The VLDB Journal (2023) 32:985–1011
https://doi.org/10.1007/s00778-022-00778-6

REGULAR PAPER

Survey of window types for aggregation in stream processing systems

Juliane Verwiebe[1] · **Philipp M. Grulich**[1] · **Jonas Traub**[1] · **Volker Markl**[1]

Received: 13 March 2022 / Revised: 12 December 2022 / Accepted: 29 December 2022 / Published online: 17 February 2023

Abstract
In this paper, we present the first comprehensive survey of window types for stream processing systems which have been presented in research and commercial systems. We cover publications from the most relevant conferences, journals, and system whitepapers on stream processing, windowing, and window aggregation which have been published over the last 20 years. For each window type, we provide detailed specifications, formal notations, synonyms, and use-case examples. We classify each window type according to categories that have been proposed in literature and describe the out-of-order processing. In addition, we examine academic, commercial, and open-source systems with respect to the window types that they support. Our survey offers a comprehensive overview that may serve as a guideline for the development of stream processing systems, window aggregation techniques, and frameworks that support a variety of window types.

Keywords Survey · Window types · Window aggregation · Stream processing systems · Out-of-order processing

1 Introduction

Modern real-time analytic workloads require the execution of complex queries over continuous, unordered, and high-velocity data streams [98]. To handle the unbounded nature of data streams, *windowing* constitutes a fundamental concept in stream processing [4]. Windowing divides streams into windows (i.e., finite subsets of data), which is a prerequisite for operators, such as window aggregations or joins [41]. The window type determines how to split the stream into windows and which tuples to include. This impacts the window content, changes the result of windowed operations, and enables different application scenarios [76].

Over the last years, researchers and practitioners proposed a wide range of different window types to address a variety of use-cases [29]. For instance, window types with a fixed-size (e.g., tumbling windows, sliding windows) do not address all requirements of complex stream processing applications. Such applications often require data-driven windows which dynamically adapt depending on data characteristics (e.g., punctuation-based windows [41,56], session windows [4,92,93], Frames [50]). These different window types are described across a large number of publications with different notations, specifications, and synonyms. This makes the adoption and development of stream processing systems (SPSs) for end-users, system developers, and researchers very challenging. System developers strive for general-purpose SPSs that support a wide range of window types and workloads. However, they are confronted with a large number of different windowing concepts, which they have to adapt to modern system features such as out-of-order stream processing. As a result, system developers have to re-implement window types leading to unclear semantic differences among systems. Additionally, researchers investigate techniques for the efficient handling of windowed operations [57]. Due to the variety of different window types, they often focus only on specific windowing semantics, which limits the applicability of their work [28] and leads to different performance profiles of window types. Consequently, end-users have to learn and resolve contradicting semantics and implementations among different systems, which increases the time to master new systems.

Despite the variety of window types and publications on the topic of windowing, there exists, to the best of our

✉ Philipp M. Grulich
grulich@tu-berlin.de

Juliane Verwiebe
juliane.verwiebe@tu-berlin.de

Jonas Traub
jonas.traub@tu-berlin.de

Volker Markl
volker.markl@tu-berlin.de

[1] Technische Universität Berlin, Sekr. EN-7, Einsteinufer 17, 10587 Berlin, Germany

knowledge, no exhaustive survey which collects, studies, and characterizes the different window types. Our goal is to provide an overview of all window types to support the work of system developers, researchers, and end-users of SPSs. To this end, we reviewed 23 publications, which introduce different window types (Sect. 4). Furthermore, we included the documentations of 17 stream processing systems: Amazon Kinesis Data Analytics [7], Google Cloud Dataflow [46], IBM Streams [59], Microsoft Azure Stream Analytics [71], Microsoft StreamInsight [6,49], Borealis [1], Naiad [74], Trill [31,32], STREAM [16], TelegraphCQ [33], Apache Beam [4,13], Flink [27], Samza [75], Spark [89,96], Storm [90], and Heron [39,64]. Across these resources, we have noticed a variety of different terminologies, definitions, and classifications. To this end, we have derived 16 window types for which we unify all specifications.

In this paper, we provide the first unified survey of window types for stream processing systems. We describe in detail how each window type is specified, provide formal notations, show visualizations, and give use-case examples. We explain out-of-order processing for each window type and make suggestions where this is not covered in literature. Due to many inconsistencies in notations across literature, we address synonyms and contradictory definitions. We provide overview tables for all window types and examined systems, which serve as an entry point for researchers and practitioners.
The contributions of this paper are as follows:

1. We provide a thorough survey of window types for aggregation in SPSs.
2. We classify all window types based on categories used in literature: context-awareness, window measures, determinism, order characteristics (FIFO vs. non-FIFO), overlapping vs. non-overlapping, fixed- vs. variable-size.
3. For each window type, we provide a formal notation, use-cases, investigate how they can be combined with out-of-order processing, and list the synonyms.
4. We provide an overview of SPSs and their support for different window types.

The remainder of this paper is structured as follows. First, we discuss foundational background on stream processing (Sect. 2). Then, we introduce classifications, which enable a unified specification of the window types (Sect. 3). We describe our methodology (Sect. 4) and present specifications of the 16 different window types (Sect. 5). We examine systems and window aggregation implementations (Sect. 6). Finally, we discuss related work (Sect. 7).

2 Background

We now introduce the preliminaries for this survey. We define stream processing (Sect. 2.1), discuss the notion of time (Sect. 2.2), present the concept of window aggregation (Sect. 2.3), describe out-of-order processing (Sect. 2.4), and examine common aggregation techniques (Sect. 2.5).

2.1 Stream processing

The real-time analysis of continuously produced data is a key requirement of modern, low-latency applications. Stream processing fulfills this requirement, enabling many applications in different areas such as machine monitoring, stock market trading, and communication technology. Data for such applications is generated by sources which are, for instance, software and hardware sensors, transactions, or clicks [9]. These sources produce data continuously, resulting in an unbounded data set termed *data stream*. They push tuples to *stream processing systems* (SPSs) which are execution engines that continuously process incoming data streams to provide real-time information and responses to long-standing queries (Table 1).

Formally, a data stream $\bar{s}$ consists of a sequence of tuples [29]. A *tuple* (also called *data item* [58], or *record* [27]) represents the unit of data that is used for communication within stream processing systems and applications [58]. It may contain a sequence of attribute values which describe an entity [56]. $T = \langle e_1, e_2, \ldots, e_N \rangle$ denotes the schema of the tuples represented as set of elements of finite arity N. The set of sequences of tuples, that are derived from this schema, is formally expressed as $Seq(T)$. The notation $s_{position}(x) = \bar{s}(x)$ defines a tuple at position x in the stream $\bar{s} \in Seq(T)$, where $position \in \{t_e, t_p, t_i, count, \ldots\}$. The position of a tuple can be defined based on different times (e.g., event-time t_e, processing-time t_p), $count$ of tuples, or other arbitrary advancing measures. Additionally, $s_{position}(x).A_i$ describes attribute A_i of the tuple at position x.

2.2 Notion of time

The concept of time plays an important role in SPSs. For instance, time is essential to divide streams into time-based windows (Sect. 2.3). In general, we can differentiate between three time domains, *event-time*, *processing-time*, and *ingestion-time* [4,27]. *Event-time* t_e specifies the time at which a tuple was generated by a source. This timestamp is attached to the produced tuple and does not change [4]. The *processing-time* t_p refers to the time the system-clock has, when the system or the operator processes the tuple [4]. *Ingestion-time* t_i denotes the time at which a tuple arrives in the system [27]. Timestamps of every time domain are mono-

Table 1 Notations

Formal notation	Explanation
$T = \langle e_1, e_2, \ldots, e_N \rangle$	Tuple schema: set of elements of finite arity N
$\bar{s} \in Seq(T)$	Stream
$s_{position}(x) = \bar{s}(x)$	Tuple at position x of stream $\bar{s}$
$position \in \{t_e, t_p, t_i, count, \ldots\}$	Different positions (Sect. 2.2, Section 3.3)
$\bar{s}([a, b[) = \{s_{position}(x) \mid x \in [a, b[\}$	Substream containing tuples at positions within interval
$w_i = s[b_i, e_i[$	Window i
b_i	Beginning of window i
e_i	End of window i
p	Window parameter (e.g., size, slide, timeout)
$\nexists y : s_{t_e}(y) \geq s_{t_e}(x) \wedge y < x$	Tuple $s_{position}(x)$ is in-order
$\exists y : s_{t_e}(x) < s_{t_e}(y) \wedge y < x$	Tuple $s_{position}(x)$ is out-of-order

tonically increasing. Other time definitions could be possible (e.g., sending time, receiving time).

2.3 Windows and window aggregation

Windows separate the unbounded stream into bounded subsets (i.e., windows) depending on a *window type* (e.g., sliding, tumbling, or session window). The tuples contained in each window are aggregated based on a *window function* (also called *aggregate function* [93]), typically to derive some metric (e.g., sum, max, average) [51]. Window aggregation constitutes an essential technique for SPSs to handle continuous streams online and to produce timely responses. Aggregations can be differentiated in temporal and spatial. Temporal aggregation refers to a summary of values over time, while a spatial aggregation summarizes values from different sources [91]. In this survey, we focus on temporal aggregations over data streams. Window types are different methods of separating the stream into windows which affect the result of the aggregation.

In a formal notation, intervals of the form $[a, b[$, with $a, b \in \mathbb{R}$ and $a < b$, are used to depict finite subsequences from this conceptually infinite stream $\bar{s}$. This interval ranges from a to b (only including a). The notation $\bar{s}([a, b[)$ (short $s[a, b[$) describes a substream that contains all tuples at positions within the interval. This is formally described with $\bar{s}([a, b[) = \{s_{position}(x) \mid x \in [a, b[\}$. A window is described as a substream $w_i = s[b_i, e_i[$, where b_i is the beginning and e_i the end of the window i. The beginning and ending of a window are called *window edges* [94] (also called window bounds [76] or window boundaries [41]). We define the positions of the edges in different window measures, such as event-time or tuple count (Sect. 3.3).

Example: $w_1 = s[3, 5[$ describes window 1 with $b_1 = 3$ and $e_1 = 5$. w_1 includes every tuple $s_{position}(x)$ with $3 \leq x < 5$. In event-time, w_1 spans the time from tuple $s_{t_e}(3)$ at timestamp 3 until tuple $s_{t_e}(5)$ at timestamp 5, excluding tuple

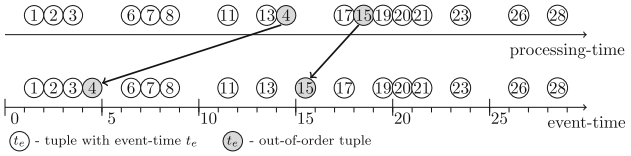

Fig. 1 Out-of-order tuples

$s_{t_e}(5)$. Every tuple that arrives before tuple $s_{t_e}(5)$ belongs to w_1 (e.g., a tuple $s_{t_e}(4.9)$ at timestamp 4.9). Tuple $s_{t_e}(5)$ at position 5 is included, for example, in the subsequent window $w_2 = s[5, 8[$. If $w_1 = s[3, 5[$ is defined in tuple counts, it contains all tuples between $s_{count}(3)$ and $s_{count}(5)$, excluding the tuple at count 5.

2.4 Out-of-order processing

This section summarizes existing literature on out-of-order stream processing (further details can be found in Akidau et al. [4], Carbone et al. [27], Li et al. [69], Traub et al. [92–94]). The requirement to process tuples in the correct event-time order leads to a distinction between *in-order* streams and *out-of-order* streams. A stream is called *in-order*, if all tuples arrive in the correct order according to their event-timestamps [4,69]. We formally define that tuple $s_{position}(x)$ is in-order, if $\nexists y : s_{t_e}(y) \geq s_{t_e}(x) \wedge y < x$ [94]. However, streams that are perfectly event-time ordered are unusual, since tuples are often produced by a variety of distributed sources. Due to network congestion, transmission delay, or sensor failure, in practice, tuples arrive unordered with respect to their event-timestamps [93]. These *out-of-order tuples* (or *late tuples* [69]) are the reason why streams are *out-of-order*. A tuple is considered out-of-order, if at least one tuple before it has a greater event-time [92]. We formally define that tuple $s_{position}(x)$ is out-of-order, if $\exists y : s_{t_e}(x) < s_{t_e}(y) \wedge y < x$. Fig. 1 illustrates a data stream ordered according to processing-time, i.e., it shows

the tuples in the order in which they arrive. The stream is out-of-order because the tuples with event-timestamps 4 and 15 are out-of-order. The stream below illustrates where the correct positions of the out-of-order tuples are when the stream is ordered by event-time.

Conceptually, SPSs can only trigger the computation of a window aggregate, if they can guarantee that all data has been seen and no further out-of-order tuple will arrive. This leads to a delay of the window aggregation output. To avoid unnecessary long delays, *watermarks* are inserted into the stream which act as measures of progress [27]. Watermarks include a timestamp t that signals the system how long to wait for out-of-order tuples that have a timestamp lower than t. As a result, some operators end the waiting for out-of-order tuples, when they receive a watermark [27]. Then, they trigger the computation of the window aggregate for every window that ended before the watermark and output the aggregate. We formally define this as $trigger(w_i) \forall w_i \mid e_i < watermark(t)$.

Systems apply different approaches to handle out-of-order tuples. In general, we can differentiate between three strategies to handle out-of-order tuples. First, the system can process the tuples in the order of arrival and ignore out-of-order tuples (e.g., Aurora [2] when $Slack = 0$). Second, systems can buffer all tuples, sort them according to their event-timestamps, and then apply the window operator (e.g., Borealis [1]). Third, the window operator handles out-of-order tuples [69], which is also implemented in Dataflow systems [4] (e.g., Apache Flink [27]).

In the last case, the window operator handles in-order tuples at the time they arrive to avoid processing delays. Precomputing intermediate results provides a lower latency for outputting the aggregation results (Sect. 2.5). The window operator has to insert the out-of-order tuples into the correct window(s) with respect to their event-time in the stream. This requires a recomputation of their window aggregates; otherwise, the aggregation results would not be equal for the same input stream. Depending on the aggregation operation, window aggregates can be incrementally updated (e.g., by incrementally adding a value to a sum) or have to be recomputed from scratch (e.g., concat strings [83]). Additionally, inserted tuples may cause a change of the edges of one or more windows that were already created in the past. For instance, if a window holds 10 tuples and an out-of-order tuple is inserted, there are 11 tuples in the window leading to a violation of the window specification. As a result, the window operator has to shift one tuple to the next window. Consequently, out-of-order tuples require special care in processing, and handling out-of-order streams increases the complexity of window aggregation [57,67,83,93]. We will describe out-of-order processing for different window types in Sect. 5.

2.5 Window aggregation optimization techniques

Windows affect the performance of SPSs since computations must be performed on the contained tuples to obtain aggregation results. Large windows cause inefficiencies as a result of high memory footprints and long response times, since all tuples in a window have to be maintained and then iterated to compute an aggregate. Overlapping windows partly include the same tuples leading to redundant computations. Thus, research has proposed several techniques to optimize window aggregation (e.g., FlatFAT [84], Pairs [63], Cutty [29], Scotty [94]). Across this work, we can identify two fundamental optimization techniques: *partial-window-aggregation*, and *aggregate sharing*.

Partial window aggregation separates an aggregate into intermediate results called partial aggregates [29]. A new partial aggregate is computed through combining two partial results [84]. The final aggregation result can be obtained with just a few last aggregation steps using the pre-computed partial aggregates. Depending on the type of aggregation, a technique may store just the partial aggregates for each window instead of keeping all tuples [93]. With each arriving tuple, a partial aggregate can be incrementally updated [28]. For instance, the values of the tuples are incrementally added to a sum every time a new tuple arrives. Partial window aggregation resolves inefficiencies by reducing the memory footprint as well as the latency.

The approach of *aggregate sharing* handles the overlap of subsequent windows from one window query as well as the overlap of concurrent windows from multiple queries [29]. The partial aggregates are shared among overlapping windows. For the output of each window, several partial aggregates are combined to calculate the final aggregate [63]. This prevents redundant computations.

3 Classification

In this section, we classify window types according to several categories. These categories allow for grouping window types with respect to performance characteristics and implementation requirements. We discuss the differentiation between fixed-size and variable-size (Sect. 3.1), overlapping and non-overlapping (Sect. 3.2), window measure (Sect. 3.3), and FIFO vs. non-FIFO (Sect. 3.4). We describe the approach of Carbone et al. [29] that introduced deterministic and non-deterministic window types (Sect. 3.5). Li et al. [67] classified window types with respect to the required processing context (Sect. 3.6). We summarize all categories in Table 2 and use them to classify the different window types in Sect. 5.

Table 2 Categories

Category	Differentiation
Fixed-size vs. variable-size	Fixed-size
	Variable-size
Overlapping vs. non-overlapping	Overlapping
	Non-overlapping
Window measure	Time-based
	Count-based
	Multi-dimensional
FIFO vs. non-FIFO	FIFO
	Non-FIFO
Deterministic vs. non-deterministic	Deterministic
	Non-deterministic
Context-awareness	Context-free (CF)
	Forward-context free (FCF)
	Forward-context aware (FCA)

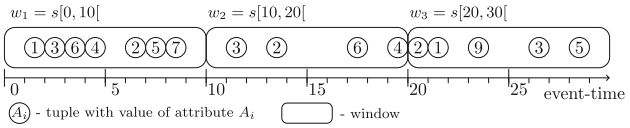

Fig. 2 Example for fixed-size windows

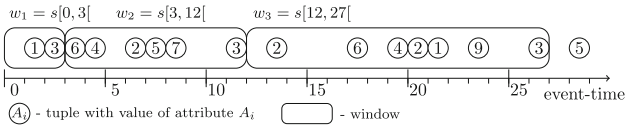

Fig. 3 Example for variable-size windows

3.1 Fixed-size vs. variable-size

The first category we consider is window size, adopting the terms *fixed-size* and *variable-size* from Bifet et al. [21]. A window is per definition a bounded subset of the stream. Consequently, its *size* (also called *length* or *range* [28]) can be quantified in different metrics such as time or number of tuples. For example, a window can have a size of 10 s, so that it includes all tuples within a 10 s timeframe. Formally, $w_i = [b_i, e_i[$ has a size of 10, if $e_i - b_i = 10$.

The size of some window types is chosen a priori and defined in the window query. Tuples are then inserted into each window until they reach the size that is specified. Those window types are referred to as *fixed-sized*, since their size does not change while they are applied onto the stream. In this case, every consecutive window has the same size. For example, if we define a size of 10 s for a window type, every window has a size of exactly 10 s as shown in Fig. 2. This 10 s size does not change throughout the application of the window.

If a window type does not require a size to be specified beforehand, the window size is adapted dynamically, and is classified as *variable-sized*. Every window has a different size which may be based on some property of the stream or values of a specified attribute of the tuples. Tuples are inserted into the windows until some criterion is fulfilled. For *variable-sized* window types, we do not define a value for the size. For example, in Fig. 3, a window ends at every tuple with value 3 (punctuation).

3.2 Overlapping vs. non-overlapping

We further differentiate into *overlapping* and *non-overlapping* window types. The logic of a window type can lead to an overlap of individual windows. For example, the configuration of the *sliding window* (Sect. 5.2) is specified by the window size and by another parameter called *slide* (also called *hop size* [76]). The slide parameter determines how often a new window starts [51]. If it is smaller than the window size, the next window starts before the first one ended. Then, the subsequent windows are *overlapping*. The smaller the slide, the more frequently a new window starts. Successive windows of overlapping window types partially contain the same tuples. Hence, one tuple is taken into account in the computation of more than one window aggregation. Formally, two windows $w_i,\ w_j \mid j = i + 1$ overlap, if $b_i \leq b_j$ and $b_j < e_i$. In this notation, two overlapping windows are for example $w_1 = [0, 10[$ and $w_2 = [5, 10[$, or $w_1 = [0, 10[$ and $w_2 = [2, 3[$.

On the other hand, if the slide parameter is not defined, equal or greater to the window size, the window type is *non-overlapping*. The successor instance starts after the previous one ended. Thus, one tuple is included in exactly one window aggregation. For two windows $w_i,\ w_j \mid j = i + 1$, this can be notated as $e_i \leq b_j$. For instance, the two windows $w_1 = [0, 5[$ and $w_2 = [5, 10[$, or $w_1 = [0, 5[$ and $w_2 = [7, 12[$ do not overlap. Besides the slide parameter, other window types can also lead to overlapping windows.

3.3 Window measure

There exist different possibilities to specify window parameters such as size and slide which are called window measures [51]. We distinguish into *time-based measures* (also called *logical measurement unit* [76]), *arbitrary advancing measures*, and *count-based measures* (also called *physical measurement unit* [76]). *Time-based measures* determine the window parameter as a time interval such as minutes or seconds. For instance, a window type can be specified by time-based window size of 10 s. In this case, each window has a size of exactly 10 s. Depending on the use of event-time or processing-time for this measure, tuples may belong to different windows. Typically, windows are defined

based on the event-time [4]. However, timestamps do not have to be a time specification. They may also represent different *arbitrary advancing measures* such as a transaction IDs, kilometers traveled, or invoice numbers [93]. Since these measures advance just as timestamps, window types with such measures can be processed the same way as time-based window types. For simplicity, we will use the term time-based throughout the paper which will include these advancing measures. In case of *count-based measures* (also called *tuple-based* [67] or *tuple-driven* [23]), window parameters are specified by a number of tuples. For example, a window type can be determined by a count-based window size of 100 tuples. The tuples included in one window are counted and the specification is fulfilled once the counter equals 100 tuples. Count-based measures are challenging when handling out-of-order tuples. In an event-time ordered stream, a tuple that arrives out-of-order changes the count of all consecutive tuples that have a greater event-time than the out-of-order tuple, affecting the window aggregates and window edges [93]. Carbone et al. also defined window types in the multi-dimensional space of different measure (e.g., combining the time-based and count-based measure) [29].

3.4 FIFO vs. non-FIFO window types

Window types are also differentiated into *FIFO* and *non-FIFO* based on the strategy of inserting and removing their tuples. In *FIFO* principle, a tuple arrives and is inserted into the window. This tuple has the most recent timestamp of tuples in the window until more tuples arrive [57]. Eventually, some have to be removed to keep the window bounded. The tuple that was inserted first into the window then represents the oldest tuple and is the first to be evicted from it [82]. Such a FIFO window type essentially matches a queue [82]. For *non-FIFO* window types, tuples may be inserted and removed in a different order. For instance, not the oldest tuple, but a tuple with a timestamp right in the middle might be removed. An example for a non-FIFO window type is the attribute-delta policy for sliding windows proposed by Gedik [41] (Sect. 5.13).

3.5 Deterministic vs. non-deterministic window types

Carbone et al. [29] differentiated between *deterministic* and *non-deterministic* window types. The term *deterministic* corresponds to a window type, "if at the time that it processes a [tuple] it can decide whether that [tuple] marks (i) the beginning of a window or (ii) the end of a window" [e.g., tumbling windows (Sect. 5.1), sliding windows (Sect. 5.2), punctuation-based windows (Sect. 5.5)] [29]. At the end of a window, results may not be triggered immediately depending on whether the system waits for out-of-order tuples

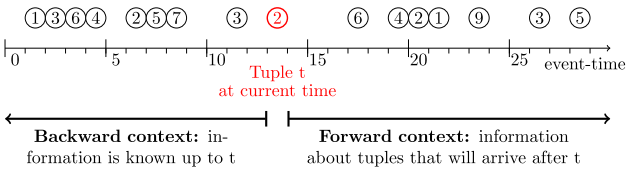

Fig. 4 Backward- and forward-context by Li et al. [67]

(Sect. 2.4). For instance, a count-based tumbling window with a fixed-size of 100 tuples is deterministic. The operator maintains an internal counter that keeps track of the number of tuples in a window. When a tuple arrives and the counter is below 100, that the operator knows that this tuple belongs to the current window. If the counter is at 100, it is immediately known that the current window ends and a new window starts. In contrast, "non-deterministic windows cannot declare immediately whether a record begins a windows or not, i.e., they need to examine more [tuples] in order to take such a decision" [e.g., slide-by-tuple windows (Sect. 5.8), frames (Sect. 5.9), multi-type windows (Sect. 5.13)] [29]. Assume a multi-type window that outputs the last 10 tuples every 5 s. If 11 tuples arrive during the 5 s, the first tuple is not part of the window. However, this is not known immediately when this tuple arrives. The following tuples must be processed before a decision about the beginning of the window can be made. This definition of determinism does not refer to the deterministic aggregation of input tuples, where tuples from different streams are processed in the correct order according to their timestamps to ensure deterministic window aggregation results [54,55].

3.6 Context-free, forward-context free, and forward-context aware window types

The concept of context-awareness was introduced by Li et al. [67]. Their approach is to classify the window types according to the information (i.e., context or state) that is needed to determine start and end timestamps of the windows. They differentiate between *backward-context* and *forward-context* (Fig. 4). "Given a tuple t, its *backward-context* is information about tuples [...] before t. *Forward-context* is information about tuples [...] after t" [67]. If only backward-context is required for a window type, "it implies that the implementation will need to maintain information about previously arrived tuples" [67]. In such cases, the window to which a tuple belongs can be determined immediately. The requirement of forward-context affects the implementation, because "then information from tuples [...] after a tuple t is required to calculate the [windows] for t. This requirement implies that the exact [windows] for tuple t cannot all be determined until those tuples arrive" [67].

Based on the requirement of forward-context, Li et al. derived three categories for window types: *forward-context*

free (FCF), *context-free* (CF), and *forward-context aware* (FCA) [67]. They "define a window [type] as FCF if the [...] implementation does not require forward-context" [67]. The window edges are known up to the timestamp t, when all arrived tuples up to t have been processed (e.g., punctuation-based window) [93]. "Under the FCF category, [...] a window [type is defined] as CF [...] if the implementation of its [window] mapping requires neither forward- nor backward-context" [67]. The edges of all windows are known without processing any tuples (e.g., tumbling window) [94]. The implementation of FCA window types requires forward-context. To map the tuples to windows and to resolve the start and end timestamps of windows before a timestamp t, tuples after t have to be processed (e.g., attribute-delta-based window [41]). As defined by Carbone et al., deterministic window types include the categories of CF and FCF, while non-deterministic window types contain FCA window types [29]. Per definition, non-deterministic window types need to examine tuples after the current tuple to determine whether this tuple starts a window. This corresponds to forward-context.

4 Methodology

In this section, we describe our approach of literature search, as well as the structure of the catalog.

For this survey, we extensively studied literature on the topics of stream processing, windowing, and window aggregation. We selected publications of the most relevant conferences in these research fields, such as SIGMOD, VLDB, EDBT, ICDE, DEBS, SDM SIAM, and CIKM. We considered publications starting in 2001 and 2002, since this was the time when the first SPSs were proposed that implemented windows (e.g., TelegraphCQ [33], STREAM [16]) and the first papers were published that discussed window types for stream processing (e.g., Gehrke et al. [42] in SIGMOD 2001, Guha et al. [53] in ICDE 2002, Carney et al. [30] in VLDB 2002). Since then, a number of different window types have been proposed in research and implemented in various SPSs. Several publications defined window specifications and classifications of window types (e.g., Gedik [41]; Hirzel et al. [56], Li et al. [67], Patroumpas et al. [76]). While these papers include multiple window types, they just cover a smaller subset of all existing window types. Gedik et al. [41] and Hirzel et al. [56] discussed tumbling, sliding, and partitioned windows and their different policies. Patroumpas et al. [76] present fixed-band and landmark windows. Li et al. [67] introduced the slide-by-tuple window. Additionally, publications use synonyms for same window types or provide a different definition causing inconsistencies in the terms used.

To our best knowledge, this paper presents the first comprehensive survey of window types. It provides a unified catalog of window types, contributes to a better understanding, and shows the differences between the window types. We aim to present a detailed description of each window type. This includes a textual and formal specification, synonyms, use-case examples, classification into categories, and out-of-order processing. Resources for finding implementation details are covered in Sects. 6.1 and 6.2.

In Sect. 6.1, selected SPSs have been studied with regard to the supported window types. We have examined academic, commercial, and open-source systems that are relevant, commonly known, or widely used in research or industry. This includes the academic systems TelegraphCQ [33] and STREAM [16], since they were published in 2003 and 2004 and represent two of the earliest systems, and their successor system Borealis [1]. Two of the more recent systems produced in research are Trill [31,32] and Naiad [74]. For commercial systems, we decided to explore the most common ones: Amazon Kinesis Data Analytics [7], Google Cloud Dataflow [46], IBM Streams [59], Microsoft Azure Stream Analytics [71], and Microsoft StreamInsight [6,49]. Our catalog also covers open-source systems, since they are widely used in business as well as in research. To this end, we include Apache Beam [4,13], Apache Flink [27], Apache Samza [75], Apache Spark [89,96], Apache Storm [90], and Apache Heron [39,64]. Specific implementations of window aggregation techniques are discussed in Sect. 6.2.

Table 3 provides a comprehensive overview of all studied window types and acts as a summary. In Sects. 5.1 until 5.12, we present each window type by systematically examining the following points:

- **Specification** describes how the stream is split into subsets of tuples by this window type and gives a formal notation. We also provide a visualization of the window type applied on a data stream.
- **Synonyms** lists the synonyms that are used in research or system specifications for the same type of window.
- **Use-case** illustrates practical application of the window type with a use-case example.
- **Classification** classifies the window type into the categories presented in Sect. 3.
- **Out-of-order processing** explains the handling of out-of-order tuples.

5 Catalog of window types

This section presents the catalog of window types. In Table 3, we provide an overview over all window types, followed by the sections that include the detailed examinations of the respective window types (Sects. 5.1–5.13).

Table 3 Overview over window types

Window type			Window size	Overlapping	Window measure	FIFO	Deterministic	Context-awareness	Short description
5.1 Tumbling Window [28,76,93]			Fixed	No	Time/ Count	✓	✓	CF	Creates periodic, consecutive windows of equal size without gaps
5.2 Sliding Window [4,28,93]			Fixed	Yes	Time/ Count	✓	✓	CF	Defines window edges by size and slide, slide determines next window start
5.3 Fixed-Band Window [76]			Fixed	No	Time	✓	✓	CF	Creates a single window based on predefined timestamps for beginning and end
5.4 Landmark Window [76]			Fixed	No	Time	✓	✓	CF	Sets one window edge on a specific timestamp (i.e., the landmark)
5.5 Punctuation-based Window [41,56]			Variable	No	Time	✓	✓	FCF	Derives window edges by punctuations embedded in the stream
5.6 Session Window [4,92,93]			Variable	No	Time	✓	✓	FCF	Ends window after timeout gap when no tuples arrive in a predefined time interval
5.7 Snapshot Window [5,49,72,73]			Variable	No	Time	✓	✓	FCF	Starts next window when an interval event starts or ends
5.8 Slide-by-Tuple Window [67]			Fixed	Yes	Time	✓	✗	FCA	Creates a sliding window with size measured in time and slide measured in tuples
5.9.1 Threshold Frames [50]			Variable	No	Time	✓	✗	FCA	Starts next frame when specified attribute is above (or below) a predefined threshold
5.9.2 Delta Frames [50]			Variable	No	Time	✓	✗	FCA	Starts next frame when attribute value changes more than predefined amount
5.9.3 Boundary Frames [50]			Variable	No	Time	✓	✗	FCA	Starts next frame when attribute crosses one of the predefined breakpoints
5.9.4 Aggregate Frames [50]			Variable	No	Time	✓	✗	FCA	Starts next frame when aggregate is greater/smaller than predefined constant
5.10 Adaptive Windowing [21]			Variable	Yes	Count	✓	✗	FCA	Adapts window size dynamically to the rate of change in the data stream
5.11 Damped Window [26,70]			Fixed	No	Time	✓	✓	FCF	Assigns a weight to each tuple according to an aging function
5.12 Tilted Window [3,35,43,70]			Fixed	No	Time	✓	✓	FCF	Stores tuples in summaries at different levels of granularity
5.13 Policy-based Window [41,56]									Provides different configurations (called policies) of tumbling and sliding windows
Tumbling Window	Eviction Policy								The policy-based tumbling window is defined by one of four different eviction policies
	Time-based		Fixed	No	Time	✓	✓	CF	
	Count-based		Fixed	No	Count	✓	✓	CF	
	Attribute-delta-based		Variable	No	Attribute-delta	✗	✗	FCA	
	Punctuation-based		Variable	No	Time	✓	✓	FCF	
Sliding Window	Eviction Policy	Trigger Policy							The policy-based sliding window is defined by one of nine combinations of eviction and trigger policies
	Time-based	Time-based	Fixed	Yes	Time	✓	✓	CF	
	Time-based	Count-based	Fixed	Yes	Multi	✓	✗	FCA	
	Time-based	Delta-based	Variable	Yes	Multi	✓	✗	FCA	The multi-dimensional measure is a combination of other measures (i.e., time-based, count-based, and attribute-delta-based)
	Count-based	Time-based	Fixed	Yes	Multi	✓	✗	FCA	
	Count-based	Count-based	Fixed	Yes	Count	✓	✓	CF	
	Count-based	Delta-based	Variable	Yes	Multi	✓	✗	FCA	
	Delta-based	Time-based	Variable	Yes	Multi	✗	✗	FCA	
	Delta-based	Count-based	Variable	Yes	Multi	✗	✗	FCA	
	Delta-based	Delta-based	Variable	Yes	Attribute-delta	✗	✗	FCA	

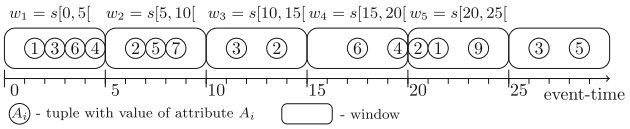

Fig. 5 The time-based tumbling window with a size of 5 s

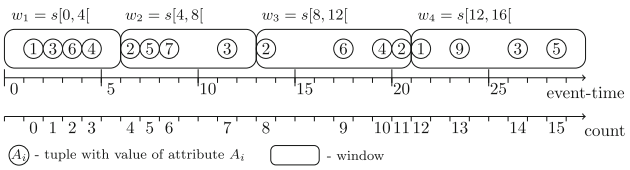

Fig. 6 The count-based tumbling window with a size of 4 tuples

5.1 Tumbling window

Specification A tumbling window splits the data stream into consecutive subsets of tuples of equal window size (Eq. 1). The end of one window represents the beginning of the next window (Eq. 2) [28]. Thus, the tumbling window is non-overlapping [76]. Additionally, there is no gap between the end of one window and the start of the next one. Each tuple belongs to exactly one window (Eq. 3) [28].

$$p_{size} = e_i - b_i \tag{1}$$

$$b_j = e_i \; \forall w_i, \; w_j \mid j = i + 1 \tag{2}$$

$$\forall s_{position}(x) \in \bar{s} \; \exists! \; w_i \mid s_{position}(x) \in w_i = [b_i, e_i[\tag{3}$$

Figure 5 shows a tumbling window with a time-based size $p_{size} = 5$ s. The first window w_1 includes all tuples with an event-timestamp within the window edges $b_1 = 0$ and $e_1 = 5$, excluding the event-timestamp 5. The second window w_2 starts directly after the first one and would include a tuple at event-time 5.

Figure 6 exemplifies a count-based tumbling window with $p_{size} = 4$ tuples, where each window contains four tuples. The window edges are indicated as the count of tuples, not as timestamps.

Synonyms Some publications (e.g., Akidau et al. [4], Begoli et al. [18]) or systems (e.g., Apache Beam [4,11,13], Google Cloud Dataflow [46]) refer to the tumbling window with the term fixed window or list it as a synonym.

The Developer's Guide to Microsoft StreamInsight [73] describes the tumbling window as a gapless and non-overlapping form of the so-called hopping window. The hopping window consists of the parameters window size and hop size (i.e., slide). By this definition, the tumbling window is a subform of the sliding window (Sect. 5.2).

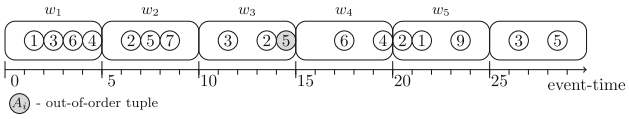

Fig. 7 Out-of-order processing of a time-based tumbling window

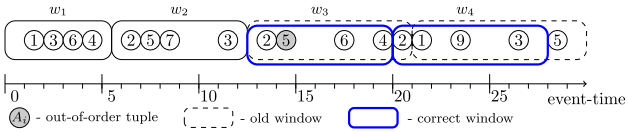

Fig. 8 Out-of-order processing of a count-based tumbling window

Use-case For instance, time-based tumbling window with a size of 24 h could be used to derive a daily report on downloads of an application. Another use-case might be a navigation system in a car that outputs the maximum speed every 30 s to ensure that the speed limit is not exceeded. A count-based tumbling window might be specified with a size of 10,000 tuples. Suppose every tuple corresponds to one click on a website. An interesting aggregation could be the sum of clicks on a topic such as politics or sports, determining the current interests of the users.

Classification (Table 3) The measure of the tumbling window can be time-based or count-based. Regardless, the size is predefined and fixed. Windows do not overlap. The edges of all windows can be computed a priori based on the size parameter which makes this window type CF [93]. In case of a count-based measure, the system has to maintain a counter of tuples. Based on the computed timestamps or the counter, it is possible to decide whether to start a new window when a new tuple arrives. Thus, this window type is deterministic.

Out-of-order processing We summarize the implications on the out-of-order processing which were studied in detail by Traub et al. [93]: For time-based tumbling windows, all timestamps of the window edges are fixed. Thus, tuples arriving out-of-order do not change the window edges and just need to be inserted into the corresponding window. Figure 7 shows that the window edges of w_3 do not change despite an out-of-order tuple. However, the insertion changes the result of the window aggregation, requiring a recomputation of the output. If the aggregate can be computed incrementally, only partial aggregates need to be stored and individual tuples can be dropped, even for out-of-order streams [93].

In contrast, an out-of-order tuple that is inserted into a count-based tumbling window changes the count of all subsequent tuples. Consequently, the last tuple of each window shifts to the next window. As shown in Fig. 8, the tuple $s_{t_e}(20)$ is the 5th instead of 4th tuple in window w_3 after the out-of-order tuple $s_{t_e}(14)$ arrives. It has to be shifted to the next window w_4. Likewise, all subsequent windows change. To

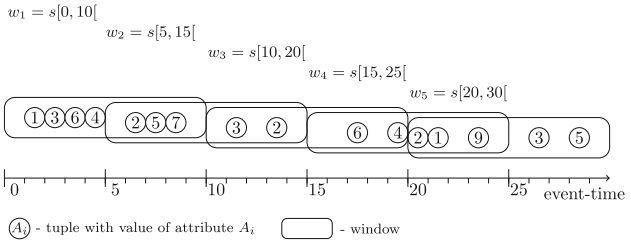

Fig. 9 The time-based sliding window with a size of 10 s and a slide of 5 s

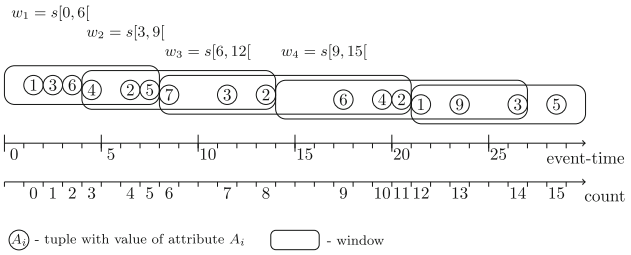

Fig. 10 The count-based sliding window with a size of 6 tuples and a slide of 3 tuples

perform these shifts, all tuples must be stored. For this reason, processing count-based tumbling windows on out-of-order streams increases the memory footprint. Again, the aggregation results of the changed windows have to be recomputed. However, it is possible that an out-of-order tuple arrives with only a small delay and still belongs to the latest window. It can then be inserted into the current window without requiring any shift of tuples.

5.2 Sliding window

Specification The window size of a sliding window defines the subset of tuples that belong to one window (Eq. 4). Additionally to the size of the window, a slide parameter is defined that determines when a new window starts (Eq. 5) [93]. This creates the characteristic overlapping windows of the sliding window (Eq. 6). One tuple belongs to more than one window and is therefore used in the computation of multiple window aggregation results [28]. Windows only overlap, if the slide has a smaller value than the size (Eq. 7) [93]. Equal parameter values $p_{slide} = p_{size}$ or $p_{slide} = 0$ would produce a tumbling window.

$$p_{size} = e_i - b_i \tag{4}$$

$$p_{slide} = b_j - b_i \; \forall w_i,\; w_j \mid j = i + 1 \tag{5}$$

$$b_i \leq b_j \wedge b_j < e_i \; \forall w_i,\; w_j \mid j = i + 1 \tag{6}$$

$$p_{slide} < p_{size},\; p_{slide} > 0 \tag{7}$$

Figure 9 illustrates a time-based sliding window with $p_{size} = 10$ s and a $p_{slide} = 5$ s. The first window w_1 includes all tuples within the window edges $b_1 = 0$ and $e_1 = 10$. Due to the slide parameter, a new window starts every 5 s. As a result, the second window w_2 starts at $b_2 = 5$ s.

An example of a count-based sliding window is shown in Fig. 10, where the window edges are specified in tuple counts. The window ends when the number of tuples equals $p_{size} = 6$ tuples. Window w_1 starts at count of 0 and ends at a count of six tuples. Since a new window starts every three tuples, the start of the subsequent window w_2 is at $b_2 = 3$ tuples.

Synonyms A common synonym for the sliding window is the term hopping window (e.g., Microsoft Azure Stream Analytics [71], Begoli et al. [18], Google Cloud Dataflow [46], Microsoft StreamInsight [73]). The Microsoft StreamInsight Developer's Guide refers to the slide as the hop size and denotes the sliding window as an overlapping hopping window [73]. However, the meaning of the term hopping window may differ from our definition of the sliding window. A hopping window can be understood as a window type that has gaps between its window instances. The window size would then be smaller than its slide [63]. Accordingly, Carbone et al. listed the hopping window additionally to the sliding window and not as a synonym [29].

In Apache Kafka Streams, a sliding window can be specified using the provided hopping window [22]. As this implementation leads to redundant computations for overlapping windows, an additional sliding window implementation was added where each window includes a distinct set of tuples.

The definition of Microsoft Azure Stream Analytics [71] differs from the sliding window described here. A window ends and outputs a result, when the content of the window changes, i.e., a tuple is inserted or evicted from the window.

Use-case A time-based sliding window can output a window of 1 h that is updated every minute, i.e., having a size of 60 min and a slide of 1 min. This can be used in navigation systems to determine the average speed per hour to compute the arrival time.

Sliding windows can also be used for calculating moving averages to smooth time series data and reduce noise [50]. An example is the 5 day moving average of stock market prices. In this scenario, a count-based sliding window with size of five tuples and a slide of one tuple can be applied to calculate the average price every day from the daily closing prices of the last five days.

In the fields data mining [70] and machine learning [21], this window type is often used to determine which subset of tuples contributes, for instance, to the data mining pattern or the cluster partition.

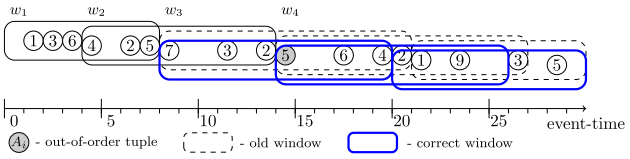

Fig. 11 Out-of-order processing of a count-based sliding window

Classification (Table 3) The sliding window can have a count- or a time-based measure. Due to the predefined parameters, it belongs to the category of fixed-size window types. As mentioned before, the windows are overlapping. A sliding window can be categorized as CF and deterministic, because the start and end of the windows can be computed based on the size and slide of the window. By the arrival of a new tuple, it is clear whether a new window starts or not. This does not require to process tuples or to maintain context.

Out-of-order processing Exactly like the tumbling window, all window edges of the time-based sliding window are fixed. An out-of-order tuple does not change any window edges, but it does change the aggregation results of each window it belongs to. Due to the overlapping windows, an out-of-order tuple could affect several window results, which must then be recomputed. If the aggregate can be computed incrementally, only partial aggregates need to be stored and individual tuples can be dropped, even for out-of-order streams [93].

Inserting an out-of-order tuple into a count-based window changes the tuple count, and thereby the window edges. Figure 11 shows how the last tuple of each window has to be shifted into the next window, beginning from the window in that the out-of-order tuple is inserted. These changes require recomputations of the aggregations results. An out-of-order tuple may arrive with a small delay so that it belongs to the current window according to the counter [93]. This tuple requires a simple insertion without any shift of tuples.

5.3 Fixed-band window

Specification The fixed-band window is defined by two timestamps that specify the lower bound, i.e., the beginning of the window b_i, and the upper bound, i.e., the end of the window e_i (Fig. 12). This leads to a single predefined window (Eq. 8). Every tuple with a timestamp that is between these edges belongs to the window. After the stream reaches the end timestamp, the window content remains unchanged [76].

$$w_1 = s[b_1, e_1[\tag{8}$$

Synonyms The snapshot query in TelegraphCQ [33] resembles the fixed-band window.

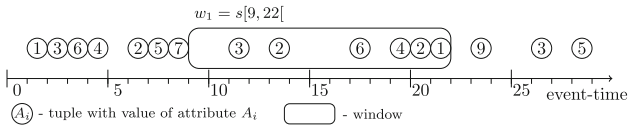

Fig. 12 The fixed-band window $w_1 = s[9, 22[$

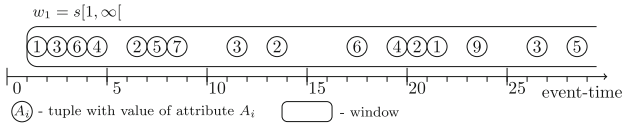

Fig. 13 The lower-bound landmark window

Use-case The fixed-band window could be used to monitor a certain important time period, e.g., the aggregation of air-quality data in the night of New Year's Eve. These are interesting values to maintain for possible comparisons between measurements of cities and the country side, or between the last year and next year.

Classification (Table 3) The fixed-band window belongs to the category of fixed-sized window types, because the size is predefined. For the same reason, this window type can be classified as deterministic and CF.

Out-of-order Processing Out-of-order tuples do not change the window edges of the window. If they contain a timestamp within the window edges, they just have to be inserted into the window.

5.4 Landmark window

Specification For a landmark window, one window edge is fixed at a specific timestamp, the so-called landmark, while the other edge progresses over time. Patroumpas et al. [76] differentiate the lower-bounded landmark window and upper-bounded landmark window. For the lower-bounded landmark window (Fig. 13), the beginning b_1 of window w_1 is predefined and the end timestamp e_1 moves forward (Eq. 9). The window is triggered by a punctuation in the stream [29]. All tuples that belong to the window up to this punctuation are aggregated and the window aggregation result is output. A new landmark can be set after a specified time or number of tuples, for instance, weekly, monthly, or after 1000 tuples, which ends the current window and starts a new window [70].

The upper-bounded landmark window is specified by end timestamp e_1, which represents a future time instant [76]. It can be formally defined by a window that starts at the first position of the stream τ_0 and ends at the specified end timestamp e_1 (Eq. 10). Tuples are inserted into the window immediately at the first timestamp τ_0 when the query is

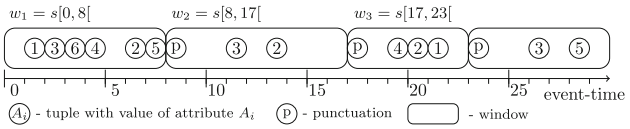

Fig. 14 The punctuation-based window

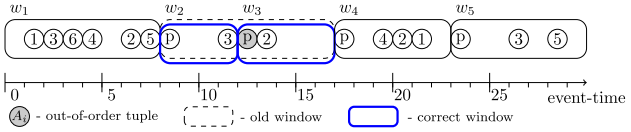

Fig. 15 Out-of-order processing of a punctuation-based window

started until the end timestamp e_1 occurs.

$$w_1 = s[b_1, e_1[\mid p_{punctuation}(e_1) = 1 \tag{9}$$

$$w_1 = [s_{position}(\tau_0), e_1[\tag{10}$$

Use-case The landmark window can be used for searching a maximum or minimum value within the tuples of the stream [33]. Another application area of the landmark window are stream clustering algorithms [70]. Clustering is applied from the landmark to the current timepoint [70].

Classification (Table 3) The lower-bound landmark window starts with the first tuple that arrives and its size changes with each arriving tuple, since no size is predefined. If a new landmark is set, it could be considered as having a predefined length and belonging to fixed-size window types. The upper-bound landmark window has a fixed-size, since its end timestamp is predefined. This window type belongs to the category of non-overlapping windows. It is deterministic and CF, because either its start or end timestamp is predefined [29].

Out-of-order processing Out-of-order tuples can be inserted into the landmark window based on their event-timestamp.

5.5 Punctuation-based window

Specification Punctuations are annotations embedded in the data stream which are specified by an ordered set of patterns for the attributes of the tuples. [95]. A matching function returns true, if a tuple represents a punctuation (Eq. 11). For example, *window punctuations* [41] indicate the boundaries of arbitrary windows (Eqs. 12 13). They can be used to propagate the window edges across operators, if a stream is processed by multiple operators consecutively [41]. A punctuation-based window can be non-overlapping such as a tumbling window (Eq. 14). In this case, tuples are inserted into the window until a window punctuation arrives that indicates the end of one window and the start of a new one. If each punctuation contains a specific window id, end and start punctuations could overlap. In other words, only the punctuation with the same window id ends the window and not the next arriving punctuation. This results in overlapping windows similar to the sliding window (Sect. 5.2).

Figure 14 shows window punctuations with the character p in the tuples. The punctuation with the event-timestamp 8 ends the first window w_1 and starts w_2. Tuples are inserted into this one until the next punctuation arrives, which makes these windows non-overlapping.

$$p_{punctuation}(s_{position}(x)) = \begin{cases} 0 \\ 1 \end{cases} \tag{11}$$

$$b_i = s_{position}(x) \mid p_{punctuation} = 1 \tag{12}$$

$$e_i = s_{position}(y) \mid p_{punctuation} = 1 \wedge y > x \tag{13}$$

$$b_j = e_i \; \forall w_i, \; w_j \mid j = i + 1 \tag{14}$$

Use-case Window punctuations can be differently defined, for instance, with a value, with a special character (e.g., *), or with a sequence of characters (e.g., "electric")). Consider a hybrid car that produces tuples with a schema consisting of an event-timestamp, a measurement value for the level of fuel or battery, and a string for the drive mode, i.e., $s_{t_e}(x) = \langle x, \text{level}, \text{mode} \rangle$. Whenever the defined punctuation tuple $\langle x, \text{level}, \text{"electric"} \rangle$ or $\langle x, \text{level}, \text{"gas"} \rangle$ occurs, the current window ends and a new one begins. Consequently, a window ends when the car switches from gas to electric drive mode or the other way around, because it would otherwise not be possible to calculate the consumption of battery or gas for one drive mode.

Classification (Table 3) This window type is variable-sized, since no size is predefined. The size of the window is determined by the distance of the punctuations and therefore changes dynamically. Depending on the implementation, windows may be overlapping. The window is time-based because punctuations indicate the window edges with their timestamps. The punctuation-based window belongs to the category of deterministic windows, since a window punctuation indicates the start of a new window [29]. The tuples up to a specific timestamp have to be processed to identify the punctuations and derive the windows up to the timestamp. Therefore, backward-context is required and the punctuation-based window can be classified as a FCF window type [67].

Out-of-order processing In this section, we make a suggestion how out-of-order processing could be implemented. An out-of-order tuple does not change the window edges since they are determined by window punctuations. It can be inserted into the corresponding window. However, an out-

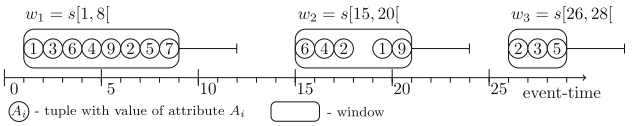

Fig. 16 The session window with a gap of 4 s

of-order window punctuation results in the separation of one window into two windows. Figure 15 depicts an out-of-order punctuation $s_{t_e}(12)$. The second window shifts to $w_2 = s[8, 12[$ and a new window instance with $w_3 = s[11, 17[$ is created. This requires a recomputation of the aggregation result.

5.6 Session window

Specification The session window records periods of activity, i.e., windows are created for periods of time where tuples arrive [4]. The predefined timeout gap $p_{timeout}$ determines after what time of inactivity a session ends (Eq. 15). If no tuple arrives within this gap, the session ends with the last tuple that has arrived [93](Eq. 16). A new session begins with the next tuple that arrives (Eq. 17). The gap $b_j - e_i$ between two session windows can be bigger than the timeout.

$$s_{t_e}(y) - s_{t_e}(x) \geq p_{timeout} \mid y = x + 1 \tag{15}$$

$$e_i = s_{t_e}(x) \tag{16}$$

$$b_j = s_{t_e}(y) \forall w_i,\ w_j \mid j = i + 1 \tag{17}$$

In Fig. 16, the timeout gap is specified as $p_{timeout} = 4$ s. The session window w_1 starts with the first tuple that arrives at timestamp 1. It ends with the tuple $s_{t_e}(8)$, because no other tuple arrives within 5 s after it. Since the time gap between $s_{t_e}(17)$ and $s_{t_e}(19)$ is smaller than 4 s, they still belong to the same session window w_2.

Use-case A typical example for session windows is the detection of browser sessions.

Classification (Table 3) This is a variable-size window type, since the size of the sessions changes according to the length of the period of activity. Sessions are non-overlapping and the timeout gap is defined in a time-based measure. The start of the window is indicated by the first tuple of a new session and the timeout marks window end, making the session window deterministic [29]. The start and end timestamps of the sessions cannot be computed a priori without processing the tuples, thus the session window requires backward-context [93]. Therefore, it belongs to the category FCF windows.

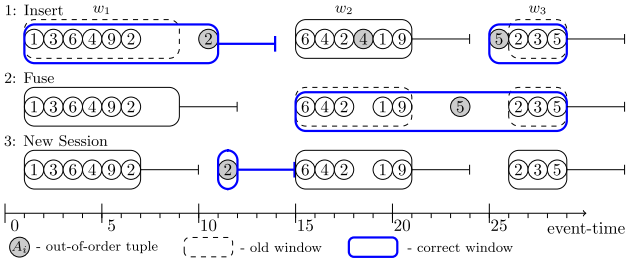

Fig. 17 Out-of-order processing of a session window

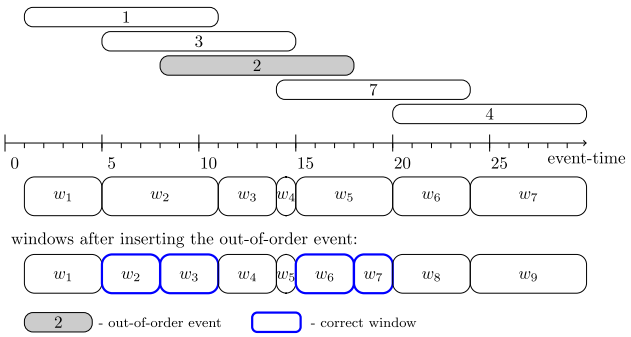

Fig. 18 Out-of-order processing of a snapshot window

Out-of-order processing Figure 17 shows examples of the three possible outcomes of processing an out-of-order tuple according to Traub et al. [92]. In the first case, an out-of-order tuple may be inserted into an existing session, extend its end, or shift its start, as shown in case 1. The tuple $s_{t_e}(17)$ falls within the start and end of the existing session w_2 and can simply be inserted into the session. Secondly, an out-of-order tuple can be located in the timeout gap of a session, such that the end of the session shifts to the timestamp of this tuple. Tuple $s_{t_e}(10)$ shifts the end of the first session w_1. Another tuple $s_{t_e}(25)$ shifts the start of session w_3, because of its position after the end of the timeout of session w_2.

Case 2 in Fig. 17 depicts the second outcome, where two sessions are fused. The two sessions w_2 and w_3 are merged, because the tuple $s_{t_e}(23)$ shrinks the gap of the first session below the specified timeout.

In case 3, the out-of-order tuple $s_{t_e}(11)$ arrives after the gap of one session and before the next session starts. This tuple forms a new session between two existing sessions.

5.7 Snapshot window

Specification The snapshot window depends on a special type of tuples called (interval) events [72] which represent a time interval. Before applying this window type, the timestamps of the tuples are modified. They can be extended to span a certain time interval with the AlterEventDuration() method [72]. This way, their lifetime is enlarged into the

future. We denote this by inserting a tuple into function p_{interval} (Eq. 18). This function outputs interval events that have a left endpoint l_x and a right endpoint r_x [5]. The left endpoint represents the event-timestamp (i.e., when the tuple was generated) and the right endpoint determines the period of time over which an event influences the output [5] (Eq. 19).

Ali et al. [5] defined the snapshot window as "the maximal time interval that contains no event endpoints". A snapshot window is created for each pair of consecutive endpoints ep_x and ep_{x+1} [5] (Eq. 20). Whenever an endpoint of an event occurs, one window ends and a new window is started. Endpoints of interval events are never between the window edges, instead they determine the timestamps of the window edges [5,73]. In Fig. 18, w_2 starts due the left endpoint of the second event $l_5 = 5$ at timestamp 5, and ends due to the right endpoint $r_1 = 10$ of the first event.

In contrast to session windows, the snapshot window detects periods of time where no change is observed in the input, i.e., where no interval event starts or ends which would change the aggregation result. Similar to the session window, this window type would not produce windows, if no interval events occur.

$$p_{interval}(s_{t_e}(x)) = [l_x, r_x[\tag{18}$$

$$r_x = l_x + t \tag{19}$$

$$b_i = ep_x \wedge e_i = ep_{x+1} \mid ep_x \in \{l_x, r_x\} \tag{20}$$

Use-case A use-case is to compute the average over some attribute value of the interval events over the last minute [72]. To obtain interval events before applying the window operator, the tuples are modified to an interval of 1 min starting at their event-timestamp (i.e., $r_x = l_x + 1\text{min}$). As soon as a new event starts, the average changes and a new window is created. The output of each window is the average of some attribute in the events within the last minute [72].

Classification (Table 3) The snapshot window produces variable-sized windows which are measured in time. The windows are non-overlapping, only the interval events overlap. It is deterministic, since the endpoints of an interval event indicate the start of a new window. Therefore, the window edges are not known beforehand and the interval events have to be processed to derive them. Consequently, the window type belongs to the category of FCF windows.

Out-of-order processing In this section, we make a suggestion how out-of-order processing could be implemented. If an endpoint is within window edges, the window has to be split into two windows. The out-of-order event in Fig. 18 spans the interval from timestamp 8 to 18 and lies within the edges of w_2. Consequently, it creates the new end timestamp

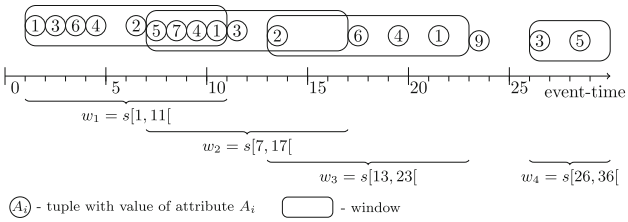

Fig. 19 The slide-by-tuple window with a size of 10 s and a slide of 5 tuples

of w_2 and starts the new window w_3. The end timestamp of the event falls within the edges of the window prior to w_5. This window also has to be separated into two windows.

5.8 Slide-by-tuple window

Specification The slide-by-tuple window represents a special type of sliding window that combines a time-based window size (Eq. 21) with a slide defined in a count measure (Eq. 22) [67].

$$p_{size} = e_i - b_i \mid e_i, b_i \in \bar{s}_{t_e} \tag{21}$$

$$p_{slide} = b_j - b_i \mid b_i, b_{j=i+1} \in \bar{s}_{count} \tag{22}$$

$$b_i \leq b_j \wedge b_j < e_i \; \forall w_i, \; w_j \mid j = i + 1 \tag{23}$$

$$p_{slide} < p_{size} \wedge p_{slide} > 0 \tag{24}$$

For example, Fig. 19 depicts a slide-by-tuple window with $p_{size} = 10$ s and $p_{size} = 5$ tuples. The window w_1 spans the time interval of 10 s from $s_{t_e}(1)$ to $s_{t_e}(11)$. A new window starts whenever the tuple counter equals the slide parameter. Hence, window w_2 begins after five tuples at $s_{t_e}(7)$ and includes all tuples until timestamp 17. If $p_{size} = 1$ tuple, each tuple starts a new window. It is also possible that a tuple neither belongs to an existing window with respect to its timestamp nor starts a new window with respect to the counter. For instance, according to its timestamp, tuple s_{23} does not belong to $w_3 = [13, 23]$, since w_3 only includes four tuples and $s_{t_e}(23)$ is at count 4. Therefore, it does not start the next window w_4.

This window type equals the combination of the time-based eviction policy and count-based trigger policy for sliding windows (Sect. 5.13).

Synonyms This window type is also a special form of the multi-type according to Carbone et al. [29], which describes sliding windows with different measures for size and slide.

Use-case In the context of traffic monitoring, an example could be a road where sensors measure the speed of each car. A slide-by-tuple window is specified with a length of 1 min and a slide of one tuple. Every time a new sensor reading

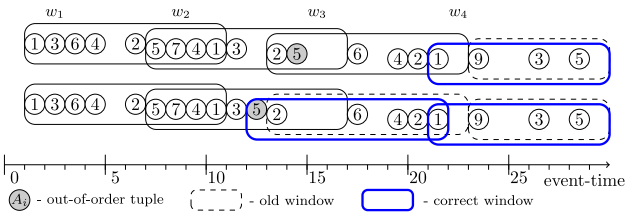

Fig. 20 Out-of-order processing of a slide-by-tuple window

arrives, i.e., a car passes the sensor and produces a tuple, the average speed is calculated over a new window with a length of 1 min.

Classification (Table 3) Due to the predefined size and slide parameters, the slide-by-tuple window belongs to the category of fixed-size window types. As explained in the specification, the window size is defined in a time-based measure. The slide-by-tuple window is non-deterministic, because tuples need to be processed to know the count of the tuples. Only after the number of tuples specified in the slide parameter has arrived, it is clear when a new window starts. Furthermore, the timestamp of the tuple that starts the window is needed to determine the end timestamp of the window which is based on the size defined in a time-based measure. The slide-by-tuple window can be classified as FCA [67] since forward-context in form of processing tuples after a tuple t is needed to determine the windows to which the tuples belong.

Out-of-order processing Since the slide is count-based, window edges will change when an out-of-order tuple arrives. An out-of-order tuple does not change the edges of the windows it belongs to, but the windows after them. In Fig. 20, the out-of-order tuple $s_{t_e}(15)$ belongs to window w_2 and w_3 with respect to its timestamp. Since the start edge of w_3 is already determined, the window is not affected by the change of the tuple count. In contrast, the subsequent window w_4 shifts to $b_4 = 21$, because one tuple is added to the tuple count. Figure 20 depicts another out-of-order tuple $s_{t_e}(12)$ in the second stream. Again the windows to which the tuple belongs do not change. As this tuple does not belong to w_3, its edges shift to $b_3 = 12$ and $e_3 = 22$.

Out-of-order tuples arriving with only a small delay may still belong to the current window or start a new window. They do not require a shift of subsequent windows, since there are none, but it has to be checked, if their count equals the slide value.

5.9 Frames

Grossniklaus et al. [50] proposed Frames, which are data-driven windows with a variable length. Frames adapt their size dynamically to the changing stream characteristics, such as stream rate or data distribution. Window parameters such as the size do not have to be determined beforehand. The desired output of frames is the start and end timestamps of certain periods, rather than an aggregation value.

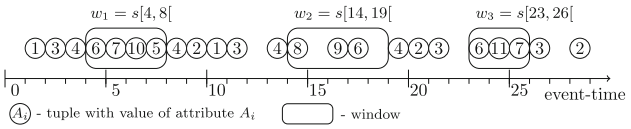

Fig. 21 Threshold frames with a threshold of 4 and a minimum duration of three tuples ($s_{position}(x).A_i > 4 \wedge p_{size} \geq 3$)

5.9.1 Threshold frames

Specification A frame is created, if the value of a specified attribute A_i of a tuple $s_{position}(x)$ is above (or below) a predetermined threshold t (Eq. 25). A minimum duration can be defined ensuring that a frame includes a minimum number of tuples n (Eq. 26).

$$b_i = s_{position}(x) \mid s_{position}(x).A_i \lesseqgtr t \tag{25}$$

$$p_{size} \geq n \text{ tuples} \tag{26}$$

Figure 21 shows threshold frames with a specified threshold $t = 4$ and $p_{size} \geq 3$ tuples. The first frame w_1 starts with $s_{t_e}(4)$, since the attribute value 6 is greater than 4. It ends with $s_{t_e}(8)$, because the attribute value is below the threshold. A frame consisting of only two tuples would be discarded, since p_{size} would be below the minimum duration of 4. Similar to session windows, threshold frames capture interesting periods in the stream. In contrast to session windows, these periods can be defined based on some attribute.

Use-case The use-case for threshold frames proposed by Grossniklaus et al. [50] is dye concentrations that are gathered by oceanographers to study movement and mixing of coastal waters. A fluorescent dye is added to the water and a fluorimeter records the dye concentrations. Dye levels near zero are uninteresting for the oceanographers, but are contained in long sequences of readings. These periods are not captured with threshold frames, which makes them fit for this purpose. For this example, frames should be reported, where the dye concentration is higher than 0.05 units for at least 10 measurements. From this, the formal condition $s_{position}(x).concentration > 0.05 \wedge p_{size} \geq 10$ can be derived.

Classification (Table 3) Frames adapt their size according to the data content and are therefore variable-sized. Fames

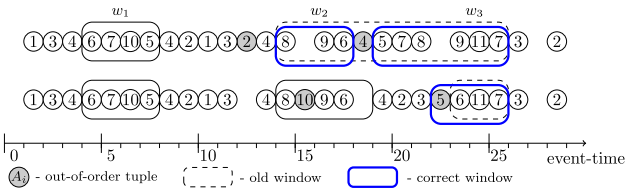

Fig. 22 Out-of-order processing of threshold frames

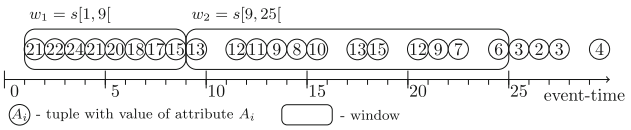

Fig. 23 Delta frames with a delta value of 10 ($|s_{position}(x).A_i - s_{position}(y).A_i > 10$)

are non-overlapping and measured in time, since the output are the start and end timestamps of their edges edges. Frames require forward-context, since the next edge can only be derived after processing the tuples that follow after a certain tuple. They can be classified as FCA windows. Hence, frames belong into the category of non-deterministic window types, since with the arrival of a record cannot be immediately determined whether this tuple begins a new window.

Out-of-order processing In this section, we make a suggestion how out-of-order processing could be implemented. At first, the threshold condition has to be checked for the out-of-order tuple. An out-of-order tuple does not belong to any frame, if it does not satisfy the condition. The out-of-order tuples depicted in the upper stream in Fig. 22 do not meet the condition $s_{position}(x).A_i > 4$. Tuple $s_{t_e}(12)$ does not change anything. In contrast, $s_{t_e}(8)$ splits an existing frame into two. Its event-time is between the timestamps of frame w_2 and therefore represents a gap between to frames.

A simple insertion can be performed for an out-of-order tuple that meets the threshold condition and contains an event-timestamp between the edges of the frame such as, for example, tuple $s_{t_e}(15)$ in Fig. 22. If a tuple belongs in front of the first tuple of a frame with respect to its event-time, such as $s_{t_e}(22)$, it shifts the start of the frame. Similar to this, an out-of-order tuple may shift the end of the frame to its event-time.

5.9.2 Delta frames

Specification Delta frames detect change of a particular attribute in the tuples. If the attribute value $s_{position}(x).A_i$ changes more than an amount t, a new frame is started (Eq. 27). Shorter frames then reflect periods of rapid change such as "spikes" in the data, whereas larger frames capture constant parts of the data. The difference (i.e., delta) is computed between the maximum value and the minimum value of an attribute in the tuples of a frame. If this value is greater (or smaller) than the specified value t, the current frame ends and a new one is started.

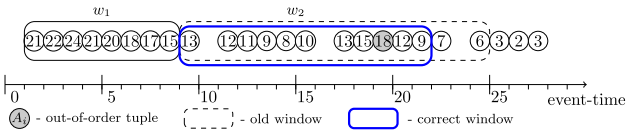

Fig. 24 Out-of-order processing of delta frames

$$b_i = s_{position}(x) \mid |s_{position}(x).A_i - s_{position}(y).A_i| \,\theta\, t \tag{27}$$

$$\theta \in \{<, \leq, >, \geq\} \tag{28}$$

Figure 23 depicts delta frames with the predefined delta value of 10. The maximum value of the first frame is 24 of $s_{t_e}(3)$. Tuple $s_{t_e}(9)$ with a value of 13 represents the new minimum value. Since the delta between 24 and 13 amounts 11 and is therefore greater than 10, the frame ends. The second frame holds a maximum value of 15. The difference with 3 amounts 12 and therefore the frame ends. Hence, the values do not need to be monotonically increasing or decreasing for this framing scheme.

Use-case In the same use-case as threshold frames (Sect. 5.9.1), delta frames can detect periods of change in the water density ρ [50]. A delta frame is started, when the condition $|s_{position}(x).\rho - s_{position}(y).\rho| < x$ is evaluated as true.

Classification (Table 3) This window type has the same classification as threshold frames (Sect. 5.9.1).

Out-of-order processing In this section, we make a suggestion how out-of-order processing could be implemented. Before inserting an out-of-order tuple, the condition for the tuples' attribute value has to be checked. A simple insertion operation can be performed, if this value meets the condition for its corresponding frame.

If the new attribute value constitutes a new maximum or minimum, the edges of the frames may change. This requires to check the condition with the values for all following tuples and shift the delta frames according to that. In Fig. 24, the tuple $s_{t_e}(19)$ with value 18 constitutes a new maximum for frame w_2. The condition check for $s_{t_e}(22)$ with value 7 is evaluated as false and the frame ends. This change affects following frames, so that they have to be shifted as well.

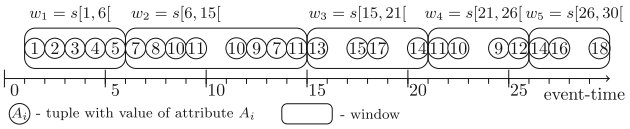

Fig. 25 Boundary frames with breakpoints at every multiple of the value 6 ($(n-1)6 < s_{position}(x).A_i \leq n6$)

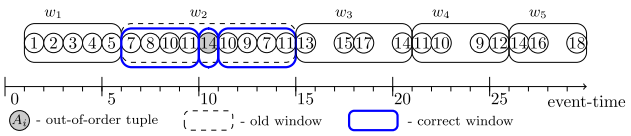

Fig. 26 Out-of-order processing of boundary frames

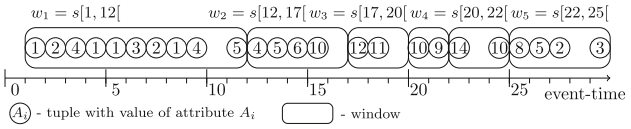

Fig. 27 Aggregate frames with a constant of 25 ($\sum_{s_{te}(x) \in w_i} (s_{te}(x).A_i) > 25$)

5.9.3 Boundary frames

Specification Boundary frames are specified by an attribute A_i and a set of breakpoints which are defined through (a multiple n) of a boundary b. If the attribute value crosses one of these points, the frame ends and a new one begins (Eq. 29).

$$b_i = s_{position}(x) \mid (n-1)b < s_{position}(x).A_i \leq nb \quad (29)$$

Boundary frames perform some kind of partitioning on the stream. In contrast to the partitioning based on a key, this is based on multiple predefined values for one or more attributes. For the boundary frames in Fig. 25, the breakpoints are defined as a multiple of 6. The condition for the first frame w_1 is $0 < s_{te}(x).A_i \leq 6$. Since the tuple $s_{t_e}(6)$ has a value of 7 and therefore crosses the breakpoint of 6, the first frame ends and a new one begins. The condition $6 < s_{te}(x).A_i \leq 12$ applies to the new frame w_2. The values of the tuples in the frame increase at first, but then decrease again. However, they never cross the breakpoint of 6 or the higher breakpoint of 12, so they all belong to the second frame. Since the value of the tuple $s_{t_e}(15)$ is greater than 12, a new frame begins. Due to the decreasing values in frame w_3, the breakpoint of 12 is passed again at $s_{t_e}(21)$ and a new frame starts. The tuples with the value 12 are still included in the frame w_4, due to $6 < s_{t_e}(x).A_i \leq 12$.

Use-case Boundary frames are useful to build heat maps showing the amount of time a soccer player spent in different areas of the field [50]. The field is divided into cells and the grid-lines form the breakpoints for the boundary frames. Since the position of the player has a x and a y component, the boundary frames are here two-dimensional (also called grid frames by Grossniklaus et al. [50]). The soccer pitch with 68x105 ms is divided into a 16x25 grid. The formal notation for the boundary frames is expressed as $(n-1)x < s_{te}(x).A_i \leq nx \wedge (n-1)y < s_{te}(x).A_i \leq ny$, where $0 < n \leq 16$, $0 < m \leq 25$, $x = 4.25$, and $y = 4.2$.

Classification (Table 3) This window type has the same classification as threshold frames (Sect. 5.9.1).

Out-of-order processing In this section, we make a suggestion how out-of-order processing could be implemented. For boundary frames, the first step is finding the existing frame to which the out-of-order tuple belongs with respect to its event-timestamp. If the value of the attribute meets the condition of this frame, the tuple can simply be inserted. Figure 26 shows that if the condition is not met, it requires to start a new frame.

5.9.4 Aggregate frames

Specification The last form are aggregate frames, where values of a predefined attribute A_i are aggregated within a frame. If this aggregate becomes greater (or smaller) than a constant c, the current frame ends and a new one starts (Eq. 30). The aggregate function is denoted as f_a.

$$b_i = s_{position}(x) \mid f_a(s_{position}(x).A_i) \lessgtr c \quad (30)$$

The size is adapted, so that shorter frames capture condition changes better and larger frames detect periods with less change. This frame already fulfills application needs, when the task is to output some specific aggregation. In Fig. 27, the aggregate function is a sum and the condition is $\sum_{s_{position}(x) \in w_i} (s_{position}(x).A_i) > 25$. The values are aggregated until the current tuple exceeds the threshold. In w_1 the sum aggregate amounts 24 at the tuple with timestamp 11. Hence, the aggregate that includes the next tuple value exceeds 25, and therefore, the tuple at timestamp 12 starts a new frame.

Use-case Aggregate frames could be used to analyze traffic data, where tuples often include the attributes speed and volume. The cumulative sum aggregation is applied on the attribute volume. After 25 cars, a new frame is started, formally $\sum_{s_{position}(x) \in w_i} (s_{position}(x).volume) > 25$.

Classification (Table 3) This window type has the same classification as threshold frames (Sect. 5.9.1).

Out-of-order processing In this section, we make a suggestion how out-of-order processing could be implemented.

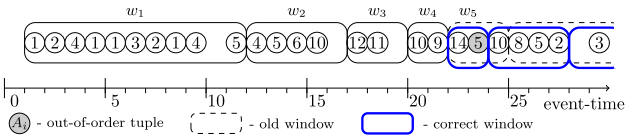

Fig. 28 Out-of-order processing of aggregate frames

The insertion of an out-of-order tuple in aggregate frames requires the recomputation of the cumulative aggregate from the out-of-order tuple on. In this case, tuples need to be kept in memory. It would be useful to save the cumulative aggregate for example at the end of each frame, otherwise it is necessary to compute it from the beginning or the last watermark on. Besides the recomputation, the constraint needs to be evaluated again for every tuple, since it is likely that the window edges change for every following window. Figure 28 exemplifies this with $s_{t_e}(23)$ that causes the end of frame w_5 to be shifted forward. This also results in an adjustment of the following frame.

5.10 Adaptive windowing

Specification Bifet at al. [21] proposed adaptive windowing (ADWIN). It represents an algorithm that automatically adapts the size of a sliding window according to the rate of change observed in the data stream. Typically, the size of the sliding window is predefined by the user and stays fixed during its application on the stream. Over time, streaming data is subject to changes such as concept drift and distribution change, which needs to be considered for choosing the right size of the window. Usually, the user has no information on the time scale of this change and therefore has to choose between a small size and a large size. Instead of defining a window size a priori, the ADWIN algorithm adjusts the size of the window dynamically with respect to changes. It extends the window size when no change is observed. In this case, there are many examples in the data, which increases the accuracy in periods of stability. In contrast, the algorithm shrinks the window size when data changes to accurately reflect the current distribution. The user does not have to deal with this trade-off anymore.

The algorithm uses a statistical test for different distributions in two subwindows that checks whether the observed average in both subwindows differs by more than the threshold (for details see Bifet et al. [21]). If it does, the distribution has changed and the algorithm will shrink the window by dropping old tuples. If no tuples are dropped, the window size grows. The algorithm can also keep the statistically optimal size by only dropping one tuple for one inserted tuple.

Use-case Bifet et al. used this algorithm for Naive Bayes Prediction and k-means clustering [21].

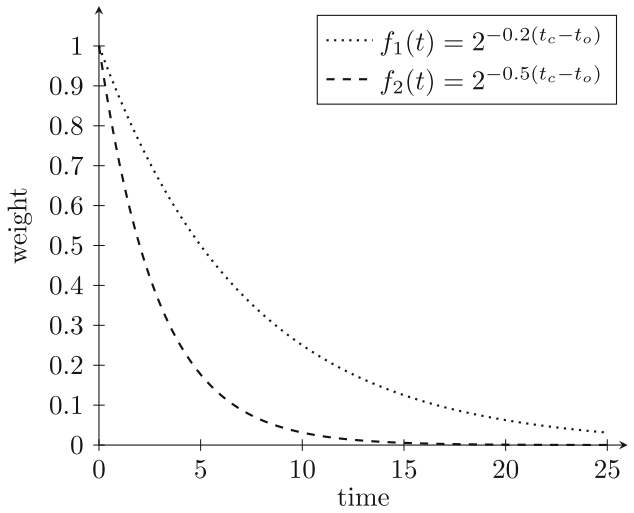

Fig. 29 Aging function of the damped window

Classification (Table 3) The window size adapts dynamically, so this window type creates variable-sized windows. Since ADWIN is based on sliding windows, the window are overlapping. The window size is defined in a count-based measure. The ADWIN algorithm implements a FCA and non-deterministic window type.

Out-of-order processing Since ADWIN implements a sliding window, the insertion of an out-of-order tuple resembles the count-based sliding window. An out-of-order tuple could cause a change in the average and hence may change the outcome of the statistical test, the algorithm might needs to change the size of the window. The handling of an out-of-order tuple thus needs to be considered in the algorithm.

5.11 Damped window

Specification The damped window spans the whole stream. A weight is assigned to each tuple depending on the arrival time of the tuple [70]. The highest weight is assigned to the current tuple, because it is the newest. Older tuples are not completely discarded but associated with lower weights [70]. Some aging function defines how this weight decreases exponentially over time. Typically, the exponential fading function (Eq. 21) is used as an aging function, expressing the current time as t_c and the event-time of the tuple as t_o [70]. λ denotes the fading factor, with $\lambda > 0$. With an increasing value of λ, the importance of historical data decreases compared to the current data [26]. Figure 29 illustrates the decreasing weight over time for one tuple. The function $f_1(t)$ represents the aging function with $\lambda = 0.2$. The aging function $f_2(t)$ shows that with a higher fading factor $\lambda = 0.5$ the

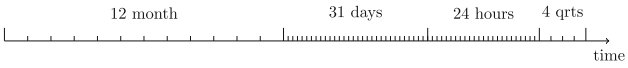

Fig. 30 The natural tilted time window

Fig. 31 The logarithmic tilted time window

weight assigned to a tuple decreases faster.

$$f(t) = 2^{-\lambda\,(t_c - t_o)} \tag{31}$$

Use-case This window type can be used to apply data mining techniques on streams. It is used in a variety of stream clustering algorithms (as shown in the survey by Mansalis et al. [70]).

Classification (Table 3) This window type is fixed-sized since it includes the whole history of the streams. The windows are non-overlapping and measured in time. The damped window requires backward-context to maintain the tuples that arrived in the past. Hence, it belongs to the category of FCF and deterministic window types.

Out-of-order processing The weight of the out-of-order tuple can be computed with the aging function based on its timestamp.

5.12 Tilted window

Specification The tilted window keeps the whole data of the stream in memory by storing summaries of it. For these summaries, time is registered at different levels of granularity [35]. The most recent tuples are stored at the finest granularity, since they represent the recent changes and are therefore the most interesting ones [70]. As time evolves, they are summarized to be stored at a coarser granularity. How coarse the scale gets, depends on the application needs [35]. The purpose of this window type is to reduce the total amount of data that is kept in memory [35].

Two techniques for tilted windows are the natural tilted time window [35,43] or the logarithmic tilted time window [3,43]. As shown in Fig. 30 in the natural tilted time frame the finest granularity is a quarter (i.e., 15 min). The most recent four quarters are accumulated to form the next window, which constitutes one hour [43]. Then, the last 24 h build one day, which are summarized to 31 days and after this to 12 month. Instead of $366 \times 24 \times 4 = 35136$ quarters per year, $4 + 24 + 31 + 12 = 71$ tilted windows need to be maintained per year [35].

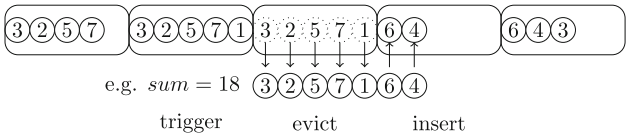

Fig. 32 Types of events according to Gedik [41]

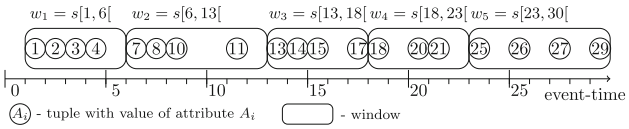

Fig. 33 The tumbling window with attribute-delta eviction policy and $\delta = 5$

The current window of a logarithmic tilted time window also holds the most recent tuples of 1 quarter. The quarters are then accumulated at an exponential rate of 2 [43]. The scale then includes windows that hold the last quarter, 2 quarters, 4 quarters, 8 quarters, 16 quarters, etc. (Fig. 31). This window schema is very space efficient, since it holds $log_2(365 \times 2 \times 4) + 1 \approx 17$ units of time for one year of data [43].

Synonyms Chen at al. [35] proposed this window type named as the tilt time frame.

Use-case This window type can be used for in data mining techniques, such as clustering [70] or frequent pattern mining [43].

Classification (Table 3) The tilted window can be classified as fixed-size, since the size of the windows is predefined through the time scale. The windows are specified through a time-based measure and do not overlap. The window edges are predefined through the time scale, which means it can be directly determined whether an arriving tuple starts a new window. Consequently, this window type is deterministic. However, backward-context is needed, since the windows in the past have to be accumulated to be at a coarser granularity. The tilted window therefore belongs to the category FCF.

Out-of-order processing An out-of-order tuple belongs into the window based on its event-time. The insertion depends on how the data is summarized to be at a coarser granularity.

5.13 Policy-based window

Specification The policy-based window refers to different configurations of tumbling and sliding windows denoted as policies which were proposed by Gedik [41]. The work aims to categorize window types and provide detailed operational semantics for window configurations [41].

Gedik differentiates into three operations which are performed while maintaining tuples in a window: *insertion*, *eviction*, and *trigger* (Fig. 32). Insertion takes place, when the arriving tuple is added to its respective window. A trigger event invokes the operator logic on a window. The corresponding policy that specifies when a window is ready for this processing is named trigger policy. An eviction removes one or more tuples from the window allowing newly incoming tuples to be inserted. The eviction policy specifies when a window is full and tuples are evicted. These "events are performed in different orders and in different ways depending on the window type and the eviction and trigger policies being applied" [41].

$$e_i = s_{position}(x) \mid s_{position}(x).A_i - s_{position}(o).A_i > \delta \tag{32}$$

$$b_j = e_i \; \forall w_i, \; w_j \mid j = i + 1 \tag{33}$$

$$p_{evict} = \{s_{position}(w) \mid s_{position}(w) \in w_i \wedge s_{position}(x).A_i - s_{position}(w).A_i > \delta\} \tag{34}$$

$$p_{trigger}(t) = s_{position}(x) \mid s_{position}(x).A_i - p_{trigger}(t-1).A_i > \delta \tag{35}$$

The eviction and trigger policies can be of type time, count, attribute-delta, or punctuation. This specifies when the trigger or eviction events happen and changes the configuration of a window. For the tumbling window, there exist only eviction policies, because the eviction operation is performed immediately after the window was processed. This leads to the differentiation into the time-based, count-based, and attribute-delta-based eviction policies as well as the punctuation-based eviction policy, which only applies to tumbling windows. Each of these policies has their corresponding order of eviction and insertion events (for details of the event order for the tumbling window see Gedik [41]).

The *time-based eviction policy* resembles the tumbling window with a time-based measure (Sect. 5.1). After the specified time has elapsed, the window is considered to be full and is processed. Once processing is complete, all tuples are evicted. This happens independently of tuple arrivals. The *count-based eviction policy* for tumbling windows corresponds to the tumbling window with a count-based measure (Sect. 5.1). A newly arriving tuple is inserted into the window. If the window then contains the predefined number of tuples, processing is performed and all tuples are removed. For a tumbling window with the *attribute-delta eviction policy*, the difference of attribute values of a specified delta attribute A_i is measured between the oldest tuple $s_{position}(o)$ of a window and the arriving tuple $s_{position}(x)$ (Eq. 32). If the difference exceeds a specified delta threshold δ, the current window is processed and tuples are evicted. Figure 33 depicts a tumbling window with threshold $\delta = 5$. The difference between the attribute value 7 of tuple s_6 and the attribute value 1 of tuple s_1 amounts 6 and therefore exceeds the threshold of 5. All tuples are evicted from w_1, before s_6 is inserted into the next window (Eq. 33). This requires the delta attribute to be non-decreasing. It is typically monotonically increasing such as a timestamp attribute [56]. The *punctuation-based eviction policy* resembles non-overlapping window punctuations (Sect. 5.5).

For the sliding window, an eviction policy can differ from the trigger policy, because trigger and eviction events do not overlap. The resulting three types of trigger or eviction policies can be combined creating nine different possible configurations for sliding windows. For each of these policy combinations, the order of the trigger, eviction and insertion events differs (details are specified in the original publication [41]). Eviction and trigger policy can be referred to as the size and slide parameter of the sliding window (Sect. 5.2). However, different combinations of policies enable configurations of the sliding window that cannot be expressed through a specification with window size and slide.

The *time-based eviction policy* for sliding windows determines how long a tuple belongs to a window before it is evicted. This is independent from tuple insertions. Similarly, the *time-based trigger policy* defines a time period after which the operator logic is applied onto the window, even if no tuple arrived within that time (i.e., trigger is independent of insertion). A sliding window that follows the *count-based eviction policy* keeps a maximum number of tuples that is specified. Evictions are performed before inserting new tuples. The *count-based trigger policy* determines after how many inserted tuples the trigger event happens. Before processing a window, tuples are evicted if necessary. When applying the *attribute-delta eviction policy* on sliding windows, the tuples that are kept in the window are determined by the difference between their values for the delta attribute A_i and the attribute value of newly arrived tuple. Every tuple $s_{position}(w)$ in window w_i for which the difference exceeds the defined delta threshold δ, is evicted before the new tuple $s_{position}(x)$ is inserted into the window (Eq. 34). Insertions for this policy take place after evictions. The *attribute-delta trigger policy* computes the difference between the arriving tuple $s_{position}(x)$ and the last tuple that triggered the processing $p_{trigger}(t-1)$ (Eq. 35). If it is greater than the delta threshold, the window is triggered again. Evictions take place after the window was processed.

Synonyms Carbone et al. referred to sliding windows where size and slide are defined in different measures as multi-type [29]. Traub et al. used the term multi-measure window for a window that combines count- and time-measure [93].

Use-case We describe use-cases in previous sections for the time-based eviction policy and count-based eviction policy of

the tumbling window (Sect. 5.1), punctuation-based eviction policy of the tumbling window (Sect. 5.5), time-based eviction and trigger policy of the sliding window (i.e., time-based measure), and count-based eviction and trigger policy (i.e., count-based measure) of the sliding window (Sect. 5.2). The time-based eviction policy combined with the count-based trigger policy in a sliding window resembles the slide-by-tuple window (Sect. 5.8). These policies can be combined the other way around as the count-based eviction policy and time-based trigger policy of a sliding window, for instance, to output the average over the last 20 tuples every 2 min. A use-case in the context of machine monitoring could be to output the average temperature of the last 20 measurements of a machine every 2 min to observe the heat build-up.

Next, we will provide use-cases for the attribute-delta policies. Examples of monotonically increasing attributes for the attribute-delta policy beyond timestamps are invoice numbers or mileage. An attribute-delta policy-based tumbling window can be based on the invoice number as the specified delta attribute and the value 1000 as the specified delta threshold. If the difference between the first and the last invoice number of a tumbling window instance exceeds 1000, the average invoice amount is computed. In a sliding window, the attribute-delta policy can also be used as a trigger policy combined with for example the count-based eviction policy. For instance, this sliding window keeps 50 tuples, while the attribute-delta policy makes sure that the window is triggered whenever the traveled mileage exceeds the delta of 25 kms. In a use-case of a car, this window configuration can calculate the average over the last 50 measurements of tire pressure every 25 kms.

Classification (Table 3) The time-based and count-based policies for the policy-based tumbling window can be classified in exactly the same categories as the tumbling window (Sect. 5.1). The difference between those two policies is the window measure, which is time-based for the time policy and count-based for the count policy. All policies for the tumbling window can be classified as non-overlapping. The attribute-delta eviction policy produces variable-sized windows because the window edges are not known beforehand and depend on the attribute values of a tuple. It requires to compare the oldest and the arriving tuple to determine whether a new window begins, classifying as a non-deterministic and FCA window type. The punctuation-based policy exhibits the same classifications as the punctuation-based window (Sect. 5.5).

In contrast to the tumbling window, all of the policies for the policy-based sliding window classify as overlapping. The attribute-delta trigger policy produces variable-sized windows, because the evictions happen after the trigger. The window is first processed and then the tuples are evicted. A sliding window with the attribute-delta eviction policy is non-FIFO, since instead of the oldest tuples, tuples with values whose difference exceeds the delta threshold are evicted. With the arrival of a tuple cannot be determined immediately, whether a new window begins with this tuple or not. This requires forward-context to derive the window edges of each window and therefore belongs in the category of FCA and non-deterministic. The attribute-delta trigger policy evicts the tuples in order, resulting in a FIFO window type. However, to trigger the windows, tuples need to be processed, which again requires forward-context and leads this policy to be non-deterministic and FCA.

Out-of-order processing In this section, we make a suggestion how out-of-order processing could be implemented. For time-based policies, an out-of-order tuple has to be inserted into the window with respect to its timestamp. Count-based policies require an adaptation of the window edges, since late tuples change the count of subsequent tuples. The attribute-delta policy requires to store attribute values of the tuples which were relevant for the trigger or eviction to happen for a window. For instance, the attribute-delta tumbling window would require to keep the oldest tuple of each window. Then the difference between the out-of-order tuple and these stored values can be computed. The out-of-order tuple can be inserted into a window, if the deviation does not exceed the threshold. The different combinations of policies require a combination of these out-of-order processing schemes leading to an increased complexity for out-of-order streams.

6 Discussion

In this section, we examine SPSs with regard to supported window types (Sect. 6.1), discuss different window aggregation concepts (Sect. 6.2), and summarize our findings (Sect. 6.3).

6.1 Systems

In this section, we are going to examine which SPSs support the studied window types. Our results are listed in Table 4. We found that all the investigated systems support the tumbling window and the sliding window. These are the most commonly known and used window types. The session window is also supported by many of the systems. Despite the complexity of the policy-based window, it is implemented by multiple systems: Google Cloud Dataflow [47], IBM Streams [60], Apache Beam [12], and Apache Flink [14]. IBM InfoSphere Streams supports the four types of eviction and trigger policies exactly like described by Gedik [60]. Flink implements window evictors and triggers of the types count, time, and delta that resemble the policy-based window [14]. Consequently, Flink supports the most window types (e.g., sliding,

Table 4 Overview over window types supported by investigated systems

Window type[1]	Borealis [1]	STREAM [16]	TelegraphCQ [33]	Naiad [74]	Trill [31,32]	Amazon Kinesis Data Analytics [7]	Google Cloud Dataflow [46,48]	IBM Streams [59]	Microsoft Azure Stream Analytics [71]	Microsoft Stream Insight [6,49]	Apache Beam [4,13]	Apache Flink [27]	Apache Samza [75]	Apache Spark [89,96]	Apache Storm [90]	Apache Heron [39,64]	Scotty [92,93,94]
Tumbling Window [28,76,93]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Sliding Window [4,28,93]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Fixed-Band Window [76]			✓														✓
Landmark Window [76]			✓									✓					
Punctuation-based Window [41,56]								✓				✓					✓
Session Window [4,92,93]					✓		✓		✓		✓	✓	✓	✓			✓
Snapshot Window [5,49,72,73]										✓							
Slide-by-Tuple Window [67]								✓				✓					✓
Threshold Frames [50]																	✓
Policy-based Window [41,56]							✓	✓			✓	✓					
User-defined Window							✓				✓	✓	✓	✓			✓

[1] Delta Frames, Boundary Frames, Aggregate Frames, Adaptive Windowing, Damped Window, and Tilted Window are not listed in this table since they are not supported out-of-the-box. These window types may be implemented as custom window types by users themselves.

tumbling, session, punctuation, landmark, delta-based windows) [15,27]. The slide-by-tuple window can be supported by systems that provide the policy-based window, since this allows to combine the different window measures count and time. Microsoft StreamInsight is the only system that supports the snapshot window [5,72,73]. The window types delta frames, boundary frames, aggregate frames, adaptive windowing, damped window, and tilted window are not provided by any of the investigated systems. SPSs often enable user-defined windows (also called custom windows), that allow the user to extend them with additional window types (e.g., Google Cloud Dataflow, Beam, Flink, Samza, Spark) [10].

As shown in Table 4, the systems developed in academic research generally support less window types. However, some of these systems were developed several years ago when there were not as many window types. The focus of these systems may not be to provide extensive functionality, but rather to serve a research purpose. The number of supported window types of commercial systems differs. For instance, IBM Streams and Google Cloud Dataflow provide a rich functionality by implementing trigger and eviction functions [47,60], while other commercial systems do not offer as many window types such as Amazon Kinesis Data Analystics [8]. The open-source systems we investigated typically support the tumbling window, sliding window, and session window. In contrast to most of the academic and commercial systems, they provide the opportunity to extend them with user-defined windows. However, most systems do not implement more complex and data-driven window types such as the snapshot window, frames, or adaptive windowing. Table 4 includes Scotty, because it has connectors to multiple systems and provides implementations of various window types. Example implementations of the window types can be found in the systems and the listed publications.

6.2 Window aggregation implementation

This section presents different concepts that have been proposed to enable efficient window aggregation. Traub et al. [93,94] studied different window aggregation concepts with respect to their throughput, latency, and memory consumption. A *tuple buffer* simply stores the incoming tuples and iterates them to perform an aggregation when a window ends (i.e., lazy computation). *Aggregate trees* [17,84] maintain partial aggregates in a tree structure, which are then combined to final aggregates and allow for a lower latency (e.g., B-Int [17], FlatFAT [84], FlatFIT [80], Slider [20], FIBA [83], LightSaber [87]). Additionally, this enables aggregate sharing among overlapping windows. Another window aggregation approach is *bucketing* [67–69], where one bucket stores the content of one window (e.g., Li et al. [67,69], AdaptWID [68]). An incoming tuple is assigned to all buckets of its corresponding windows. Buckets can store individual tuples (i.e., tuple buckets) with the drawback that some tuples are then redundant in buckets of overlapping windows. It is also possible to drop tuples and maintain only partial aggregates in aggregate buckets to reduce the memory footprint. Aggregates can be incrementally updated to reduce the latency [84]. The bucket-per-window approach creates a throughput bottleneck for overlapping windows because multiple aggregation operations are required for each bucket to which the input tuple is assigned. Additionally, it does not allow for sharing those aggregates between overlapping or concurrent windows. *Stream slicing* [63,66] avoids the overhead of these redundant computations by diving the stream into non-overlapping subsets of data (i.e., slices) (e.g., Panes [66], Pairs [63], Cutty [29], Scotty [92–94], Disco [19], Parallel Boundary Aggregator [100], SABER [62]). For each slice, one partial aggregate is computed and stored. The final aggregate of a window is calculated by combining partial results of several slices. Partial aggregates of slices can be shared among overlapping windows and concurrent window queries. Other algorithms have been proposed that use different concepts, such as stacks (e.g., Two-Stacks [82], DABA [82], SlideSide [88], Hammer Slide [86]), an array combined with an aggregate tree (e.g., CBiX [24], CPiX [25]), or a double-ended queue (e.g., SlickDeque [81]). Most research on window aggregation optimization was conducted on tumbling and sliding windows, since these are the most commonly used window types. Li et al. [67] proposed an algorithm for processing slide-by-tuple windows. Tree-based techniques are generally applicable. The stream slicing approach has been generalized to support all window types (e.g., Cutty [29], Scotty [92–94]), while providing high-performance guarantees. To further reduce latencies, some techniques store aggregate trees on top of slices (e.g., FlatFIT [80]) [93].

6.3 Summary

The classifications are important to derive implementation requirements and to take decisions for efficient window aggregation, for instance, whether individual tuples need to be stored or not [93]. Common window types such as tumbling windows and sliding windows typically have a fixed-size and classify as CF. Window types with these classifications often allow an easier handling of out-of-order tuples. In general, more complex window types require context. For instance, session windows and snapshot windows can be classified as deterministic window types but have a higher complexity since they require backward-context. Additionally, more complex windows are often data-driven and dynamically adapt their size resulting in variable-size windows. However, windowing semantics become even more complex for window types that require forward-context and are non-deterministic, for instance, Frames and ADWIN. Com-

plex window semantics also lead to an increased complexity for the out-of-order processing. For context-aware windows, out-of-order tuples change the context which makes modifying window edges necessary.

7 Related work

In this section, we present related work, which we did not discuss before. Our survey focuses on the specification of window types and shows which systems support these windows types. In our survey, we discussed most related work on window types and streaming systems in previous sections. This section discusses related surveys, approximate stream processing, and modern hardware.

Related surveys Patroumpas et al. [76] formally specified count-based windows, time-based tumbling, as well as sliding windows, landmark windows, and fixed-band windows. Furthermore, the work introduced the earliest categorization of window types based on their properties. In Sect. 5.13, we included the policies for tumbling and sliding windows of Gedik [41], since they provide additional specifications for the tumbling and sliding window. These window types and policies were also described by Hirzel et al. [56]. Li et al. [67] introduced and provided a detailed view on the slide-by-tuple window. Moreover, they proposed the classification of window types based on their context which we adopted in the classification section. Our catalog covers a number of different window types beyond these mentioned in the named surveys. While these publications, except for Li et al. [67], do not cover out-of-order processing, we provide a description for handling out-of-order tuples with respect to each window type. Additionally, we summarize multiple classifications that allow for grouping window types with respect to performance characteristics and implementation requirements, and we categorize every window type. Hirzel et al. [57] summarized that the aggregation algorithm should be selected depending on the aggregation operation, latency requirements, window type, sharing requirements, and out-of-order processing to avoid performance loss.

Complementary to our survey, Lal et al. [65] compared window types on the basis of CPU utilization, memory consumption, time efficiency, and operation compatibility. Summarizing the earliest work in stream processing, Golab et al. [45] reviewed streaming applications, data models, query languages, streaming operators including sliding windows, as well as query processing and optimization techniques. Hirzel et al. [58] conducted a survey for stream processing optimizations. The tutorial of Schneider et al. [79] extends this survey by fission optimization which exploits data-parallelism on multiple cores to achieve a high throughput. Giouroukis et al. [44] surveyed algorithms for adaptive filtering and adaptive sampling for real-time distributed sensor networks in the Internet of Things. In addition to these existing surveys, we contribute an extensive survey and analyses of various window types with respect to their out-of-order processing, synonyms, use-cases, implementation, and systems which support them.

Approximate stream processing In this survey, we focused on window aggregation with precise results. Approximate stream processing can be used to further improve performance if precise results are not essentially needed. Synopses (i.e., samples histograms, wavelets, sketches) provide approximate aggregates which can be used for stream processing to answer aggregation queries with lower latency [40]. Cormode and Muthukrishnan [36] proposed the Count-Min Sketch for streaming queries which can be used to, e.g., answer range queries, compute quantiles, or find heavy hitters (i.e., frequent items). The load shedding technique of Tatbul et al. [85] mitigates overload of a SPS by dropping tuples of less importance. Chen and Zhang [34] presented two algorithms for linear bias-aware sketches that are applicable to streaming data and provide improved error guarantees. The hyperloglog algorithm [38] estimates the cardinality of data streams and adapts to sliding windows. The online reservoir sampling algorithm StreamApprox [78] answers queries with a sliding window in a pipelined stream processing model. The framework Condor [77] generates windowed synopses for streaming jobs that allow an efficient distributed computation. Condor supports tumbling, sliding, and session windows and utilizes bucketing [67] or general stream slicing [92–94] as an aggregation technique.

Modern hardware Window aggregation on modern hardware by Zeuch et al. [98] utilizes a lock-free double-buffer implementation where one buffer stores incoming tuples and non-active buffers store the previous window results. It achieves a high throughput since non-active buffers compute the final aggregates and output the results without deferring the processing of the incoming tuples. Grizzly [51] performs query compilation to enable a highly efficient execution on modern hardware. Query code is generated with respect to the special requirements for streaming workloads which include different window types, window measures, and window functions. With a novel processing model, Slash [37] accelerates distributed windowing by leveraging Remote Direct Memory Access (RDMA) hardware that allows high throughput and low-latency data transfer. The data-management platform for the Internet-of-Things NebulaStream [97,99] combines general stream slicing [93], adaptive query compilation [51], and data sharing among multiple streaming queries [61]. Additionally, it enables the user to define window types and window functions with the technique Babelfish [52] that accelerates UDF-based operators.

8 Conclusion

Splitting data streams into windows is a key functionality of SPSs to enable operators such as aggregations and joins. Different window types are proposed in the literature of stream processing systems, window aggregation techniques, and other related topics. Information is fragmented across a large number of publications. This paper presents the first comprehensive survey of window types. We extensively reviewed existing literature on windowing for stream processing as well as several documentations of stream processing systems. To this end, we collected information on 16 different window types. We standardized the notation to unify the formal definitions of window types and to simplify the descriptions in future work. We investigated various classifications that exist in literature and classify every window type according to them. For 7 out of these 16 window types, we suggested approaches for the out-of-order processing. To resolve inconsistencies in the terminology, we list existing synonyms across the resources and describe contradictory definitions for the same window type. We summarize all window types, their classifications, and a short description in a table. Additionally, we provide a matrix of stream processing systems that reveals the deficits of existing systems in providing all window types and especially shows the lack of data-driven windows. This matrix can serve as orientation when choosing a suitable system based on provided window types. With this survey, we aim to provide researchers and practitioners an overview of the many window types in literature that supports them in implementing general-purpose systems. Our paper should enable them to provide systems, frameworks, and window aggregation techniques that are compatible with a broad range of window types. It also represents a means for system users to find the right window type for their use-cases.

Funding Open Access funding enabled and organized by Projekt DEAL. This work was funded by German Research Foundation (MA4662-5 and 410830482), German Federal Ministry for Education and Research as BIFOLD - Berlin Institute for the Foundations of Learning and Data (ref. 01IS18025A and ref. 01IS18037A) and Software Campus (01IS17052).

References

1. Abadi, D.J., Ahmad, Y., Balazinska, M., Cetintemel, U., Cherniack, M., Hwang, J.H., Lindner, W., Maskey, A., Rasin, A., Ryvkina, E., et al.: The design of the borealis stream processing engine. In: CIDR, vol. 5, pp. 277–289 (2005)
2. Abadi, D.J., Carney, D., Cetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., Zdonik, S.: Aurora: a new model and architecture for data stream management. VLDB J **12**(2), 120–139 (2003)
3. Aggarwal, C.C., Philip, S.Y., Han, J., Wang, J.: A framework for clustering evolving data streams. In: VLDB, pp. 81–92 (2003)
4. Akidau, T., Bradshaw, R., Chambers, C., Chernyak, S., Fernández-Moctezuma, R.J., Lax, R., McVeety, S., Mills, D., Perry, F., Schmidt, E., Whittle, S.: The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. PVLDB **8**(12), 1792–1803 (2015)
5. Ali, M., Chandramouli, B., Goldstein, J., Schindlauer, R.: The extensibility framework in microsoft streaminsight. In: ICDE, pp. 1242–1253. IEEE (2011)
6. Ali, M.H., Gerea, C., Raman, B.S., Sezgin, B., Tarnavski, T., Verona, T., Wang, P., Zabback, P., Ananthanarayan, A., Kirilov, A., et al.: Microsoft cep server and online behavioral targeting. PVLDB **2**(2), 1558–1561 (2009)
7. Amazon Kinesis Data Analytics Documentation. https://docs.aws.amazon.com/kinesis/index.html. Accessed 20 Oct 2021
8. Amazon Kinesis Data Analytics Documentation. Windowed Queries. https://docs.aws.amazon.com/kinesisanalytics/latest/dev/windowed-sql.html. Accessed 20 Oct 10.2021
9. Andrade, H.C.M., Gedik, B., Tugara, D.S.: Fundamentals of Stream Processing: Application Design, Systems, and Analytics. Cambridge University Press, Cambridge (2014)
10. Apache Beam Documentation. Beam Capability Matrix. https://beam.apache.org/documentation/runners/capability-matrix/where-in-event-time/. Accessed 18 Oct 2021
11. Apache Beam Documentation. Provided windowing functions. https://beam.apache.org/documentation/programming-guide/#provided-windowing-functions. Accessed 18 Oct 2021
12. Apache Beam Documentation. Triggers. https://beam.apache.org/documentation/programming-guide/#triggers. Accessed 18 Oct 2021
13. Apache Beam. https://beam.apache.org/. Accessed 18 Oct 2021
14. Apache Flink Streaming Data API. Evictors. https://ci.apache.org/projects/flink/flink-docs-release-1.13/docs/dev/datastream/operators/windows/#evictors. Accessed 18 Oct 2021
15. Apache Flink Documentation. Windows. https://ci.apache.org/projects/flink/flink-docs-stable/dev/stream/operators/windows.html. Accessed 18 Oct 2021
16. Arasu, A., Babcock, B., Babu, S., Cieslewicz, J., Datar, M., Ito, K., Motwani, R., Srivastava, U., Widom, J.: Stream: The Stanford data stream management system. In: Data Stream Management, pp. 317–336. Springer, Berlin (2016)
17. Arasu, A., Widom, J.: Resource sharing in continuous sliding-window aggregates. In: VLDB, pp. 336–347 (2004)
18. Begoli, E., Akidau, T., Hueske, F., Hyde, J., Knight, K., Knowles, K.: One SQL to rule them all—an efficient and syntactically idiomatic approach to management of streams and tables. In: SIGMOD, pp. 1757–1772 (2019)
19. Benson, L., Grulich, P.M., Zeuch, S., Markl, V., Rabl, T.: Disco: Efficient distributed window aggregation. In: EDBT, vol. 20 (2020)
20. Bhatotia, P., Acar, U.A., Junqueira, F.P., Rodrigues, R.: Slider: incremental sliding window analytics. In: ACM/IFIP/USENIX Middleware, pp. 61–72 (2014)

21. Bifet, A., Gavaldà, R.: Learning from time-changing data with adaptive windowing. In: SIAM SDM, pp. 443–448 (2007)
22. Blee-Goldmann, A.S., Thomas, L.: Kip-450: sliding window aggregations in the DSL. https://cwiki.apache.org/confluence/display/KAFKA/KIP-450%3A+Sliding+Window+Aggregations+in+the+DSL (2020)
23. Botan, I., Derakhshan, R., Dindar, N., Haas, L., Miller, R.J., Tatbul, N.: Secret: a model for analysis of the execution semantics of stream processing systems. PVLDB **3**(1–2), 232–243 (2010)
24. BOU, S., KITAGAWA, H., AMAGASA, T.: Cbix: incremental sliding-window aggregation for real-time analytics over out-of-order data streams (2018)
25. Bou, S., Kitagawa, H., Amagasa, T.: Cpix: real-time analytics over out-of-order data streams by incremental sliding-window aggregation. IEEE TKDE (2021)
26. Cao, F., Estert, M., Qian, W., Zhou, A.: Density-based clustering over an evolving data stream with noise. In: SIAM SDM, pp. 328–339. SIAM (2006)
27. Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., Tzoumas, K.: Apache flink^{TM}: Stream and batch processing in a single engine. In: IEEE CS **36**(4) (2015)
28. Carbone, P., Katsifodimos, A., Haridi, S.: Stream window aggregation semantics and optimization. In: Encyclopedia of Big Data Technologies (2019)
29. Carbone, P., Traub, J., Katsifodimos, A., Haridi, S., Markl, V.: Cutty: aggregate sharing for user-defined windows. In: CIKM, pp. 1201–1210 (2016)
30. Carney, D., undefinedetintemel, U., Cherniack, M., Convey, C., Lee, S., Seidman, G., Stonebraker, M., Tatbul, N., Zdonik, S.: Monitoring streams: a new class of data management applications. In: Proceedings of the 28th International Conference on Very Large Data Bases, pp. 215–226. VLDB Endowment (2002)
31. Chandramouli, B., Goldstein, J., Barnett, M., DeLine, R., Fisher, D., Platt, J.C., Terwilliger, J.F., Wernsing, J.: Trill: a high-performance incremental query processor for diverse analytics. PVLDB **8**(4), 401–412 (2014)
32. Chandramouli, B., Goldstein, J., Barnett, M., DeLine, R., Fisher, D., Platt, J.C., Terwilliger, J.F., Wernsing, J.: The trill incremental analytics engine. Technical report, MSR-TR-2014-54 (2014)
33. Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M.J., Hellerstein, J.M., Hong, W., Krishnamurthy, S., Madden, S.R., Reiss, F., Shah, M.A.: Telegraphcq: continuous dataflow processing. In: SIGMOD, p. 668 (2003)
34. Chen, J., Zhang, Q.: Bias-aware sketches. arXiv preprint arXiv:1610.07718 (2016)
35. Chen, Y., Dong, G., Han, J., Wah, B.W., Wang, J.: Multi-dimensional regression analysis of time-series data streams. In: VLDB, pp. 323–334. Elsevier, Amsterdam (2002)
36. Cormode, G., Muthukrishnan, S.: An improved data stream summary: the count-min sketch and its applications. J. Algorithms **55**(1), 58–75 (2005)
37. Del Monte, B., Zeuch, S., Rabl, T., Markl, V.: Rethinking stateful stream processing with rdma. In: SIGMOD (2022) (to appear)
38. Flajolet, P., Fusy, É., Gandouet, O., Meunier, F.: Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. In: Discrete Mathematics and Theoretical Computer Science, pp. 137–156 (2007)
39. Fu, M., Agrawal, A., Floratou, A., Graham, B., Jorgensen, A., Li, M., Lu, N., Ramasamy, K., Rao, S., Wang, C.: Twitter heron: towards extensible streaming engines. In: ICDE, pp. 1165–1172. IEEE (2017)
40. Garofalakis, M.N., Gibbons, P.B.: Approximate query processing: taming the terabytes. In: VLDB vol. 10, pp. 645927–672356 (2001)
41. Gedik, B.: Generic windowing support for extensible stream processing systems. Softw.: Pract. Exp. **44**(9), 1105–1128 (2014)
42. Gehrke, J., Korn, F., Srivastava, D.: On computing correlated aggregates over continual data streams. ACM SIGMOD Record **30**(2), 13–24 (2001)
43. Giannella, C., Han, J., Pei, J., Yan, X., Yu, P.S.: Mining frequent patterns in data streams at multiple time granularities. Next Gener. Data Min. **212**, 191–212 (2003)
44. Giouroukis, D., Dadiani, A., Traub, J., Zeuch, S., Markl, V.: A survey of adaptive sampling and filtering algorithms for the internet of things. In: DEBS, pp. 27–38 (2020)
45. Golab, L., Özsu, M.T.: Issues in data stream management. SIGMOD Rec. **32**(2), 5–14 (2003)
46. Google Cloud Dataflow. https://cloud.google.com/dataflow. Accessed 20 Oct 2021
47. Google Cloud Dataflow Documentation. Triggers. https://cloud.google.com/dataflow/docs/concepts/streaming-pipelines#triggers. Accessed 21 Oct 2021
48. Google Cloud Dataflow Documentation. Streaming pipelines. https://cloud.google.com/dataflow/docs/concepts/streaming-pipelines. Accessed 20 Oct 2021
49. Grabs, T., Schindlauer, R., Krishnan, R., Goldstein, J., Fernández, R.: Introducing microsoft streaminsight. Technical report (2009)
50. Grossniklaus, M., Maier, D., Miller, J., Moorthy, S., Tufte, K.: Frames: data-driven windows. In: DEBS, pp. 13–24. ACM (2016)
51. Grulich, P.M., Sebastian, B., Zeuch, S., Traub, J., Bleichert, J.v., Chen, Z., Rabl, T., Markl, V.: Grizzly: efficient stream processing through adaptive query compilation. In: SIGMOD, pp. 2487–2503 (2020)
52. Grulich, P.M., et al.: Babelfish: efficient execution of polyglot queries. In: VLDB. VLDB endowment (2021)
53. Guha, S., Koudas, N.: Approximating a data stream for querying and estimation: algorithms and performance evaluation. In: ICDE, pp. 567–576. IEEE (2002)
54. Gulisano, V., Jimenez-Peris, R., Patino-Martinez, M., Soriente, C., Valduriez, P.: Streamcloud: an elastic and scalable data streaming system. IEEE TPDS **23**(12), 2351–2365 (2012)
55. Gulisano, V., Nikolakopoulos, Y., Cederman, D., Papatriantafilou, M., Tsigas, P.: Efficient data streaming multiway aggregation through concurrent algorithmic designs and new abstract data types. ACM TOPC **4**(2), 1–28 (2017)
56. Hirzel, M., Andrade, H., Gedik, B., Kumar, V., Losa, G., Nasgaard, H., Soule, R., Wu, K.L.: Spl stream processing language specification. Technical report IBM (2009)
57. Hirzel, M., Schneider, S., Tangwongsan, K.: Tutorial: sliding-window aggregation algorithms. In: DEBS, pp. 11–14 (2017)
58. Hirzel, M., Soulé, R., Schneider, S., Gedik, B., Grimm, R.: A catalog of stream processing optimizations. ACM CSUR (2014). https://doi.org/10.1145/2528412
59. IBM Streams Documentation. https://www.ibm.com/docs/en/streams/5.5?topic=welcome-introduction-streams. Accessed 19 Oct 2021
60. IBM Streams Documentation. Window clause. https://www.ibm.com/docs/en/streams/5.5?topic=invocations-window-clause. Accessed 19 Oct 2021
61. Karimov, J., Rabl, T., Markl, V.: Astream: Ad-hoc shared stream processing. In: SIGMOD, pp. 607–622 (2019)
62. Koliousis, A., Weidlich, M., Castro Fernandez, R., Wolf, A.L., Costa, P., Pietzuch, P.: Saber: window-based hybrid stream processing for heterogeneous architectures. In: SIGMOD, pp. 555–569 (2016)
63. Krishnamurthy, S., Wu, C., Franklin, M.J.: On-the-fly sharing for streamed aggregation. In: SIGMOD, pp. 623–634 (2006)
64. Kulkarni, S., Bhagat, N., Fu, M., Kedigehalli, V., Kellogg, C., Mittal, S., Patel, J.M., Ramasamy, K., Taneja, S.: Twitter heron: stream processing at scale. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, pp. 239–250 (2015)

65. Lal, D.K., Suman, U.: A survey of real-time big data processing algorithms. In: Reliability and Risk Assessment in Engineering, pp. 3–10. Springer, Berlin (2020)
66. Li, J., Maier, D., Tufte, K., Papadimos, V., Tucker, P.A.: No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. SIGMOD Record **34**(1), 39–44 (2005)
67. Li, J., Maier, D., Tufte, K., Papadimos, V., Tucker, P.A.: Semantics and evaluation techniques for window aggregates in data streams. In: SIGMOD, pp. 311–322 (2005)
68. Li, J., Tufte, K., Maier, D., Papadimos, V.: Adaptwid: an adaptive, memory-efficient window aggregation implementation. IEEE Internet Comput. **12**(6), 22–29 (2008)
69. Li, J., Tufte, K., Shkapenyuk, V., Papadimos, V., Johnson, T., Maier, D.: Out-of-order processing: a new architecture for high-performance stream systems. PVLDB **1**(1), 274–288 (2008)
70. Mansalis, S., Ntoutsi, E., Pelekis, N., Theodoridis, Y.: An evaluation of data stream clustering algorithms. Stat. Anal. Data Min.: ASA Data Sci. J. **11**, 167–187 (2018)
71. Microsoft Azure Stream Analytics Documentation. Introduction to Stream Analytics windowing functions. https://docs.microsoft.com/en-us/azure/stream-analytics/stream-analytics-window-functions. Accessed 02 Feb 2020
72. Microsoft Blog. Windows in StreamInsight: Hopping vs. Snapshot. https://blogs.msdn.microsoft.com/streaminsight/2010/12/15/windows-in-streaminsight-hopping-vs-snapshot/. Accessed 02 Feb 2020
73. Microsoft StreamInsight Developer's Guide. Using Event Windows. https://docs.microsoft.com/en-us/previous-versions/sql/streaminsight/ee842704(v=sql.111). Accessed 18 Oct 2021
74. Murray, D.G., McSherry, F., Isaacs, R., Isard, M., Barham, P., Abadi, M.: Naiad: a timely dataflow system. In: SOSP ACM, pp. 439–455 (2013)
75. Noghabi, S.A., Paramasivam, K., Pan, Y., Ramesh, N., Bringhurst, J., Gupta, I., Campbell, R.H.: Samza: stateful scalable stream processing at linkedin. PVLDB **10**(12), 1634–1645 (2017)
76. Patroumpas, K., Sellis, T.: Window specification over data streams. In: EDBT, pp. 445–464 (2006)
77. Poepsel-Lemaitre, R., Kiefer, M., von Hein, J., Quiané-Ruiz, J.A., Markl, V.: In the land of data streams where synopses are missing, one framework to bring them all. VLDB Endowment **14**(10), 1818–1831 (2021)
78. Quoc, D.L., Chen, R., Bhatotia, P., Fetzer, C., Hilt, V., Strufe, T.: Streamapprox: approximate computing for stream analytics. In: ACM/IFIP/USENIX Middleware, pp. 185–197 (2017)
79. Schneider, S., Hirzel, M., Gedik, B.: Tutorial: stream processing optimizations. In: DEBS, pp. 249–258 (2013)
80. Shein, A.U., Chrysanthis, P.K., Labrinidis, A.: Flatfit: accelerated incremental sliding-window aggregation for real-time analytics. In: SSDBM, pp. 1–12 (2017)
81. Shein, A.U., Chrysanthis, P.K., Labrinidis, A.: Slickdeque: High throughput and low latency incremental sliding-window aggregation. In: EDBT, pp. 397–408 (2018)
82. Tangwongsan, K., Hirzel, M., Schneider, S.: Low-latency sliding-window aggregation in worst-case constant time. In: DEBS, pp. 66–77 (2017)
83. Tangwongsan, K., Hirzel, M., Schneider, S.: Optimal and general out-of-order sliding-window aggregation. PVLDB **12**(10), 1167–1180 (2019)
84. Tangwongsan, K., Hirzel, M., Schneider, S., Wu, K.L.: General incremental sliding-window aggregation. PVLDB **8**(7), 702–713 (2015)
85. Tatbul, N., Çetintemel, U., Zdonik, S., Cherniack, M., Stonebraker, M.: Load shedding in a data stream manager. In: Proceedings 2003 VLDB conference, pp. 309–320. Elsevier, Amsterdam (2003)
86. Theodorakis, G., Koliousis, A., Pietzuch, P., Holger, P.: Hammer slide: work-and CPU-efficient streaming window aggregation (2018)
87. Theodorakis, G., Koliousis, A., Pietzuch, P., Pirk, H.: Lightsaber: efficient window aggregation on multi-core processors. In: SIGMOD, pp. 2505–2521 (2020)
88. Theodorakis, G., Pietzuch, P.R., Pirk, H.: Slideside: a fast incremental stream processing algorithm for multiple queries. In: EDBT, pp. 435–438 (2020)
89. Torres, J., Armbrust, M., Das, T., Zhu, S.: Introducing low-latency continuous processing mode in structured streaming in apache spark 2.3. Databricks Blog (2018)
90. Toshniwal, A., Taneja, S., Shukla, A., Ramasamy, K., Patel, J.M., Kulkarni, S., Jackson, J., Gade, K., Fu, M., Donham, J., et al.: Storm@ twitter. In: SIGMOD, pp. 147–156 (2014)
91. Traub, J.: Demand-based data stream gathering, processing, and transmission. Ph.D. thesis, Technische Universität Berlin (2019). https://www.depositonce.tu-berlin.de/handle/11303/10519
92. Traub, J., Grulich, P., Cuéllar, A.R., Breß, S., Katsifodimos, A., Rabl, T., Markl, V.: Scotty: efficient window aggregation for out-of-order stream processing. In: IEEE ICDE, pp. 1300–1303 (2018)
93. Traub, J., Grulich, P., Cuéllar, A.R., Breß, S., Katsifodimos, A., Rabl, T., Markl, V.: Efficient window aggregation with general stream slicing. In: EDBT, pp. 97–108 (2019)
94. Traub, J., Grulich, P.M., Cuéllar, A.R., Breß, S., Katsifodimos, A., Rabl, T., Markl, V.: Scotty: general and efficient open-source window aggregation for stream processing systems. ACM TODS **46**(1), 1–46 (2021)
95. Tucker, P.A., Maier, D., Sheard, T., Fegaras, L.: Exploiting punctuation semantics in continuous data streams. IEEE TKDE **15**(3), 555–568 (2003)
96. Zaharia, M., Xin, R.S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M.J., et al.: Apache spark: a unified engine for big data processing. CACM **59**(11), 56–65 (2016)
97. Zeuch, S., Chaudhary, A., Monte, B., Gavriilidis, H., Giouroukis, D., Grulich, P., Breß, S., Traub, J., Markl, V.: The NebulaStream Platform: data and application management for the internet of things. In: CIDR (2020)
98. Zeuch, S., Monte, B.D., Karimov, J., Lutz, C., Renz, M., Traub, J., Breß, S., Rabl, T., Markl, V.: Analyzing efficient stream processing on modern hardware. PVLDB **12**(5), 516–530 (2019)
99. Zeuch, S., Zacharatou, E.T., Zhang, S., Chatziliadis, X., Chaudhary, A., Del Monte, B., Giouroukis, D., Grulich, P.M., Ziehn, A., Mark, V.: NebulaStream: complex analytics beyond the cloud. In: VLIoT (2020)
100. Zhang, C., Akbarinia, R., Toumani, F.: Efficient incremental computation of aggregations over sliding windows. In: ACM SIGKDD, pp. 2136–2144 (2021)