Efficient Window Aggregation with General Stream Slicing

Jonas Traub[1], Philipp Grulich[2], Alejandro Rodríguez Cuéllar[1], Sebastian Breß[1,2]
Asterios Katsifodimos[3], Tilmann Rabl[1,2], Volker Markl[1,2]

[1]Technische Universität Berlin [2]DFKI GmbH [3]Delft University of Technology

ABSTRACT

Window aggregation is a core operation in data stream processing. Existing aggregation techniques focus on reducing latency, eliminating redundant computations, and minimizing memory usage. However, each technique operates under different assumptions with respect to workload characteristics such as properties of aggregation functions (e.g., invertible, associative), window types (e.g., sliding, sessions), windowing measures (e.g., time- or count-based), and stream (dis)order. Violating the assumptions of a technique can deem it unusable or drastically reduce its performance.

In this paper, we present the first general stream slicing technique for window aggregation. General stream slicing automatically adapts to workload characteristics to improve performance without sacrificing its general applicability. As a prerequisite, we identify workload characteristics which affect the performance and applicability of aggregation techniques. Our experiments show that general stream slicing outperforms alternative concepts by up to one order of magnitude.

1 INTRODUCTION

The need for real-time analysis shifts an increasing number of data analysis tasks from batch to stream processing. To be able to process queries over unbounded data streams, users typically formulate queries that compute aggregates over bounded subsets of a stream, called windows. Examples of such queries on windows are average vehicle speeds per minute, monthly revenue aggregations, or statistics of user behavior for online sessions.

Large computation overlaps caused by sliding windows and multiple concurrent queries lead to redundant computations and inefficiency. Consequently, there is an urgent need for *general* and *efficient* window aggregation in industry [7, 41, 50]. In this paper, we contribute a general solution which not only improves performance but also widens the applicability with respect to window types, time domains, aggregate functions, and out-of-order processing. Our solution is generally applicable to all data flow systems which adopt a tuple-at-a-time processing model (e.g., Apache Storm, Apache Flink, and other Apache Beam-based systems).

To calculate aggregates of overlapping windows, the database community has been working on *aggregation techniques* such as B-Int [3], Pairs [28], Panes [30], RA [42] and Cutty [10]. These techniques compute partial aggregates for overlapping parts of windows and reuse these partial aggregates to compute final aggregates for overlapping windows. We believe that these techniques are not widely adopted in open-source streaming systems for two main reasons: first, the literature on streaming window aggregation is fragmented and, second, every technique has its own assumptions and limitations. As a consequence, it is not clear for researchers and practitioners under which conditions which streaming window aggregation techniques should be used.

General purpose streaming systems require a window operator which is applicable to many types of aggregation workloads. At the same time, the operator should be as efficient as specialized techniques which support selected workloads only.

As our first contribution we identify the workload characteristics which may or may not be supported by existing specialized window aggregation techniques. Those characteristics are: i) *window types* (e.g., sliding, session, tumbling), ii) *windowing measures* (e.g., time or tuple-count), iii) *aggregate functions* (e.g., associative, holistic), and iv) *stream order*. We then conduct an extensive literature survey and classify existing techniques with respect to their underlying concepts and their applicability.

We identify stream slicing as a common denominator on top of which window aggregation can be implemented efficiently. Consequently, our second main contribution is *a general stream slicing technique*. Existing slicing-based techniques do not support complex window types such as session windows [28, 30], do not consider out-of-order processing [10], or limit the type of aggregation functions [10, 28, 30]. With general stream slicing, we provide a single, generally applicable, and highly efficient solution for streaming window aggregation. Our solution inherits the performance of specialized techniques, which use stream slicing, and generalizes stream slicing to support diverse workloads. Because we integrate all workloads into one general solution, we enable computation sharing among all queries with different window types (sliding, sessions, user-defined, etc.) and window measures (e.g., tuple-count or time). General stream slicing is available open source and can be integrated into streaming systems directly as a library[1].

General stream slicing breaks down slicing into three operations on slices, namely `merge`, `split`, and `update`. Specific workload characteristics influence what each operation costs and how often operations are performed. By taking into account the workload characteristics, our slicing technique i) stores the tuples themselves only when it is required which saves memory and ii) minimizes the number of slices that are created, stored, and recomputed. One can extend our techniques with additional aggregations and window types without changing the three core slicing operations. Thus, these core operations may be tuned by system experts while users can still implement custom windows and aggregations.

The contributions of this paper are as follows:

(1) We identify the workload characteristics which impact the applicability and performance limitations of existing aggregation techniques (Section 4).
(2) We contribute *general stream slicing*, a generally applicable and highly efficient solution for streaming window aggregation in dataflow systems (Section 5).
(3) We evaluate the performance implications of different use-case characteristics and show that stream slicing is generally applicable while offering better performance than existing approaches (Section 6).

The remainder of this paper is structured as follows: We first provide background information in Section 2 and present concepts of aggregation techniques in Section 3. We then present our contributions in Section 4, 5, and 6 and discuss related work in Section 7.

[1]Open Source Link: https://github.com/TU-Berlin-DIMA/scotty-window-processor

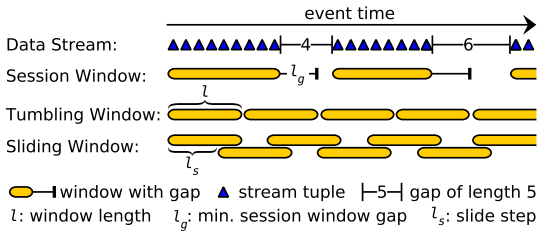


Figure 1: Common Window Types.

2 PRELIMINARIES

Streaming window aggregation involves special terminology with respect to window types, timing, stream order, and data expiration. This section revisits terms and definitions, which are required for the remainder of this paper.

Window Types. A window type refers to the logic based on which systems derive finite windows from a continuous stream. There exist diverse window types ranging from common sliding windows to more complex data-driven windows [17]. We address the diversity of window types with a classification in Section 4.4. For now, we limit the discussion to *tumbling* (or *fixed*), *sliding*, and *session* windows (Figure 1) which we use in subsequent examples. A *tumbling* window splits the time into segments of equal length l. The end of one window marks the beginning of the next window. *Sliding* windows, in addition to the length l, also define a slide step of length l_s. This length determines how often a new window starts. Consecutive windows overlap when $l_s < l$. In this case, tuples may belong to multiple windows. A *session* window typically covers a period of activity followed by a period of inactivity [1]. Thus, a session window times out (ends) if no tuple arrives for some time gap l_g. Typical examples of sessions are taxi trips, browser sessions, and ATM interactions.

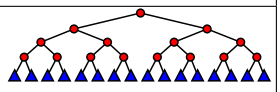
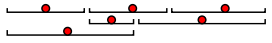
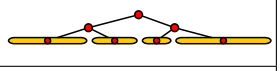
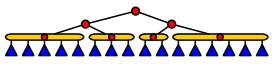
Notion of Time. One can define windows on different measures such as times and tuple-counts. The *event-time* of a tuple is the time when an event was captured and the *processing-time* is the time when an operator processes a tuple [1, 9]. Technically, an event-time is a timestamp stored in the tuple and processing-time refers to a system clock. If not indicated otherwise, we refer to event-time windows in our examples because applications typically define windows on event-time.

Stream Order. Input tuples of a stream are in-order if they arrive chronologically with respect to their event-times, otherwise, they are out-of-order [1, 33]. In practice, streams regularly contain out-of-order tuples because of transmission latencies, network failures, or temporary sensor outages. We differentiate in-order tuples from out-of-order tuples and in-order streams from out-of-order streams. Let a stream S consist of tuples $s_1, s_2, s_3, \ldots$ where the subscripts denote the order in which an operator processes the tuples. Let the event-time of any tuple s_x be $t_e(s_x)$.

- A tuple s_x is *in-order* if $t_e(s_x) \geq t_e(s_y)\ \forall\ y < x$.
- A stream is in-order iff all its tuples are in-order tuples.

Punctuations, Watermarks, and Allowed Lateness. *Punctuations* are annotations embedded in a data stream [47]. Systems use punctuations for different purposes: *low-watermarks* (in short *watermarks*) indicate that no tuple will arrive with a timestamp smaller than the watermark's timestamp [1]. Many systems use watermarks to control how long they wait for out-of-order tuples before they output a window aggregate [2]. *Window punctuations* mark window starts and endings in the stream [14, 20]. The *allowed lateness*, specifies how long systems store window aggregates. If an out-of-order tuple arrives after the watermark, but in the allowed lateness, we output updated aggregates.

Partial Aggregates and Aggregate Sharing. The key idea of partial aggregation is to compute aggregates for subsets of the stream as intermediate results. These intermediate results are *shared* among overlapping windows to prevent repeated computation [3, 28, 51]. In addition, one can compute partial aggregates incrementally when tuples arrive [42]. This reduces the memory footprint if a technique stores few partial aggregates instead of all stream tuples in the allowed lateness. It also reduces the latency because aggregates are pre-computed when windows end.

	Memory Usage	Example
1. Tuple Buffer	$\lvert▲\rvert \cdot \texttt{size}(▲)$	
2. Aggregate Tree	$\lvert▲\rvert \cdot \texttt{size}(▲)$ $+(\lvert▲\rvert - 1) \cdot \texttt{size}(●)$	
3. Agg. Buckets	$\lvert\texttt{win}\rvert \cdot \texttt{size}(●)$ $+\lvert\texttt{win}\rvert \cdot \texttt{size}(\llcorner\!\lrcorner)$	
4. Tuple Buckets	$\lvert\texttt{win}\rvert \cdot [\texttt{avg}(▲\ \texttt{per win.})$ $\cdot \texttt{size}(▲) + \texttt{size}(\llcorner\!\lrcorner)]$	
5. Lazy Slicing	$\lvert⬭\rvert \cdot \texttt{size}(⬭)$	
6. Eager Slicing	$\lvert⬭\rvert \cdot \texttt{size}(⬭)$ $+(\lvert⬭\rvert - 1) \cdot \texttt{size}(●)$	
7. Lazy Slicing on tuples	$\lvert▲\rvert \cdot \texttt{size}(▲)$ $+\lvert⬭\rvert \cdot \texttt{size}(⬭)$	
8. Eager Slicing on tuples	$\lvert▲\rvert \cdot \texttt{size}(▲)$ $+\lvert⬭\rvert \cdot \texttt{size}(⬭)$ $+(\lvert⬭\rvert - 1) \cdot \texttt{size}(●)$	

Legend: ▲ Tuple ● Aggregate ⬭ Slice incl. Aggregate ⌴ Bucket

Table 1: Memory Usage of Aggregation Techniques.

3 WINDOW AGGREGATION CONCEPTS

In this section, we survey concepts for streaming window aggregation and give an intuition for each solution's memory usage, throughput, and latency. We provide a detailed comparison of all concepts in our experiments. Techniques which support out-of-order streams store values for an *allowed lateness* (see above). In the following discussion, we refer to allowed lateness only. Techniques which do not process out-of-order tuples, store values for the duration of the longest window.

Table 1 provides an overview of all techniques we discuss in the following subsections. We write $\lvert▲\rvert$ for the number of values (i.e., tuples), $\lvert⬭\rvert$ for the number of slices, and $\lvert\texttt{win}\rvert$ for the number of windows in the allowed lateness.

3.1 Tuple Buffer

A tuple buffer (Table 1, Row 1) is a straightforward solution which does not share partial aggregates.

The *throughput* of a tuple buffer is fair as long as there are few or no concurrent windows (i.e., no window overlaps), and there are few or no out-of-order tuples. Window overlaps decrease the throughput because of repeated aggregate computations. Out-of-order tuples decrease the throughput because of memory copy operations which are required for inserting values in the middle of a sorted ring buffer.

The *latency* of a tuple buffer is high because aggregates are computed lazily when windows end. Thus, all aggregate computations contribute to the latency at the window end.

A tuple buffer stores all tuples for the allowed lateness, which is $\lvert▲\rvert \cdot \texttt{size}(▲)$. Thus, the more tuples we process per time, the higher the memory consumption and the higher the memory copy overhead for out-of-order tuples.

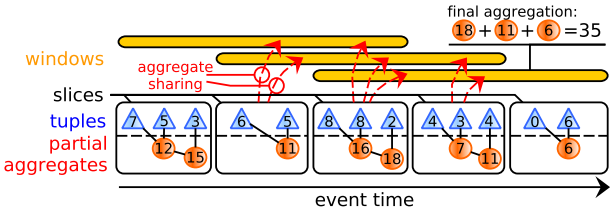


Figure 2: Example Aggregation with Stream Slicing.

3.2 Aggregate Trees

Aggregate trees such as FlatFAT [42] and B-INT [3] store partial aggregates in a tree structure and share them among overlapping windows (Table 1, Row 2). FlatFAT stores a binary tree of partial aggregates on top of stream tuples (leaves) which roughly doubles the memory consumption.

In-order tuples require $\log(|▲|)$ updates of partial aggregates in the tree. Thus, the *throughput* is decreased logarithmically when the number of tuples in the allowed lateness increases. Out-of-order tuples decrease the throughput drastically: they require the same memory copy operation as in tuple buffers. In addition, they cause a rebalancing of the aggregate tree and the respective aggregate updates.

The *latency* of aggregate trees is much lower than for tuple buffers because they can compute final aggregates for windows from pre-computed partial aggregates. Thus, only a few final aggregation steps remain when windows end [39].

3.3 Buckets

Li et al. introduce *Window-ID* (WID) [31–33], a bucket-per-window approach which is adopted by many systems with support for out-of-order processing [1, 2, 9]. Each window is represented by an independent bucket. A system assigns tuples to buckets (i.e., windows) based on event-times, independently from the order in which tuples arrive [33]. Buckets do not utilize aggregate sharing. Instead, they compute aggregates for each bucket independently.

Systems can compute aggregates for buckets incrementally [42]. This leads to very low *latencies* because the final window aggregate is pre-computed when windows end.

We consider two versions of buckets. *Tuple buckets* keep individual tuples in buckets (Table 1, Row 4). This leads to data replication for overlapping buckets. *Aggregate buckets* store partial aggregates in buckets plus some overhead (e.g., start and end times), but no tuples (Table 1, Row 3).

We prefer to store aggregates only in order to save memory. However, some use-cases (e.g., holistic aggregates over count-based windows) require us to keep individual tuples.

Buckets process in-order tuples as fast as out-of-order tuples for most use-cases: they assign the tuple to buckets and incrementally compute the aggregate of these buckets. The throughput bottleneck for buckets are overlapping windows. For example, one sliding window with $l = 20$s and $l_s = 2$s results in 10 overlapping windows (i.e., buckets) at any time. This causes 10 aggregation operations for each input tuple.

3.4 Stream Slicing

Slicing techniques divide (i.e., *slice*) a data stream into non-overlapping chunks of data (i.e., *slices*) [28, 30]. The system computes a partial aggregate for each slice. When windows end, the system computes window aggregates from slices.

We show stream slicing with an example in Figure 2. Slicing techniques compute partial aggregates incrementally when tuples arrive (bottom of Figure 2). We show multiple intermediate aggregates per slice to illustrate the workflow.

Partial aggregates (i.e., slices) are shared among overlapping windows which avoids redundant computations. In Figure 2, dashed arrows mark multiple uses of slices. In contrast to aggregate trees and buckets, slicing techniques require just one aggregation operation per tuple because each tuple belongs to exactly one slice. This results in a high *throughput*.

Similar to aggregate trees, the *latency* of stream slicing techniques is low because only a few final aggregation steps are required when a window ends. We consider a lazy and an eager version of stream slicing. The lazy version of stream slicing stores slices including partial aggregates (Table 1, Row 5). The eager version stores a tree of partial aggregates on top of slices to further reduce latencies (Table 1, Row 6). Both variants compute aggregates of slices incrementally when tuples arrive. The term *lazy* refers to the lazy computation of aggregates for combinations of slices.

There are usually many tuples per slice ($|⬭| \ll |▲|$) which leads to huge memory savings compared to aggregate trees and tuple buffers. Some use-cases such as holistic aggregates over count-based windows require us to keep individual tuples in addition to aggregates (Table 1, Row 7 and 8). In these cases, stream slicing requires more memory than tuple buffers, but saves memory compared to buckets and aggregate trees.

In this paper, we focus on stream slicing because it offers a good combination of high throughputs, low latencies, and memory savings. Moreover, our experiments show that slicing techniques scale to many concurrent windows, high ingestion rates, and high fractions of out-of-order tuples.

4 WORKLOAD CHARACTERIZATION

In this section, we identify workload characteristics which either limit the applicability of aggregation techniques or impact their performance. These characteristics are the basis for subsequent sections in which we generalize stream slicing.

4.1 Characteristic 1: Stream Order

Out-of-order streams increase the complexity of window aggregation because out-of-order tuples can require changes in the past. For example, tuple buffers and aggregate trees process in-order tuples efficiently using a ring buffer (FIFO principle) [42]. Out-of-order tuples break the FIFO principle and require memory copy operations in buffers.

We differentiate whether or not out-of-order processing is required for a use-case. For techniques which support out-of-order processing, we study how the fraction of out-of-order tuples and the delay of such tuples affect the performance.

4.2 Characteristic 2: Aggregation Function

We classify aggregation functions with respect to their algebraic properties. Our notation splits the aggregation in incremental steps and is consistent with related works [10, 42]. We write input values as lower case letters, the operation which adds a value to an aggregate as $\oplus$, and the operation which removes a value from an aggregate as $\ominus$. We first adopt three algebraic properties used by Tangwongsan et al. [42]. These properties focus on the incremental computation of aggregates:

(1) Associativity: $(x \oplus y) \oplus z = x \oplus (y \oplus z) \;\; \forall\, x, y, z$
(2) Invertibility: $(x \oplus y) \ominus y = x \;\; \forall\, x, y$
(3) Commutativity: $x \oplus y = y \oplus x \;\; \forall\, x, y$

Stream slicing requires *associative* aggregate functions because it computes partial aggregates per slice which are shared among windows. This requirement is inherent for all techniques which share partial aggregates [3, 10, 28, 30, 42]. Our general slicing approach does not require invertibility or commutativity, but exploits these properties if possible to increase performance.

We further adopt the classification of aggregations in *distributive*, *algebraic*, and *holistic* [16]. Aggregations such as `sum`, `min`, and `max` are *distributive*. Their partial aggregates equal the final aggregates of partials and have a constant size. An aggregation is *algebraic* if its partial aggregates can be summarized in an intermediate result of fixed size. The final aggregate is computed from this intermediate result. The remainder of aggregations, which have an unbounded size of partial aggregates, is *holistic*.

4.3 Characteristic 3: Windowing Measure

Windows can be specified using different measures (also called *time domains* [8] or `WATTR` [31]). For example, a tumbling window can have a length of 5 minutes (time-measure), or a length of 10 tuples (count-measure). To simplify the presentation, we refer to *timestamps* in the rest of the paper. However, bear in mind that a timestamp can actually be a point in time, a tuple count, or any other monotonically increasing measure [10]:

- **Time-Based Measures:** Common time-based measures are *event-time* and *processing-time* as introduced in Section 2.
- **Arbitrary Advancing Measures** are a generalization of event-times. Typically, it is irrelevant for a stream processor if *"timestamps"* actually represent a time or another advancing measure. Examples of other advancing measures are transaction counters in a database, kilometers driven by a car, and invoice numbers.
- **Count-Based Measures** (also called *tuple-based* [31] or *tuple-driven* [8]) refer to a tuple counter. For example, a window can start at the 100th and end at the 200th tuple of a stream. Count-based measures cause challenges when combined with out-of-order processing: If tuples are ordered with respect to their event-times and a tuple arrives out-of-order, it changes the count of all other tuples which have a greater event-time. This changes the aggregates of all count-based windows which start or end after the out-of-order tuple.

If we process multiple queries which use different window-measures, timestamps are represented as vectors which contain multiple measures as dimensions. This representations allows for slicing the stream with respect to multiple dimensions (i.e., measures) while slices are still shared among all queries [10].

4.4 Characteristic 4: Window Type

We classify window types with respect to the *context* (or *state*) which is required to know where windows start and end. We adopt the classification in context free (CF), forward-context aware (FCA), and forward-context free (FCF) introduced by Li et al. [31]. Here we present those classes along with the most common window types belonging to those classes.

- **Context Free (CF).** A window type is context free if one can tell all start and end timestamps of windows without processing any tuples. Common *sliding* and *tumbling* windows are context free because we can compute all start and end timestamps a priori based on the parameters l and l_s.
- **Forward Context Free (FCF).** Windows are forward context free, if one can tell all start and end timestamps of windows up to any timestamp t, once all tuples up to this timestamp t have been processed. An example are *punctuation-based* windows where punctuations mark start and end timestamps [14]. Once we processed all tuples up to t (including out-of-order tuples), we also processed all punctuations before t and, thus, we know all start and end positions up to t.
- **Forward Context Aware (FCA).** The remaining window types are forward context aware. Such window types require us to process tuples after a timestamp t in order to know all window start and end timestamps before t. An example of such windows

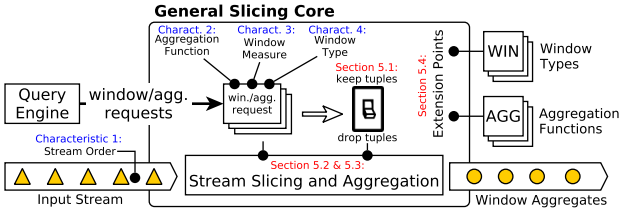


Figure 3: Architecture of General Stream Slicing

are *Multi-Measure Windows* which define their start and end timestamps on different measures. For example, *output the last 10 tuples (count-measure) every 5 seconds (time-measure)* is forward context aware: we need to process tuples up to a window end in order to compute the window begin.

5 GENERAL STREAM SLICING

We now present our general stream slicing technique which supports high-performance aggregation for multiple queries with diverse workload characteristics. General stream slicing replaces alternative operators for window aggregation without changing their input or output semantics. Our technique minimizes the number of partial aggregates (saving memory), reduces the final aggregation steps when windows end (reducing latency), and avoids redundant computation for overlapping windows (increasing throughput). The main idea behind our technique is to exploit workload characteristics (Section 4) and to automatically adapt aggregation strategies. Such adaptivity is a highly desired feature of an aggregation framework: current non-adaptive techniques fail to support multiple window types, process in-order streams only, cannot share aggregates among windows defined on different measures, lack support for holistic aggregations, or incur dramatically reduced performance in exchange for being generally applicable.

Approach Overview. Figure 3 depicts an overview of our general slicing and aggregation technique. Users specify their queries in a high-level language such as a flavor of stream SQL or a functional API. The query translator observes the characteristics of a query (i.e., window type, aggregate function, and window measure) as well as the characteristics of input streams (in-order vs. out-of-order streams) and forwards them to our aggregator. Once those characteristics are given to our aggregator, our general slicing technique adapts automatically to the given workload characteristics.

More specifically, general slicing detects if individual tuples need to be kept in memory (to ensure generality) or if they can be dropped after computing partial aggregates (to improve performance). We further discuss this in Section 5.1. Queries can be added or removed from the aggregator and due to that, the workload characteristics can change. To this end, our aggregator adapts when one adds or removes queries. Weather we need to keep tuples in memory or not solely depends on workload characteristics. Thus, there is no need to adapt on changes in the input data streams such as a changing ratio of out-of-order tuples. When processing input tuples, the stream slicing component automatically decides when it needs to apply our three fundamental slicing operations: merge, split, and update (discussed in Section 5.2). General slicing has extension points that can be used to implement user-defined window types and aggregations (discussed in Section 5.4).

5.1 Storing Tuples vs. Partial Aggregates

General aggregation techniques [3, 42] achieve generality by storing all input tuples and by computing high-level partial aggregates. Specialized techniques, on the other hand, only store (partial) aggregates. A general slicing technique needs to decide when to store what, according to workload characteristics of each of the queries

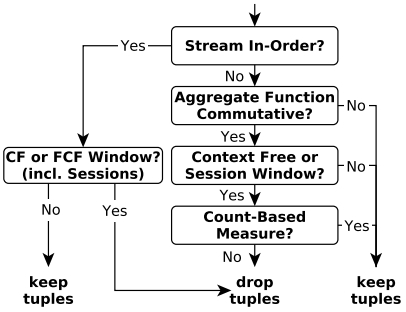


Figure 4: Decision Tree - Which workload characteristics require storing individual tuples in memory?

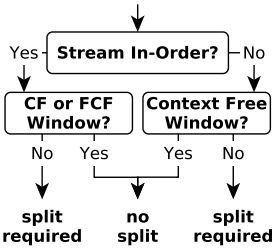


Figure 5: Decision Tree. Are splits required?

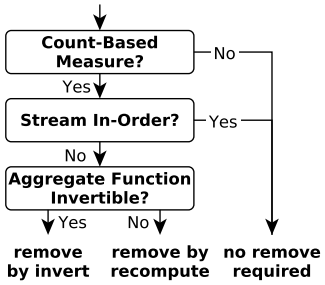


Figure 6: Decision Tree. How to remove tuples?

that it serves. In this section, we discuss how we match the performance of specialized techniques, by choosing on-the-fly whether to keep tuples for a workload or to store partial aggregates only.

For example, consider an aggregation function which is non-commutative ($\exists x, y : x \oplus y \neq y \oplus x$) defined over an unordered stream. When an out-of-order tuple arrives, we need to recompute aggregates from the source tuples in order to retain the correct order of the aggregation. Thus, one would have to store the actual tuples for possible later use. Storing all tuples for the whole duration of the allowed lateness requires more memory but allows for computing arbitrary windows from stored tuples. The decision tree in Figure 4 summarizes when storing source tuples is required depending on different workload characteristics.

In-order Streams. For in-order streams, we drop tuples for all context free and forward context free windows but must keep tuples if we process forward context aware windows. For such windows, forward context leads to additional window start or end timestamps. Thus, we must be able to compute partial aggregates for arbitrary timestamp ranges from the original stored tuples.

Out-of-order Streams. For out-of-order streams, we need to keep tuples if at least one of the following conditions is true:

(1) The aggregation function is non-commutative.
An out-of-order tuple changes the order of the incremental aggregation, which forces us to recompute the aggregate using source tuples. For in-order processing, the commutativity of aggregation functions is irrelevant because tuples are always aggregated in-order. Thus, there is no need to store source tuples in addition to partial aggregates.

(2) The window is neither context free nor a session window.
In combination with out-of-order tuples, all context aware windows require tuples to be stored. This is because out-of-order tuples change backward context which can lead to additional window start or end timestamps. Such additional start and end timestamps require to split slices and to recompute the respective partial aggregates from the original tuples. Session windows are an exception, because they are context aware, but never require recomputing aggregates [46].

(3) The query uses a count-based window measure.
An out-of-order tuple changes the count of all succeeding tuples. Thus, the last tuple of each count-based window shifts to a succeeding window.

5.2 Slice Management

Stream slicing is the fundamental concept that allows us to build partial aggregates and share them among concurrently running queries and overlapping windows. In this section, we introduce three fundamental operations which we can perform on slices.

Slice Metadata. A slice stores its start timestamp ($t_{\texttt{start}}$), its end timestamp ($t_{\texttt{end}}$), and the timestamp of the first ($t_{\texttt{first}}$) and last tuple it contains ($t_{\texttt{last}}$). Note that the timestamps of the first and last tuples do not need to conincide with the start and end timestamps of a slice. For instance, consider a slice A that starts at $t_{\texttt{start}}(A) = 1$ and ends at $t_{\texttt{end}}(A) = 10$ but the first (earliest) tuple contained is timestamped as $t_{\texttt{first}}(A) = 2$ and its last/latest one as $t_{\texttt{last}}(A) = 9$. We remind the reader that the *timestamp* can refer not only to actual time, but to any measure presented in Section 4.3.

We identify three fundamental operations which we perform on stream slices. These operations are *i*) *merging* of two slices into one, *ii*) *splitting* one slice into two, and *iii*) *updating* the state of a slice (i.e., aggregate and metadata updates). In the following paragraphs, we discuss `merge`, `split`, and `update` as well as the impact of our workload characteristics on each operation. We use upper case letters to name slices and corresponding lower case letters for slice aggregates.

Merge. Merging two slices A and B happens in three steps:

1. Update the end of A such that $t_{\texttt{end}}(A) \leftarrow t_{\texttt{end}}(B)$.
2. Update the aggregate of A such that $a \leftarrow a \oplus b$.
3. Delete slice B, which is now merged into A.

Steps one and three have a constant computational cost. The complexity of the second step ($a \leftarrow a \oplus b$) depends on the type of aggregate function. For instance, the cost is constant for algebraic and distributive functions such as `sum`, `min`, and `avg` because they require just a few basic arithmetic operations. Holistic functions such as quantiles can be more complex to compute. Except from the type of aggregation function, no other workload characteristics impact the complexity of the merge operation. However, stream order and window types influence when and *how often* we merge slices. We discuss this influence in Section 5.3.

Split. Splitting a slice A at timestamp t requires three steps:

1. Add slice B: $t_{\texttt{start}}(B) \leftarrow t + 1$ and $t_{\texttt{end}}(B) \leftarrow t_{\texttt{end}}(A)$.
2. Update the end of A such that $t_{\texttt{end}}(A) \leftarrow t$.
3. Recompute the aggregates of A and B.

Note that splitting slices is an expensive operation because it requires recomputing slice aggregates from scratch. Moreover, if splitting is required, we need to keep individual tuples in memory to enable the recomputation.

We show in Figure 5 when split operations are required. For in-order streams, only forward context aware (FCA) windows require split operations. For such windows, we split slices according to a window's start and end timestamp as soon as we process the required forward context. In out-of-order data streams, all context aware windows require split operations because out-of-order tuples possibly contain relevant backward context. We never split slices for context free windows such as tumbling and sliding ones.

Update. Updating a slice can involve adding in-order tuples, adding out-of-order tuples, removing tuples, or changing metadata ($t_{\texttt{start}}$, $t_{\texttt{end}}$, $t_{\texttt{first}}$, and $t_{\texttt{last}}$).

Metadata changes are simple assignments of new values to the existing variables. Adding a tuple to a slice requires one incremental aggregation step ($\oplus$), with the exception of processing out-of-order tuples with a non-commutative aggregation function.

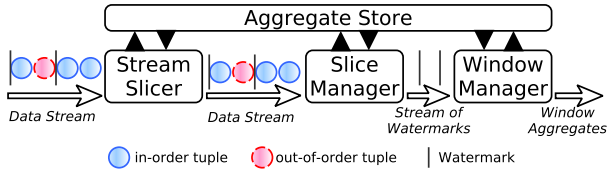


Figure 7: The Stream Slicing and Aggregation Process

For this, we recompute the aggregate of the slice from scratch to retain the order of aggregation steps.

For some workloads we need to remove tuples from slices. We show in Figure 6 when and how we remove tuples from slices. Generally, a remove operation is required only if a window is defined on a count-based measure and if we process out-of-order tuples. An out-of-order tuple changes the count of all succeeding tuples. This requires us to shift the last tuple of each slice one slice further starting at the slice of the out-of-order tuple. If the aggregation function is invertible, we exploit this property by performing an incremental update. Otherwise, we have to recompute the slice aggregate from scratch. If the out-of-order tuple has a small delay, such that it still belongs to the latest slice, we can simply add the tuple without performing a remove operation.

5.3 Processing Input Tuples

The stream slicing and aggregation logic (bottom of Figure 3) consists of four components which we show in Figure 7. The Aggregate Store is our shared data structure which is accessed by the Stream Slicer to create new slices, by the Slice Manager to update slices, and by the Window Manager to compute window aggregates.

The input stream can contain in-order tuples, out-of-order tuples, and watermarks. Note that in-order tuples can either arrive from an in-order stream (i.e., one that is guaranteed to never contain an out-of-order tuple) or from an out-of-order stream (i.e., one that does not guarantee in-order arrival). If the the stream is in-order (i.e., all tuples are in-order tuples), there is no need to ingest watermarks. Instead, we output windows directly since there is no need to wait for potentially delayed tuples.

Step 1 - The Stream Slicer. The Stream Slicer initializes new slices on-the-fly when in-order tuples arrive [28]. In an in-order stream, it is sufficient to start slices when windows start [10]. In an out-of-order stream, we also need to start slices when windows end to allow for updating the last slice of windows later on with out-of-order tuples. We always cache the timestamp of the next upcoming window edge and compare in-order tuples with this timestamp. As soon as the timestamp of a tuple exceeds the cached timestamp, we start a new slice and cache the timestamp of the next edge. This is highly efficient because the majority of tuples do not end a slice and require just one comparison of timestamps.

The Stream Slicer does not process out-of-order tuples and watermarks but forwards them directly to the Slice Manager. This is possible because the slices for out-of-order tuples have already been initialized by previous in-order tuples.

Step 2 - The Slice Manager. The Slice Manager is responsible for triggering all `split`, `merge`, and `update` operations on slices.

First, the Slice Manager checks whether a `merge` or `split` operation is required. We always merge and split slices such that all slice edges match window edges and vice versa. This guarantees that we maintain the minimum possible number of slices [10, 46].

In an out-of-order stream, context aware windows can cause `merges` or `splits`. In an in-order stream, only forward context aware windows can cause these operations. Context free windows never require `merge` or `split` operations, as the window edges are known in advance and slices never need to change.

In-order tuples can be part of the forward context which indicates window start or end timestamps earlier in the stream. When processing forward context aware windows, we check if the new tuple changes the context such that it introduces or removes window start or end timestamps. In such case, we perform the required `merge` and `split` operation to match the new slice and window edges. Out-of-order tuples can change forward and backward context such that a `merge` operation or `split` operation are required.

If the new context causes new window edges and, thus, `merge` or `split` operations, we notify the Window Manager which outputs window aggregates up to the current watermark.

Finally, the Slice Manager adds the new tuple to its slice and updates the slice aggregate accordingly. In-order tuples always belong to the current slice and are added with an incremental aggregate update [42]. For out-of-order tuples, we look up the slice which covers the timestamp of the out-of-order tuple and add the tuple to this slice. For commutative aggregation functions, we add the new tuple with an incremental aggregate update. For non-commutative aggregation functions, we need to recompute the aggregate from individual tuples to retain the correct order.

Step 3 - The Window Manager. The Window Manager computes the final aggregates for windows from slice aggregates.

When processing an in-order stream, the Window Manager checks if the tuple it processes is the last tuple of a window. Therefore, each tuple can be seen as a watermark which has the timestamp of the tuple. If a window ended, the window manager computes and outputs the window aggregate (final aggregation step).

For out-of-order streams, we wait for the watermark (see Section 2) before we output results of windows which ended before a watermark.

The Slice Manager notifies the Windows Manager when it performs `split`, `merge`, or `update` operation on slices. Upon such notification, the Window Manager performs two operations:

(1) If an out-of-order tuple arrives within the allowed lateness but after the watermark, the tuple possibly changes aggregates of windows which were output before. Thus, the Window Manager outputs updates for these window aggregates.
(2) If a tuple changes the context of context aware windows such that new windows end before the current watermark, the window manager computes and outputs the respective aggregates.

Parallelization. We parallelize stream processing with key partitioning which is the common approach used in stream processing systems [21] such as Flink [9], Spark [4], and Storm [44]. Key partitioning enables intra-node as well as inter-node parallelism and, thus, results in good scalability. Since our generic window aggregation is a drop in replacement for the window aggregation operator, the input and output semantics of the operator remains unchanged. Thus, neither the query interface nor optimizations unrelated to window aggregations are affected.

5.4 User-Defined Windows and Aggregations

Our architecture decouples the general logic of stream slicing from the concrete implementation of window types and aggregation functions. This makes it easy to add window types and aggregation functions as no changes are required in the slicing logic. In this section, we describe how we implement aggregation functions and window types.

5.4.1 Implementing Aggregation Functions. We adopt the same approach of incremental aggregation introduced by Tangwongsan et al. [42]. Each aggregation type consists of three functions: `lift`, `combine`, and `lower`. In addition, aggregations may implement an

`invert` function. We now discuss the concept behind these functions, and refer the reader to the original paper for an overview of different aggregations and their implementation.

Lift. The `lift` function transforms a tuple to a partial aggregate. For example, consider an average computation. If a tuple $\langle t, v \rangle$ contains its timestamp t and a value v, the `lift` function will transform it to $\langle \texttt{sum} \leftarrow v, \texttt{count} \leftarrow 1 \rangle$, which is the partial aggregate of that one tuple.
Combine. The `combine` function ($\oplus$) computes the combined aggregate from partial aggregates. Each incremental aggregation step results in one call of the `combine` function.
Lower. The `lower` function transforms a partial aggregate to a final aggregate. In our example, the lower function computes the average from sum and count: $\langle \texttt{sum}, \texttt{count} \rangle \mapsto \texttt{sum/count}$.
Invert. The optional `invert` function removes one partial aggregate from another with an incremental operation.

In this work, we consider holistic aggregation functions which have an unbounded size of partial aggregates. A widely used holistic function is the computation of quantiles. For instance, windowed quantiles are the basis for billing models of content delivery networks and transit-ISPs [13, 23]. For quantile computations, we sort tuples in slices to speed up succeeding merge operations and apply run length encoding to save memory [37].

5.4.2 Implementing Different Window Types. We use a common interface for the in-order slicing logic of all windows. We extend this interface with additional methods for context-aware windows. One can add additional window types by implementing the respective interface.

Context Free Windows. The slicing logic for context free windows depends on in-order tuples only. When a tuple is processed, the slicing core initializes all slices up to the timestamp of that tuple. Our interface for context free windows has two methods: The first method has the signature `long getNextEdge(long timestamp)`. It receives a timestamp as parameter and returns the next window edge (begin or end timestamp) after this timestamp. We use this method to retrieve the next window edge for on-the-fly stream slicing (Step 1 in Section 5.3). For example, a tumbling window with length l would return $timestamp + l - (timestamp \bmod l)$.

The second method triggers the final window aggregation according to a watermark and has the following signature:
`void triggerWin(Callback c, long prevWM, long currWM)`. The Window Manager calls this method when it processes a watermark. `c` is a callback object, `prevWM` is the timestamp of the previous watermark and `currWM` is the timestamp of the current watermark. The method reports all windows which ended between `prevWM` and `currWM` by calling `c.triggerWin(long startTime, long endTime)`. This callback to the Window Manager triggers the computation and output of the final window aggregate.
Context Aware Windows. Context aware windows use the same interface as context free windows to trigger the initialization of slices when processing in-order tuples. In addition, context aware windows require to keep a state (i.e., context) in order to derive window start and end timestamps when processing out-of-order tuples. We initialize context aware windows with a pointer to the Aggregate Store. This prevents redundancies among the state of the shared aggregator and the window state. When the Slice Manager processes a tuple, it notifies context aware windows by calling `window.notifyContext(callbackObj, tuple)`. This method can then add and remove window start and end timestamps through the callback object and the Slice Manager splits and merges slices as required to match window start and end timestamps. We detect whether or not a window is context aware based on the interface which is implemented by the window specification. We provide examples for different context free and context aware window implementations in our open source repository.

6 EVALUATION

In this section, we evaluate the performance of general stream slicing and compare stream slicing with alternative techniques introduced in Section 3.

6.1 Experimental Setup

Setup. We implement all techniques on Apache Flink v1.3. We run our experiments on a VM with 6 GB main memory and 8 processing cores with 2.6 GHz.
Metrics. In our experiments, we report throughput, latency, and memory consumption. We measure throughput as in the Yahoo Streaming Benchmark implementation for Apache Flink [12, 48]. We determine latencies with the JMH benchmarking suite [35]. JMH provides precise latency measurements on JVM-based systems. We use the *ObjectSizeCalculator* of Nashorn to determine memory footprints [36].
Baselines. We compare an eager and a lazy version of general stream slicing with non-slicing techniques from Section 3: As representative for aggregate trees, we implement FlatFAT [42]. For the buckets technique, we use the implementation of Apache Flink [9]. For tuple buffers, we use an implementation based on a ring buffer array. We also include Pairs [28] and Cutty [10] as specialized slicing techniques where possible.
Data. We replay real-world sensor data from a football match [34] and from manufacturing machines [25]. The original data sets track the position of the football with 2000 and the machine states with 100 updates per second. We generate additional tuples based on the original data to simulate higher ingestion rates. We add 5 gaps per minute to separate sessions. This is representative for the ball possession moving from one player to another. If not indicated differently, we show results for the football data. The results for other data sets are almost identical because the performance depends on workload characteristics rather than data characteristics.
Queries. We base our queries (i.e., window length, slide steps, etc.) on the workload of a live-visualization dashboard which is built for the football data we use [45]. If not indicated differently, we use the `sum` aggregation in Sections 6.2 and Section 6.3. In Section 6.4, we use the M4 aggregation technique [26] to compress the data stream for visualization. M4 computes four algebraic aggregates per window (i.e., minimum, maximum, first and last value of each window). We show in Section 6.3.2 how the performance differs among diverse aggregation functions. Because we do not change the input and output semantics of the window and aggregation operation, there is no impact on upstream or downstream operations. We ensure that windowing and aggregation are the bottleneck and, thus, we measure the performance of aggregation techniques.

We do not alternate between tumbling and sliding windows because they lead to identical performance: For example, 20 concurrent tumbling window queries cause 20 concurrent windows (1 window for each query at any time). This is equivalent to a single sliding window with a window length of 20 seconds and and a slide step of one second (again 20 concurrent windows). In the following, we refer to *concurrent windows* instead of *concurrent tumbling window queries. Sliding window queries* yield identical results if they imply the same number of *concurrent windows*.
Structure. We split our evaluation in three parts. First, we compare stream slicing and alternative approaches with respect to their throughput, latency, and memory footprint (Section 6.2). Second, we study the impact of each workload characteristic introduced in Section 4 (Section 6.3). Third, we integrate general slicing in

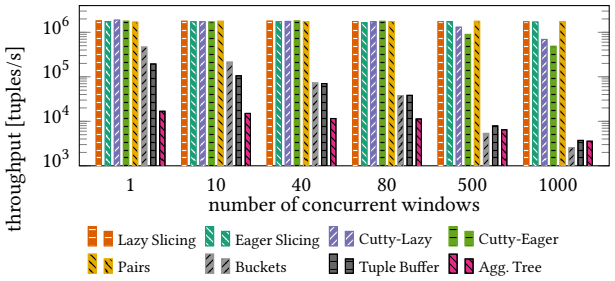
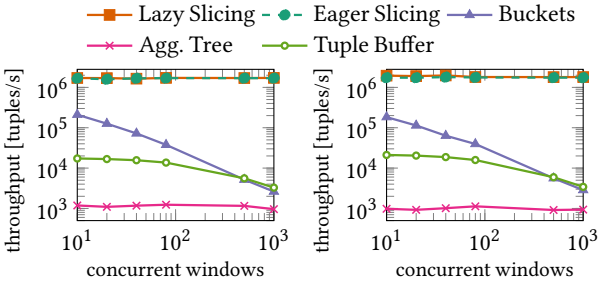


Figure 8: In-order Processing with Context Free Windows.



(a) Football data set [34]. **(b) Machine data set [25].**

Figure 9: Increasing the number of concurrent windows including 20% out-of-order tuples and session windows.

Apache Flink and show the performance gain for a concrete application (Section 6.4). Sections 6.2 and 6.3 focus on the performance per operator instance. Section 6.4 studies the parallelization.

6.2 Stream Slicing Compared to Alternatives

We now compare stream slicing with alternative techniques discussed in Section 3. We first study the throughput for in-order processing on context-free windows in Section 6.2.1. Our goal is to understand the performance of stream slicing compared to alternative techniques, including specialized slicing techniques. In Section 6.2.2, we evaluate how the throughput changes in the presence of out-of-order tuples and context-aware windows. In Section 6.2.3, we evaluate the memory footprint and in Section 6.2.4 the latency of different techniques.

6.2.1 *Throughput.*

Workload. We execute multiple concurrent tumbling window queries with equally distributed lengths from 1 to 20 seconds. These window lengths are representative of window aggregations which facilitate plotting line charts at different zoom levels (Application of Section 6.3). We chose Pairs [28] and Cutty [10] as example slicing techniques because Pairs is one of the first and most cited techniques and Cutty offers a high generality with respect to window types.

Results. We show our results in Figure 8. All three slicing techniques process millions of tuples per second and scale to large numbers of concurrent windows.

Buckets achieves orders of magnitude less throughput than Slicing techniques and does not scale to large numbers of concurrent windows. The reason is that we must assign each tuple to all concurrent buckets (i.e., windows). Thus, tuples belong to up to 1000 buckets causing 1000 redundant aggregation steps per tuple. In contrast, slicing techniques always assign tuples to exactly one slice. Similar to buckets, the tuple buffer causes redundant aggregation steps for each window as we compute each window independently. Aggregate Trees show a throughput which is orders of magnitude smaller than the one of slicing techniques. This is because each tuple requires several updates in the tree.

Summary. We observe that slicing techniques outperform alternative concepts with respect to throughput and scale to large numbers of concurrent windows.

6.2.2 *Throughput under Constraints.*

We now analyze the throughput under constraints, i.e., including out-of-order tuples and context-aware windows.

Workload. The workload remains the same as before but we add a time-based session window (l_g = 1sec.) as representative for a context-aware window. We add 20% out-of-order tuples with random delays between 0 and 2 seconds.

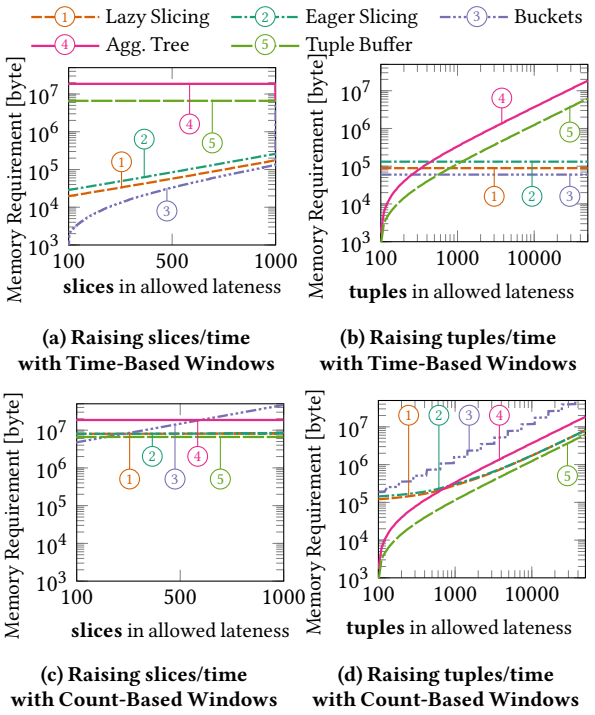
Results. We show the results in Figure 9. Slicing techniques achieve an order of magnitude higher throughput than alternative techniques which do not use stream slicing. Moreover, slicing scales to large numbers of concurrent windows with almost constant throughput. This is because the per-tuple complexity remains constant: we assign each tuple to exactly one slice. Lazy Slicing has the highest throughput (1.7 Million tuples/s) because it uses stream slicing and does not compute an aggregate tree. Eager Slicing achieves slightly lower throughput than Lazy Slicing. This is due to out-of-order tuples which cause updates in the aggregate tree. Buckets show the same performance decrease as in the previous experiment. The performance decrease for the Tuple Buffer is intensified due to out-of-order inserts in the ring buffer array. Aggregate Trees process less than 1500 tuples/s with 20% out-of-order tuples. This is because out-of-order tuples require expensive leaf inserts in the aggregate tree (rebalance and update of inner nodes). Eager slicing seldom faces this issue because it stores slices instead of tuples in the aggregate tree. The majority of out-of-order tuples falls in an existing slice which prevents rebalancing. We exemplary show our results on two different datasets for this experiment. Because the performance depends on workload characteristics rather than data characteristics, the results are almost identical. We omit similar results for different data sets in the following experiments and focus on the impact of workload characteristics.

Summary. For workloads including out-of-order tuples and context-aware windows, we observe that general stream slicing outperforms alternative concepts with respect to throughput and scales to large numbers of concurrent windows.

6.2.3 *Memory Consumption.*

We now study the memory consumption of different techniques with four plots: In Figures 10a and 10c, we vary the number of slices in the allowed lateness and fix the number of tuples in the allowed lateness to 50 thousand. In Figures 10b and 10d, we vary the number of tuples and fix the number of slices to 500. We experimentally compare time-based and count-based windows. Our measurements include all memory required for storing partial aggregates and metadata such as the start and end times of slices.

Results for Time-Based Windows. Figures 10a and 10b show the memory consumption for time-based windows, which do not require to store individual tuples. For Stream Slicing and Buckets, the memory footprint increases linearly with the number of slices in the allowed lateness. The memory footprint is independent from the number of tuples. The opposite holds for Tuple Buffers and Aggregate Trees. Slicing techniques store just one partial aggregate per slice, while buckets store one partial aggregate per window. Tuple Buffers and Aggregate Trees store each tuple individually.



(a) Raising slices/time with Time-Based Windows

(b) Raising tuples/time with Time-Based Windows

(c) Raising slices/time with Count-Based Windows

(d) Raising tuples/time with Count-Based Windows

Figure 10: Memory Experiments with Unordered Streams.

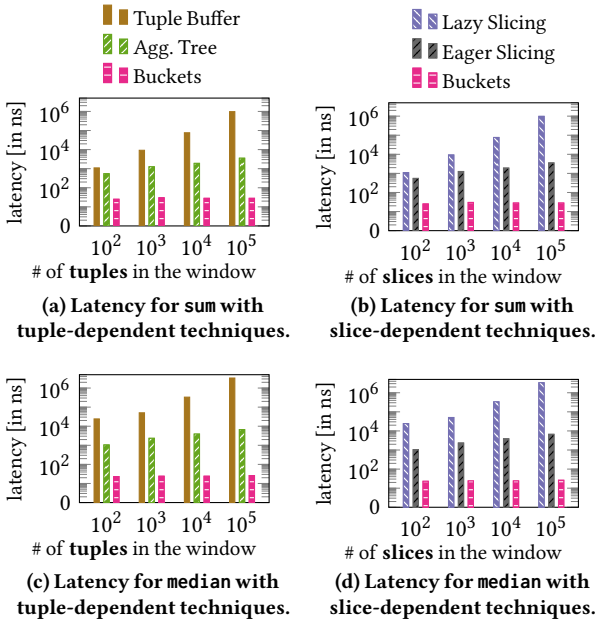
Results for Count-Based Windows. Figures 10c and 10d show the memory consumption for count-based windows, which require individual tuples to be stored. The experiment setup is the same as in Figures 10a and 10b.

The memory consumption of all techniques increases with the number of tuples in the allowed lateness because we need to store all tuples for processing count-based windows on out-of-order streams (Figure 10d). Starting from 1000 tuples in the allowed lateness, the memory consumed by tuples dominates the overall memory requirement. Accordingly, all curves become linear and parallel. Buckets show a stair shape because of the underlying hash map implementation [49]. Slicing techniques start at roughly 10^5 byte which is the space required to store 500 slices. The memory footprint of buckets also increases with the number of slices because more slices correspond to more window buckets (Figure 10c). Each bucket stores all tuples it contains which leads to duplicated tuples for overlapping buckets.

Summary. When we can drop individual tuples and store partial aggregates only (Figure 10a and 10b), the memory consumptions of slicing and buckets depends only on the number of slices in the allowed lateness. In this case, stream slicing and buckets scale to high ingestion rates with almost constant memory utilization. If we need to keep individual tuples (Figure 10c and 10d), storing tuples dominates the memory consumption.

6.2.4 Latency. The output latency for window aggregates depends on the aggregation technique, the number of entries (tuples or slices) which are stored, and the aggregation function. In Figure 11, we show the latency for different situations.

Distributive and Algebraic Aggregation. For the sum aggregation (Figure 11a), Lazy Slicing and Tuple Buffer exhibit up to 1ms latency for 10^5 entries (no matter if 10^5 tuples or 10^5 slices). Eager Slicing and Aggregate Trees show latencies below 5μs. Buckets achieve latencies below 30ns. Lazy aggregation has higher latencies because it computes final aggregates upon request. Eager



(a) Latency for `sum` with tuple-dependent techniques.

(b) Latency for `sum` with slice-dependent techniques.

(c) Latency for `median` with tuple-dependent techniques.

(d) Latency for `median` with slice-dependent techniques.

Figure 11: Output Latency of Aggregate Stores

Aggregation uses precomputed partial aggregates from an aggregate tree which reduces the latency. Buckets pre-compute the final aggregate of each window and store aggregates in a hash map which leads to the lowest latency.

Holistic Aggregation. The latencies for the holistic median aggregation (Figure 11c) are in the same order of magnitude and follow the same trends. Buckets exhibit the same latencies as before because they precompute the aggregate for each bucket. Thus, a more complex holistic aggregation decreases the throughput but does not increase the latency. The latency of slicing techniques increases for the median aggregation because we combine partial aggregates to final aggregates when windows end. This combine step is more expensive for holistic aggregates than for algebraic ones.

Summary. We observe a trade-off between throughput and latency. Lazy aggregation has the highest throughput and the highest latency. Eager aggregation has a lower throughput but achieves microsecond latencies. Buckets provide latencies in the order of nanoseconds but have an order of magnitude less throughput.

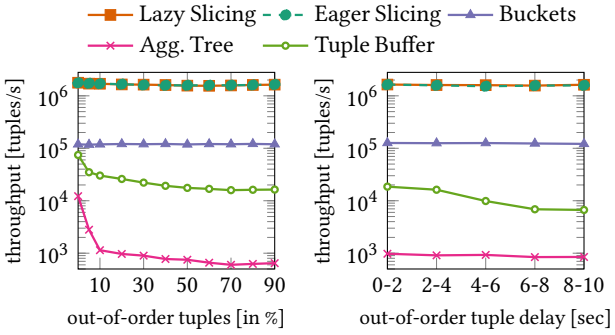
6.3 Studying Workload Characteristics

We measure the impact of the workload characteristics from Section 4 on the performance of general slicing. For comparison, we also show the best alternative techniques.

6.3.1 Impact of Stream Order. In this experiment, we investigate the impact of the amount of out-of-order tuples and the impact of the delay of out-of-order tuples on throughput (Figure 12). We use the same setup as for the throughput experiments in Section 6.2.2 with 20 concurrent windows.

Out-of-order Performance. In Figure 12a, we increase the fraction of out-of-order tuples. Slicing and Buckets process out-of-order tuples as fast as in-order tuples. The throughput of the other techniques decreases when processing more out-of-order tuples.

Slicing techniques process out-of-order tuples efficiently because they perform only one slice update per out-of-order tuple. Eager slicing also updates its aggregate tree. This update has a low overhead because there are just a few hundred slices in the allowed lateness and, accordingly, there are just a few tree levels



(a) Increasing the fraction of out-of-order tuples. **(b) Increasing the delay of out-of-order tuples.**

Figure 12: Impact of Stream Order on the Throughput.

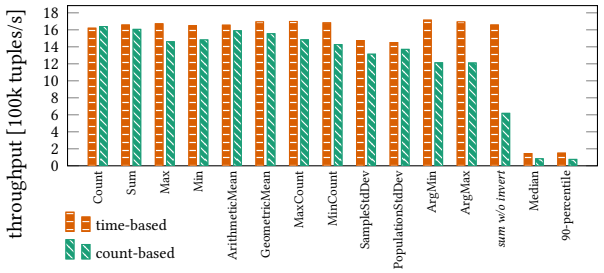
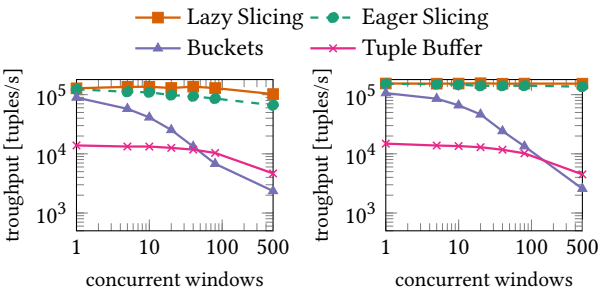


Figure 13: Impact of Aggregation Types on Throughput.



(a) Football data set [34]. **(b) Machine data set [25].**

Figure 14: Throughput for Median Aggregation.

which require updates. Aggregate Trees on tuples have a much larger number of tree levels because they store tuples instead of slices as leaf nodes.

Buckets have a constant throughput as in the previous experiments. Tuple Buffers and Aggregate Trees exhibit a throughput decay when processing out-of-order tuples. Tuple Buffers require expensive out-of-order inserts in the sorted buffer array. Aggregate Trees require inserting past leaf nodes in the aggregate tree. This causes a rebalancing of the tree and the respective recomputation of aggregates. Eager Slicing seldom faces this issue (see Section 6.2.2).

Delay Robustness. In Figure 12b, we increase the delay of out-of-order tuples. We use equally distributed random delays within the ranges specified on the horizontal axis.

All techniques except Tuple Buffers are robust against increasing delays. Slicing techniques always update one slice when they process a tuple. Small delays can sightly increase the throughput compared to longer delays if out-of-order tuples still belong to the most recent slice. In this case, we require no lookup operations to find the correct slice. The throughput of Buckets is independent of the delay because Flink stores buckets in a hashmap. The throughput of the tuple buffer decreases with increasing delay of out-or-order tuples, because the lookup and update costs in the sorted buffer array increase.

Summary. Stream slicing and Buckets scale with constant throughput to large fractions of out-of-order tuples and are robust against high delays of these tuples.

6.3.2 Impact of Aggregation Functions. We now study the throughput of different aggregation functions using the same setup as before (20 concurrent windows, 20% out-of-order tuples, delays between 0 and 2 seconds) in Figure 13. We differentiate time-based and count-based windows to show the impact of invertibility. We implement the same aggregation functions as Tangwongsang et al. [42]. The original publication provides a discussion of these functions and an overview of their algebraic properties. We additionally study the median and the 90-percentile as examples for holistic aggregation. Moreover, we study a naive version of the sum aggregation which does not use the invertibility property. This allows for making a deduction with respect to not invertible aggregations in general.

Time-Based Windows. For time-based windows, the throughput is similar for all algebraic and distributive aggregations with small differences due to different computational complexities of the aggregations. Holistic aggregations (median and 90-percentile) show a much lower throughput because they require to keep all tuples in memory and have a higher complexity.

Count-Based Windows. We observe lower throughputs than for time-based windows, which is because of out-of-order tuples. For count-based windows, an out-of-order tuple changes the sequence id (count) of all later tuples. Thus, we need to shift the last tuple of each slice to the next slice. This operation has low overhead for invertible aggregations because we can subtract and add tuples from aggregates. The operation is costly for not invertible aggregations because it requires the recomputation of the slice aggregate. Time-based windows do not require an invert operation because out-of-order tuples only change the sequence id (count) of later tuples but not the timestamps.

Impact of invertibility. There is a big difference between the performance for different not invertible aggregations on count-based windows. Although `Min`, `Max`, `MinCount`, `MaxCount`, `ArgMin`, and `ArgMax` are not invertible, they have a small throughput decay compared to time-based windows (Figure 13). This is because most invert operations do not affect the aggregate and, thus, do not require a recomputation. For example, it is unlikely that the tuple we shift to the next slice is the maximum of the slice. If the maximum remains unchanged, `max`, `MaxCount`, and `ArgMax` do not require a recomputation. In contrast, the `sum w/o invert` function shows the performance decay for a not invertible function which always requires a recomputation when removing tuples.

Impact of Holistic Aggregations. In Figure 13, we observe that holistic aggregations have a much lower throughput than algebraic and distributive aggregations. In Figure 14, we show that stream slicing still outperforms alternative approaches for these aggregations. The reason is that stream slicing prevents redundant computations for overlapping windows by sorting values within slices and by applying run length encoding. In contrast, Buckets and Tuple Buffer compute each window independently. The machine data set shows slightly higher throughputs because the aggregated column has only 37 distinct values compared to 84232 distinct values in the football dataset. Fewer distinct values increase the savings achieved by run length encoding. Aggregate trees (not shown) can hardly compute holistic aggregates. They maintain partial aggregates for all inner nodes of a large tree which is extremely expensive for holistic aggregations.

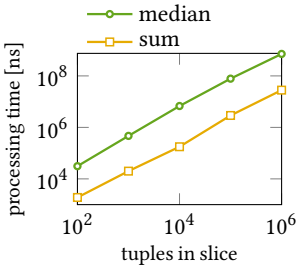


Figure 15: Processing Time for Recomputing Aggregates.

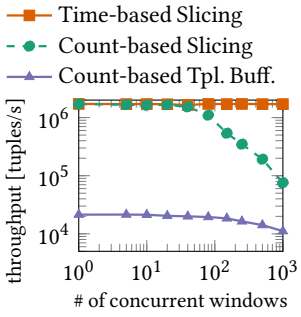
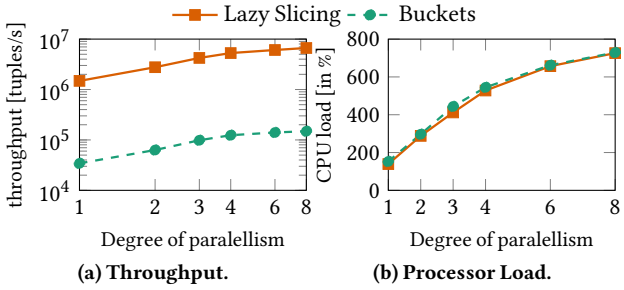


Figure 16: The Impact of Different Window Measures.



(a) Throughput. **(b) Processor Load.**

Figure 17: Parallelizing the workload of a live-visualization dashboard (80 concurrent windows per operator instance).

Summary. On time-based windows, stream slicing performs diverse distributive and algebraic aggregations with similarly high throughputs. Considering count-based windows and out-of-order tuples, invertible aggregations lead to higher throughputs than not invertible ones.

6.3.3 Impact of Window Types. The window type impacts the throughput if we process context-aware windows because these windows potentially require split operations. Note that context aware windows cover arbitrary user-defined windows which makes it impossible to provide a general statement on the throughput for all these windows. Thus, we evaluate the time required to recompute aggregates for slices of different sizes when a split operation is performed (Figure 15). Given a context aware window, one can estimate the throughput decay based on the number of split operations required and the time required for recomputing aggregates after splits. We show the sum aggregation as representative for an algebraic function and the median as example for a holistic function.

The processing time for the recomputation of an aggregate increases linearly with the number of tuples contained in the aggregate. If split operations are required to process a context aware window, a system should monitor the overhead caused by split operations and adjust the maximum size of slices accordingly. Smaller slices require more memory and cause repeated aggregate computation when calculating final aggregates for windows. In exchange, the aggregates of smaller slices are cheaper to recompute when we split slices.

6.3.4 Impact of Window Measures. We compare different window measures in Figure 16. We use the same setup as before (20% out-of-order tuples with delays between 0 and 2 seconds).

Time-Based Windows. For time-based windows, the throughput is independent from the number of concurrent windows as discussed in our throughput analysis in Section 6.2.2. The throughput for arbitrary advancing measures is the same as for time-based measures because they are processed identically [10].

Count-Based Windows. The throughput for count-based windows is almost constant for up to 40 concurrent windows and decays linearly for larger numbers. For up to 40 concurrent windows, most slices are larger than the delay of tuples. Thus, out-of-order tuples still belong to the current slice and require no slice updates. The more windows we add, the smaller our slices become. Thus, out-of-order tuples require an increasing number of updates for shifting tuples between slices which reduces the throughput. Tuple buffers are the fastest alternative to Slicing in our experiment. For 1000 concurrent windows, slicing is still an order of magnitude faster than tuple buffers.

Summary. The throughput of time-based windows stays constant whereas the throughput of count-based windows decreases with a growing number of concurrent windows.

6.4 Parallel Stream Slicing

In this experiment, we study stream slicing on the example of our dashboard application [45] which uses the M4 aggregation [26]. We vary the degree of parallelism to show the scalability with respect to the number of cores. We compare Lazy Slicing with Buckets which are used in Flink.

Results. In Figure 17, we increase the number of parallel operator instances of the windowing operation (degree of parallelism). The throughput scales linearly up to a degree of parallelism of four (Figure 17a). Up to this degree, each parallel operator instance runs on a dedicated core with other tasks (data source operator, writing outputs, operating system overhead, etc.) running on the remaining four cores. For higher degrees of parallelism the throughput and the CPU load increase logarithmically, approaching the full 800% CPU utilization (Figure 17b). Slicing achieves an order of magnitude higher throughput than buckets, because it prevents assigning tuples to multiple buckets (cf. Section 6.2.1). The memory consumption scaled linearly with the degree of parallelism for both techniques.

Summary. We conclude that stream slicing and buckets scale linearly with the number of cores for our application.

7 RELATED WORK

Optimizing Window Aggregations. Our general slicing techniques utilizes features of existing techniques such as on-the-fly slicing [28], incremental aggregation [42], window grouping [18, 19], and user-defined windows [10]. However, general stream slicing offers a unique combination of generality and performance. We base our general slicing implementation on a specialized technique which we presented earlier as a poster [46]. One can extend other slicing techniques based on this paper to reach similar generality and performance. Existing slicing techniques such as Pairs [28] and Panes [30] are limited to tumbling and sliding windows. Cutty can process user-defined window types, but does not support out-of-order processing [10]. Several publications optimize sliding window aggregations focusing on different aspects such as incremental aggregation [6, 15, 42] or worst-case constant time aggregation [43]. Hirzel et al. conclude that one needs to decide on a concrete algorithm based on the aggregation, window type, latency requirements, stream order, and sharing requirements because each specialized algorithm addresses a different set of requirements [22]. Instead of alternating between

different algorithms, we provide a single solution which is generally applicable and allows for adding aggregation functions and window types without changing the core of our technique.

Stream Processing in Batches. In contrast to our techniques, which adopts a tuple-at-a-time processing approach, several works split streams in batches of data which they process in parallel [5, 27, 52]. SABER introduces *window fragments* to decouple *slide* and *range* of sliding windows from the batch size [27]. However, in contrast to our work, SABER does not consider aggregate sharing among queries. Balkesen et al. use panes to share aggregates among overlapping windows [5]. None of these works addresses the general applicability with respect to workload characteristics.

Complementary Techniques. Weaving optimizes execution plans to reduce the overall computation costs for concurrent window aggregate queries [18, 19, 38]. We use a similar approach to fuse window aggregation queries when window edges match. This optimization is orthogonal to the generalization of slicing which is the focus of this paper. Huebsch et al. study multiple query optimization when aggregating several data streams which arrive at different nodes [24]. General stream slicing complements this work with an increased per-node performance. Truviso proposes an alternative technique based on independent stream partitions to correct outputs when tuples arrive after the watermark [29]. While our work focuses on slicing streams and computing partial aggregations for slices, recent publications of Shein et al. further accelerate the final aggregation step which is required when windows end [39, 40]. Trill [11] is an analytics system that supports streaming, historical, and exploratory queries in the same system. Trill supports incremental aggregation and performs aggregations on snapshots, the state of the window at a certain point in time.

8 CONCLUSION

Stream slicing is a technique for streaming window aggregation which provides high throughputs and low latencies with a small memory footprint. In this paper, we contribute a generalization of stream slicing with respect to four key workload characteristics: Stream (dis)order, aggregation types, window types, and window measures. Our general slicing technique dynamically adapts to these characteristics, for example, by exploiting the invertibility of an aggregation or the absence of out-of-order tuples.

Our experimental evaluation reveals that general slicing is highly efficient without limiting generality. It scales to a large number of concurrent windows, and consistently outperforms state-of-the-art techniques in terms of throughput. Furthermore, it efficiently supports application scenarios with large fractions of out-of-order tuples, tuples with high delays, time-based and count-based window measures, context-aware windowing, and holistic aggregation functions. Finally, we observed that the throughput of general slicing scales linearly with the number of processing cores.

Acknowledgements: This work was funded by the EU project E2Data (780245), Delft Data Science, and the German Ministry for Education and Research as BBDC I (01IS14013A) and BBDC II (01IS18025A).

REFERENCES

[1] Tyler Akidau et al. 2015. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing. In *PVLDB*.
[2] Apache Beam. 2018. An advanced unified programming model. https://beam.apache.org/ (project website).
[3] Arvind Arasu et al. 2004. Resource sharing in continuous sliding-window aggregates. *VLDB*.
[4] Michael Armbrust et al. 2018. Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark. In *SIGMOD*. ACM, 601–613.
[5] Cagri Balkesen et al. 2011. Scalable data partitioning techniques for parallel sliding window processing over data streams. In *DMSN*.
[6] Pramod Bhatotia et al. 2014. Slider: incremental sliding window analytics. In *ACM/IFIP/USENIX Middleware*.
[7] Brice Bingman. 2018. Poor performance with Sliding Time Windows. In *Flink Jira Issues (issues.apache.org/jira/browse/FLINK-6990)*.
[8] Irina Botan et al. 2010. SECRET: a model for analysis of the execution semantics of stream processing systems. In *PVDLB*.
[9] Paris Carbone et al. 2015. Apache Flink: Stream and batch processing in a single engine. *IEEE CS* (2015).
[10] Paris Carbone et al. 2016. Cutty: Aggregate Sharing for User-Defined Windows. In *CIKM*.
[11] Badrish Chandramouli et al. 2014. Trill: A High-performance Incremental Query Processor for Diverse Analytics. *PVLDB* 8, 4 (2014), 401–412.
[12] Sanket Chintapalli et al. 2016. Benchmarking streaming computation engines: Storm, Flink and Spark streaming. In *IPDPS*.
[13] Xenofontas Dimitropoulos et al. 2009. On the 95-percentile billing method. In *PAM*.
[14] Buğra Gedik. 2014. Generic windowing support for extensible stream processing systems. *SPE* (2014).
[15] Thanaa Ghanem et al. 2007. Incremental evaluation of sliding-window queries over data streams. In *TKDE*.
[16] Jim Gray et al. 1997. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *DMKDFD* (1997).
[17] Michael Grossniklaus et al. 2016. Frames: data-driven windows. In *DEBS*.
[18] Shenoda Guirguis et al. 2011. Optimized processing of multiple aggregate continuous queries. In *CIKM*.
[19] Shenoda Guirguis et al. 2012. Three-Level Processing of Multiple Aggregate Continuous Queries. In *IEEE ICDE*.
[20] Martin Hirzel et al. 2009. SPL stream processing language specification. *IBM Research Report* (2009).
[21] Martin Hirzel et al. 2014. A Catalog of Stream Processing Optimizations. *Comput. Surveys* 46 (2014).
[22] Martin Hirzel et al. 2017. Sliding-Window Aggregation Algorithms: Tutorial. In *DEBS*.
[23] Kartik Hosanagar et al. 2008. Service adoption and pricing of content delivery network (CDN) services. *INFORMS MSCIAM* (2008).
[24] Ryan Huebsch et al. 2007. Sharing aggregate computation for distributed queries. In *SIGMOD*.
[25] Zbigniew Jerzak et al. 2012. The DEBS 2012 grand challenge. In *DEBS*.
[26] Uwe Jugel et al. 2014. M4: a visualization-oriented time series data aggregation. *PVLDB* (2014).
[27] Alexandros Koliousis et al. 2016. SABER: Window-based hybrid stream processing for heterogeneous architectures. In *SIGMOD*.
[28] Sailesh Krishnamurthy et al. 2006. On-the-fly sharing for streamed aggregation. In *SIGMOD*.
[29] Sailesh Krishnamurthy et al. 2010. Continuous analytics over discontinuous streams. In *SIGMOD*.
[30] Jin Li et al. 2005. No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. In *SIGMOD Record*.
[31] Jin Li et al. 2005. Semantics and evaluation techniques for window aggregates in data streams. In *SIGMOD*.
[32] Jin Li et al. 2008. AdaptWID: An adaptive, memory-efficient window aggregation implementation. *IICOFX* (2008).
[33] Jin Li et al. 2008. Out-of-order processing: a new architecture for high-performance stream systems. In *PVLDB*.
[34] Christopher Mutschler et al. 2013. The DEBS 2013 grand challenge. In *DEBS*.
[35] OpenJDK. 2018. JMH Benchmarking Suite Project Website. goo.gl/TPQSDw.
[36] OpenJDK. 2018. Nashorn Project, ObjectSizeCalculator. goo.gl/962BfX.
[37] David Salomon. 2007. *Variable-length codes for data compression*. Springer Science & Business Media.
[38] Anatoli U. Shein et al. 2015. F1: Accelerating the Optimization of Aggregate Continuous Queries. In *CIKM*.
[39] Anatoli U Shein et al. 2017. Flatfit: Accelerated incremental sliding-window aggregation for real-time analytics. In *SSDBM*.
[40] Anatoli U. Shein et al. 2018. SlickDeque: High Throughput and Low Latency Incremental Sliding-Window Aggregation. In *EDBT*.
[41] Leo Syinchwun. 2016. Lightweight Event Time Window. In *Flink Jira Issues (issues.apache.org/jira/browse/FLINK-5387)*.
[42] Kanat Tangwongsan et al. 2015. General incremental sliding-window aggregation. In *PVLDB*.
[43] Kanat Tangwongsan et al. 2017. Low-Latency Sliding-Window Aggregation in Worst-Case Constant Time. In *DEBS*.
[44] Ankit Toshniwal et al. 2014. Storm@ twitter. In *SIGMOD*.
[45] Jonas Traub et al. 2017. I2: Interactive Real-Time Visualization for Streaming Data.. In *EDBT*.
[46] Jonas Traub et al. 2018. Scotty: Efficient Window Aggregation for out-of-order Stream Processing. In *ICDE*.
[47] Peter Tucker et al. 2003. Exploiting punctuation semantics in continuous data streams. In *TKDE*.
[48] Kostas Tzoumas et al. 2015. High-throughput, low-latency, and exactly-once stream processing with Apache Flink. (2015). goo.gl/QdTkEq.
[49] Mikhail Vorontsov. 2013. Memory consumption of popular Java data types - part 2. *Java Performance Tuning Guide* (2013). goo.gl/7CuJtf.
[50] Jark Wu. 2017. Improve performance of Sliding Time Window with pane optimization. In *Flink Jira Issues (issues.apache.org/jira/browse/FLINK-7001)*.
[51] Yuan Yu et al. 2009. Distributed aggregation for data-parallel computing: interfaces and implementations. In *SIGOPS*.
[52] Matei Zaharia et al. 2013. Discretized streams: Fault-tolerant streaming computation at scale. In *SOSP*. ACM.