishnamurthy, S., Franklin, M.J., Davis, J., et al.: Continuous analytics over discontinuous streams. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010. ACM*, pp 1081–1092 (2010)
30. Zaharia, M., Xin, R.S., Wendell, P., et al.: Apache spark: a unified engine for big data processing. *Commun. ACM* **59**(11), 56–65 (2016)
31. Zaharia, M., Das, T., Li, H., et al.: Discretized streams: Fault-tolerant streaming computation at scale. In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. ACM, New York, NY, USA, SOSP '13*, pp 423–438 (2013) <https://doi.org/10.1145/2517349.2522737>
32. Foundation, A.S.: Beam (2016). <https://beam.apache.org/>
33. Noghabi, S.A., Paramasivam, K., Pan, Y.: Samza: Stateful scalable stream processing at linkedin. *Proc VLDB Endow* **10**(12), 1634–1645 (2017). <https://doi.org/10.14778/3137765.3137770>
34. Traub, J., Grulich, P.M., Cuellar, A.R., et al.: Scotty: General and efficient open-source window aggregation for stream processing systems. *ACM Trans. Database Syst.* **46**(1), 1:1–1:6 (2021). <https://doi.org/10.1145/3433675>
35. Gehrke, J., Korn, F., Srivastava, D.: On computing correlated aggregates over continual data streams. *SIGMOD Rec* **30**(2), 13–24 (2001)
36. Chandrasekaran, S., Cooper, O., Deshpande, A., et al.: Telegraphcq: continuous dataflow processing. In: Halevy AY, Ives ZG, Doan A (eds) *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003. ACM*, p 668 (2003) <https://doi.org/10.1145/872757.872857>
37. Arasu, A., Babu, S., Widom, J.: The CQL continuous query language: semantic foundations and query execution. *VLDB J.* **15**(2), 121–142 (2006)
38. Mencagli, G., Torquati, M., Lucattini, F., et al.: Harnessing sliding-window execution semantics for parallel stream processing. *J. Parall. Distrib. Comput.* **116**, 74–88 (2018)
39. Sun, D., Hwang, S.: Dssp: Stream split processing model for high correctness of out-of-order data processing. In: *2018 IEEE First International Conference on Artificial Intelligence and Knowledge Engineering (AIKE). IEEE*, pp 193–197 (2018) <https://doi.org/10.1109/AIKE.2018.00044>
40. Verwiebe, J., Grulich, P.M., Traub, J., et al.: Survey of window types for aggregation in stream processing systems. *VLDB J.* **32**(5), 985–1011 (2023). <https://doi.org/10.1007/s00778-022-00778-6>

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor (e.g. a society or other partner) holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.