Scotty: General and Efficient Open-Source Window Aggregation for Stream Processing Systems

JONAS TRAUB, Technische Universität Berlin, Germany
PHILIPP MARIAN GRULICH, Technische Universität Berlin, Germany
ALEJANDRO RODRÍGUEZ CUÉLLAR, Galápago Agroconsultores S.A.S., Colombia
SEBASTIAN BRESS, Technische Universität Berlin, Germany
ASTERIOS KATSIFODIMOS, Delft University of Technology, Netherlands
TILMANN RABL, HPI, University of Potsdam, Germany
VOLKER MARKL, Technische Universität Berlin & DFKI, Germany

Window aggregation is a core operation in data stream processing. Existing aggregation techniques focus on reducing latency, eliminating redundant computations, or 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. In this paper, we present *Scotty*, an efficient and general open-source operator for sliding-window aggregation in stream processing systems, such as Apache Flink, Apache Beam, Apache Samza, Apache Kafka, Apache Spark, and Apache Storm. One can easily extend Scotty with user-defined aggregation functions and window types. Scotty implements the concept of general stream slicing and derives workload characteristics from aggregation queries to improve performance without sacrificing its general applicability. We provide an in-depth view on the algorithms of the general stream slicing approach. Our experiments show that Scotty outperforms alternative solutions by up to one order of magnitude.

CCS Concepts: • **Information systems** → **Stream management**; *Data streams*; **Data streaming**; Query operators; • **Theory of computation** → Streaming models.

Additional Key Words and Phrases: Window, Aggregation, Sliding-Window, Session Window, Tumbling Window, Aggregate Sharing, Open-Source, Stream Processing, Scotty, Apache Flink, Apache Storm, Apache Samza, Apache Beam, Apache Spark, Apache Kafka Streams

Authors' addresses: Jonas Traub, jonas.traub@tu-berlin.de, Technische Universität Berlin, Sekr. EN-7, Einsteinufer 17, 10587 Berlin, Germany; Philipp Marian Grulich, grulich@tu-berlin.de, Technische Universität Berlin, Sekr. EN-7, Einsteinufer 17, 10587 Berlin, Germany; Alejandro Rodríguez Cuéllar, a.rodriguez@galapagoagro.co, Galápago Agroconsultores S.A.S., Bicaramanga, Colombia; Sebastian Breß, sebastian.bress@tu-berlin.de, Technische Universität Berlin, Sekr. EN-7, Einsteinufer 17, 10587 Berlin, Germany; Asterios Katsifodimos, a.katsifodimos@tudelft.nl, Delft University of Technology, Building 28, E080, Van Mourik Broekmanweg 6, 2628 XE Delft, Netherlands; Tilmann Rabl, tilmann.rabl@hpi.de, HPI, University of Potsdam, August-Bebel-Str. 88, 14482 Potsdam, Germany; Volker Markl, volker.markl@tu-berlin.de, Technische Universität Berlin & DFKI, Sekr. EN-7, Einsteinufer 17, 10587 Berlin, Germany.

111

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. The transformation from streams to windows is called *stream discretization*.

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 [11, 56, 72]. In this paper, we contribute a general solution that 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 that adopt a tuple-at-a-time processing model (e.g., Apache Storm [61], Apache Flink [2, 16], and other Apache Beam-based systems [1, 4]).

To calculate aggregates of overlapping windows, the database community has been working on *aggregation techniques* such as B-Int [5], Pairs [42], Panes [43], RA [59] and Cutty [17]. 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 litera-ture 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 that is applicable to many types of aggregation workloads. At the same time, the operator should be as efficient as specialized techniques that support selected workloads only. In order to provide such an operator, we have implemented *Scotty*. Scotty is available as Open Source Project[1] under the Apache 2.0 license and, right now, provides connectors for Apache Flink [16], Apache Storm [61], Apache Beam [1], Apache Samza [48], Apache Kafka Streams [40], and Apache Spark Continous Processing [60, 75]. Scotty serves as general operator for many systems and provides extension points to make it easy to add new window types and aggregation functions.

In this paper, we discuss the core concept of Scotty: *General Stream Slicing*. To this end, we classify existing aggregation techniques with respect to their underlying concepts and their applicability (Section 3). We then identify and define the workload characteristics which may or may not be supported by existing specialized window aggregation techniques (Section 4). 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 identify stream slicing as a concept on top of which window aggregation can be implemented efficiently. With Scotty, we contribute *a general stream slicing technique* (Section 5). Existing slicing-based techniques do not support complex window types such as session windows [42, 43], do not consider out-of-order processing [17], or limit the type of aggregation functions [17, 42, 43]. With Scotty, we provide a single, generally applicable, and highly efficient approach for streaming window aggregation. General stream slicing inherits the performance of specialized techniques that use stream slicing and generalizes stream slicing to support diverse workloads. Because we integrate all workloads into one general solution, we enable aggregate sharing among all queries with different window types (sliding, sessions, user-defined, etc.) and window measures (e.g., tuple-count or time).

---

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

Scotty breaks down slicing into three operations on slices, namely `merge`, `split`, and `update`. Specific workload characteristics influence the cost of each operation 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 Scotty 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.

This paper is an extended version of two earlier publications presented at EDBT [65] and ICDE [64]. We integrate both publication into one consistent manuscript and make the following new contributions: We focus on the progress made in the open-source Scotty project and provide more detailed explanations and additional insights. We also add the new Sections 7, 8, and 9. Sections 7 and 8 provide an in-depth discussion and formal specification of the algorithms used in Scotty. Section 9 shows examples that illustrate how to use Scotty in different stream processing systems and how to extend Scotty with new aggregation functions and window types.

The remainder of this paper is structured as follows:

- We define terminology with respect to window types, stream order, timing, and data expiration, which we use throughout the paper (Section 2).
- We survey different window aggregation concepts and identify their limitations with respect to different workloads (Section 3).
- We identify the workload characteristics that impact the applicability and performance limitations of existing aggregation techniques (Section 4).
- We contribute *general stream slicing*, a generally applicable and highly efficient solution for streaming window aggregation in dataflow systems (Section 5).
- We take a close look on *session windows* and show how concurrent queries using session windows can benefit from general stream slicing (Section 6).
- We present the algorithms and optimizations used in the two main components of of Scotty, namely the *Stream Slicer* (Section 7) and the *Slice Manager* (Section 8).
- We show several examples that illustrate how one can use Scotty in different stream processing systems and how one can extend Scotty with new aggregation functions and window types (Section 9).
- We evaluate the performance implications of different workload characteristics and show that general stream slicing is generally applicable while offering better performance than existing approaches (Section 10).

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 that 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 (stream discretization). There exist diverse window types ranging from common sliding windows to more complex data-driven windows [24]. 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
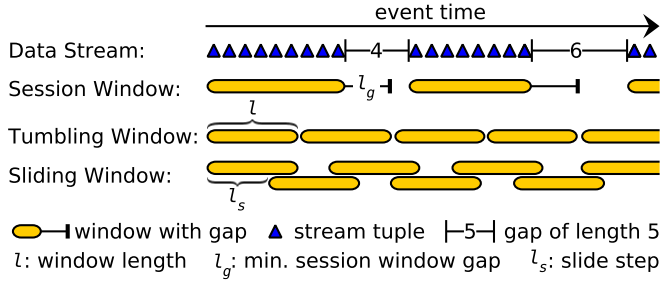
Fig. 1. Common Window Types.

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, 16]. 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, 46]. 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 $s_1, s_2, s_3, \dots$ 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 $\nexists y : t_e(s_y) > t_e(s_x) \wedge y < x$.
- 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 [68]. 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 [4]. *Window punctuations* mark window starts and endings in the stream [21, 30]. 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 [5, 42, 73]. In addition, one can compute partial aggregates incrementally when tuples arrive [59]. 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. We say that a window *ends* when the systems has to output the aggregate for a window. When processing in-order streams, a window *ends* as soon as the time progresses beyond the end-timestamp of the window. When processing out-of-order streams, a window *ends* as soon as the watermark progresses beyond the end-timestamp of the window.

	Memory Usage	**Example**
1. Tuple Buffer	$\lvert\blacktriangle\rvert\cdot\text{size}(\blacktriangle)$	
2. Aggregate Tree	$\lvert\blacktriangle\rvert\cdot\text{size}(\blacktriangle)$ $+(\lvert\blacktriangle\rvert-1)\cdot\text{size}(\bullet)$	
3. Aggregate Buckets	$\lvert\text{win}\rvert\cdot\text{size}(\bullet)$ $+\lvert\text{win}\rvert\cdot\text{size}(\text{⌴})$	
4. Tuple Buckets	$\lvert\text{win}\rvert\cdot[\text{avg}(\blacktriangle \text{ per win.})$ $\cdot\text{size}(\blacktriangle)+\text{size}(\text{⌴})]$	
5. Lazy Slicing	$\lvert\text{⬭}\rvert\cdot\text{size}(\text{⬭})$	
6. Eager Slicing	$\lvert\text{⬭}\rvert\cdot\text{size}(\text{⬭})$ $+(\lvert\text{⬭}\rvert-1)\cdot\text{size}(\bullet)$	
7. Lazy Slicing on tuples	$\lvert\blacktriangle\rvert\cdot\text{size}(\blacktriangle)$ $+\lvert\text{⬭}\rvert\cdot\text{size}(\text{⬭})$	
8. Eager Slicing on tuples	$\lvert\blacktriangle\rvert\cdot\text{size}(\blacktriangle)$ $+\lvert\text{⬭}\rvert\cdot\text{size}(\text{⬭})$ $+(\lvert\text{⬭}\rvert-1)\cdot\text{size}(\bullet)$	

Legend: ▲ Tuple   ● Aggregate   ⬭ Slice with Aggregate   ⌴ Bucket

Table 1. Memory Usage and Visualization 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 that do not process out-of-order tuples, store values for the duration of the longest window. All presented concepts process a single input stream. However, one can merge (join) two or more data streams in a preceding operator and apply windowing on the merged stream. If windows depend on the stream from which a tuple originates, one can label tuples when merging streams.

Table 1 provides an overview of all techniques we discuss in the following subsections. We denote the number of values (i.e., tuples) as $\lvert\blacktriangle\rvert$, the number of slices as $\lvert\text{⬭}\rvert$, and the number of

windows in the allowed lateness as |win|. We further denote the size of a tuple in bytes as $\text{size}(\blacktriangle)$, the size of a slice including an aggregate as $\text{size}(\text{⬤})$, the size of an aggregate as $\text{size}(\bullet)$, and the size of a bucket as $\text{size}(\smile)$. The size of slices and buckets covers metadata such as their start and end timestamps and hashes. The metadata is of equal size for all buckets and slices.

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 that 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. Thus, all aggregate computations contribute to the latency when the window ends.

A tuple buffer stores all tuples for the allowed lateness, which is $|\blacktriangle| \cdot \text{size}(\blacktriangle)$. 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.

3.2 Aggregate Trees

Aggregate trees such as FlatFAT [59] and B-INT [5] 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 $\log(|\blacktriangle|)$ updates of partial aggregates in the tree. Thus, the *throughput* is decreases 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 [54].

3.3 Buckets

Li et al. introduce *Window-ID* (WID) [44–46], a bucket-per-window approach which is adopted by many systems with support for out-of-order processing [1, 4, 16]. 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 [46]. Buckets do not utilize aggregate sharing. Instead, they compute aggregates for each bucket independently. Systems can compute aggregates for buckets incrementally [59]. 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 instead of individual tuples to reduce the memory footprint. However, some use-cases (e.g., holistic aggregates over count-based windows) require us to keep individual tuples in memory. 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 $l = 20s$ and $l_s = 2s$ results in 10 overlapping windows (i.e., buckets) at any time. This causes 10 aggregation operations for each input tuple.
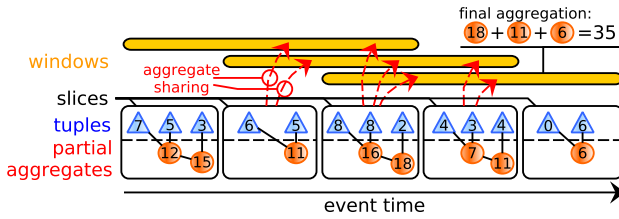
Fig. 2. Example Aggregation with Stream Slicing.

3.4 Stream Slicing

Slicing techniques divide (i.e., *slice*) a data stream into non-overlapping chunks of data (i.e., *slices*) [42, 43]. 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* for in-order as well as out-of-order tuples.

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 ($|\text{⬭}| \ll |\blacktriangle|$) 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.

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. We create slices such that they can be shared among all queries.

4 Workload Characterization

In this section, we identify workload characteristics that 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) [59]. 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 that 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 [17, 59]. We write input values as lower case letters, the operation that adds a value to an aggregate as $\oplus$, and the operation that removes a value from an aggregate as $\ominus$. E.g., if we compute a `sum`, $\oplus$ corresponds to the arithmetic $+$ and $\ominus$ corresponds to the arithmetic $-$. We first adopt three algebraic properties used by Tangwongsan et al. [59]. 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 that share partial aggregates [5, 17, 42, 43, 59]. 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* [23]. 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. For example, an `average` is *algebraic* because `average=sum/count` and partial aggregates can be represented by a tuple $\langle$`sum`, `count`$\rangle$. The remainder of aggregations, which have an unbounded size of partial aggregates, is *holistic*. An example are quantiles (e.g., the `median`) that require to store all input values as part of the intermediate result. General stream slicing is beneficial for distributive, algebraic, and many holistic aggregations.

4.3 Characteristic 3: Windowing Measure

Windows can be specified using different measures (also called *time domains* [12] or `WATTR` [44]). 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 this paper. However, bear in mind that a timestamp can actually be a time, a tuple count, or any other monotonically increasing measure [17]:

- **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* [44] or *tuple-driven* [12]) 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 that have a greater event-time. This changes the aggregates of all count-based windows that start or end after the out-of-order tuple.

If we process multiple queries that 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 [13, 17].
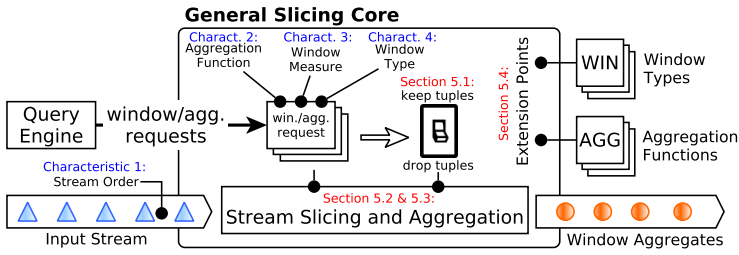
Fig. 3. Architecture of General Stream Slicing.

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. [44]. Here we present those classes along with the most common window types belonging to them.

- **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 [21]. 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 are *Multi-Measure Windows* that 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 that 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
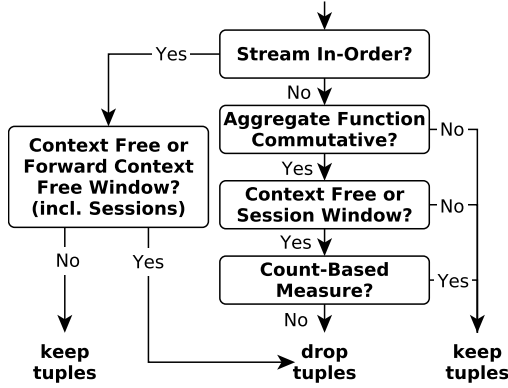
Fig. 4. Decision Tree - Which workload characteristics require
storing individual tuples in memory?

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. Moreover, 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). Queries can be added or removed from the aggregator and due to that, the workload characteristics can change. To this end, our aggregator adapts on the fly. In Section 5.3, we discuss the workflow for processing input tuples and introduce the required software components. General slicing has extension points that can be used to implement user-defined window types and aggregations (discussed in Section 5.4).

We structure the following sections such that we start from an abstract presentation of the concept and continue towards in-depth explanations of each component. Thus, Section 5 presents the concept of general stream slicing. Section 6 discusses how general stream slicing works with forward context aware windows on the example of session windows. The following Sections 7 and 8 then present the involved algorithms in detail, before Section 9 shows programming examples.

5.1 Storing Tuples vs. Partial Aggregates

Existing aggregation techniques achieve generality by storing all input tuples and by computing high-level partial aggregates [5, 59]. 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 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 or to store partial aggregates only.

For example, consider an aggregation function that 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 (CF) and forward context free (FCF) windows but must keep tuples if we process forward context aware (FCA) 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 originally 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, as we will show in Section 6.

(3) **The query uses a count-based window measure.**
An out-of-order tuple (see definition in Section 2) changes the count of all succeeding tuples. Thus, the last tuple of each window shifts to its 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_{start}), its end timestamp (t_{end}), and the timestamp of the first (t_{first}) and last tuple it contains (t_{last}). Note that the timestamps of the first and last tuples do not need to coincide with the start and end timestamps of a slice. For instance, consider a slice A that starts at $t_{\text{start}}(A) = 1$ and ends at $t_{\text{end}}(A) = 10$, but the first (earliest) tuple contained is timestamped as $t_{\text{first}}(A) = 2$ and its last/latest one as $t_{\text{last}}(A) = 9$. Note 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 $t_{\text{end}}(A) \leftarrow t_{\text{end}}(B)$.
(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
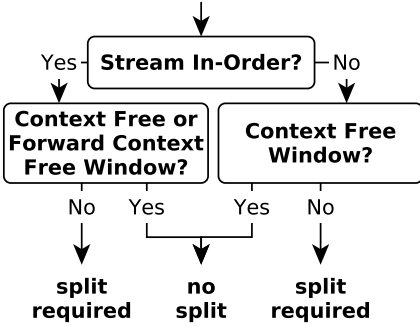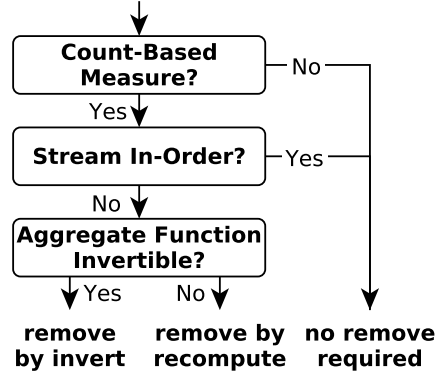
Fig. 5.
Decision Tree:
Are splits required?

Fig. 6.
Decision Tree:
How to remove tuples?

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: $t_{\text{start}}(B) \leftarrow t + 1$ and $t_{\text{end}}(B) \leftarrow t_{\text{end}}(A)$.
(2) Update the end of A such that $t_{\text{end}}(A) \leftarrow t$.
(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 can require split operations because out-of-order tuples contain 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_{start}, t_{end}, t_{first}, and t_{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. 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. Figure 6 depicts 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.
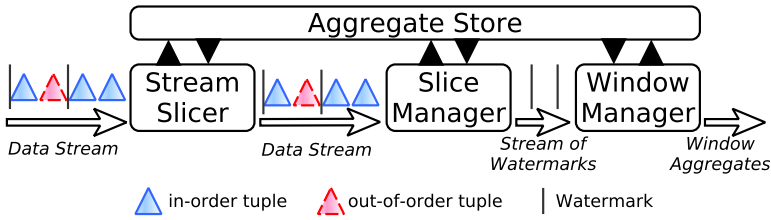
Fig. 7. The Stream Slicing and Aggregation Process.

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 [42]. In an in-order stream, it is sufficient to start slices when windows start [17]. 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 call the beginnings and endings of windows *window edges* and the beginnings and endings of slices *slice edges*. We always keep the timestamp of the next upcoming window edge in memory and compare in-order tuples with this timestamp. As soon as the timestamp of a tuple exceeds the stored timestamp, we start a new slice and save 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 [13, 17, 64].

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 that 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 [59]. For out-of-order tuples, we look up the slice that 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 that 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 that 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 [32] such as Flink [16], Spark [6], and Storm [61]. 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. [59]. 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 $\langle \text{sum} \leftarrow v, \text{count} \leftarrow 1 \rangle$, 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: $\langle \text{sum}, \text{count} \rangle \mapsto \text{sum}/\text{count}$.

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 [20, 33]. For quantile computations, we sort tuples in slices to speed up succeeding merge operations and apply run length encoding to save memory [52].

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 following signature:

```
long getNextEdge(long timestamp)
```

The method 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 subsection 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 that 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 that is implemented by the window specification.
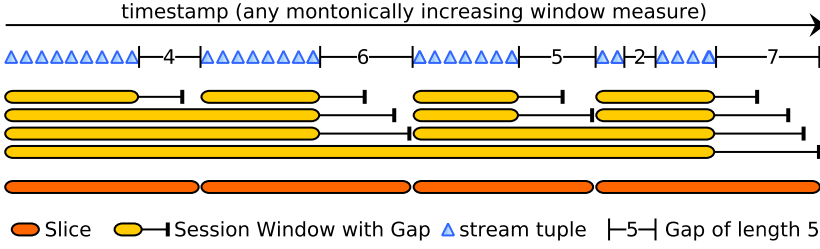
Fig. 8. Session Window Aggregate Sharing.

6 Stream Slicing for Session Windows

Recently, session windows evolved to a common window type supported by programming models such as Apache Beam [4] and the Dataflow Model [1]. Many systems implement these models and process session windows in addition to sliding and tumbling windows [3, 17, 75]. In this section, we take a close look on session windows and show that session window aggregation benefits from stream slicing. In the remainder of the paper, we use session windows as one example of a context aware window type.

6.1 Aggregate Sharing for Session Windows

We show an example for session window stream slicing in Figure 8. The example has four session windows with the minimum gaps $l_g = 3, 5, 6,$ and 7. We make five observations based on our example:

(1) Multiple session window queries with different gaps can share slices and, thus, partial aggregates.
(2) Session windows would also share slices with other types of windows.
(3) Sessions of a single query have no overlap. Thus, a single session window query cannot benefit from aggregate sharing.
(4) Slices can cover the gaps between sessions because gaps do not cover any tuples by definition. Respectively, a partial aggregate that covers a session and a gap is equal to an aggregate that covers the session only.
(5) The slicing logic solely depends on one session window - the one with the smallest gap. All session windows with larger gaps are compositions of the slices made for the session window with the smallest minimum gap. In our example $\min(l_g) = 3$.

We utilize the observations above and create slices with respect to the session window with the smallest gap only. This allows for creating stream slices with a constant workload, which is independent from the number concurrent sessions.

6.2 Session Windows on Out-Of-Order Streams

Stream slicing for session windows is more complex than for sliding or tumbling windows, because session windows are context aware. Thus, we do not know start and end positions of sessions up front. Instead, start and end positions of sessions depend on the gaps between the tuples we process.

Out-of-order tuples either belong to an existing session (`update`), fuse sessions (`merge`), or form new sessions (`split`). We show all cases in Figure 9. Interestingly, we can rewrite all required `split` operations to `update` operations. Thus, we completely prevent expensive slice splits and do not need to store tuples in additions to aggregates when processing session windows.
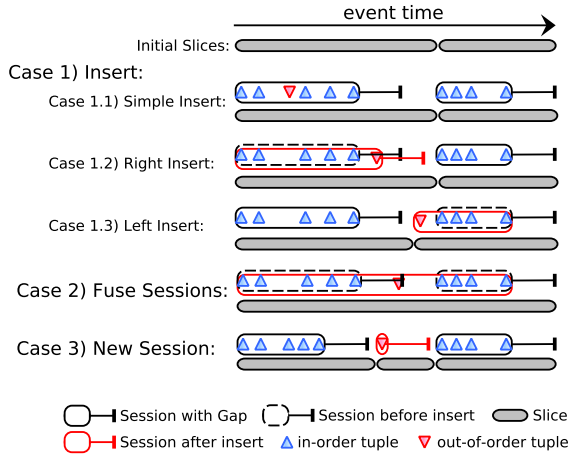
Fig. 9. Out-of-order Processing with Session Windows.

- If an out-of-order tuple belongs to an existing session (Case 1.1) or extends a session at the session end (Case 1.2), we insert the tuple into the respective slice (one update). Thereby, the start and end times of slices remain unchanged.
- If an out-of-order tuple extends a session at the session start (Case 1.3), we change session edges respectively and add the tuple (two updates).
- An out-of-order tuple can also fuse two sessions. This is the case whenever the gap between sessions shrinks below the minimum session gap (Case 2). Fusing sessions also combines the slices of the sessions (one merge).
- Finally, an out-of-order tuple can form a new session on its own if its gap on both sides is larger than the minimum session gap (Case 3). In this case, we split a slice between sessions (i.e., within the gap). Because gaps contain no data by definition, we can create a new slice that contains the out-of-order tuple and and update the end of the existing slice without changing its aggregate (one update).

7 The Stream Slicer

We introduced the architecture and the components of Scotty in Section 5.3 and Figure 7. In the following sections, we present the algorithms used in the *Stream Slicer* and the *Slice Manager* component of Scotty in detail to deepen the understanding of these components.

7.1 Notations and Terminology

We use the following notation in all algorithms presented in the remainder of the paper:

7.1.1 General Nomenclature

e	An event, which can be an in-order tuple, out-of-order tuple, or a watermark.
$t_e(e)$	The event-time of an event e. Note that this could also be any other window measure as discussed in Section 4.3. We refer to event-time in the following sections to ease the explanation of the presented algorithms.
l	The length of a window as introduced in Figure 1.
l_s	The slide step of sliding window as introduced in Figure 1.
gap	The minimum gap which separates sessions as introduced in Section 6.1. If there are multiple session window queries running concurrently, then gap is the smallest gap l_g of all queries (see Figures 1 and 8).

7.1.2 Window Edges

Any beginning or ending of a window is a *window edge*. In Scotty, each window edge marks the start of a new slice. We determine upcoming window edges on-the-fly in the stream slicer.

In general, all event-times of upcoming window edges are known for context free windows. For example, a query with a tumbling window has two parameters which determine all window edges: the length l and the start time t_s of the query execution. At any time t_c, we can compute time of the next window edge after t_c as $t_c + l - (t_c - t_s) \bmod l$. A sliding window has the additional parameter l_s defining the slide step. When determining window edges, a sliding window is equivalent with two tumbling windows: One tumbling window represents all starts of a sliding window (start t_s and length l_s) and the other all window ends (start $t_s + l$ and length l_s).

For context aware windows, windows edges may or may not be known up front and can constantly change. We allow for declaring the next upcoming window edge in the stream slicer based on the processed in-order tuples. In addition, the slice manager can remove, shift, and add window edges for context aware windows.

7.1.3 Slice Separators

We manage window edges as *slice separators*. A *slice separator* is placed between two consecutive tuples and marks the ending of a slice. Thus, several window edges may stick to the same slice separator, e.g., if two windows end or start at the same time. We distinguish three types of separators:

- *Fixed Slice Separators* mark the edges of context free windows such as tumbling and sliding windows. These edges will never be shifted or removed.
- *Flexible Slice Separators* mark the edges of context aware windows such as session windows. These edges may be shifted or removed.
- *Combined Slice Separators* mark coinciding edges of context free and context aware windows, e.g., a session window and a tumbling window ending at the same time.

This differentiation increases the performance of the Slice Manager when processing out-of-order tuples later on as the type of slice separator determines the required actions independent of the underlying window types.

7.2 The Overall Slicing Algorithm

We show the overall algorithm of our Stream Slicer in Algorithm 1. The Stream Slicer operates based on in-order tuples and adds slice separators to start new slices.

Note that we present all algorithms on the example of sliding, tumbling, and session windows as representatives for context free and context aware windows. The presented algorithms generalize to any other window type as well, as they are based on the general concepts of *window edges* and *slice separators* introduced in Section 7.1.

As first step, we check in Line 4 if the current session has timed out. Then, we reset the session timeout depending on the event-time of the current tuple in Line 5 and continue with tumbling and sliding windows. The loop starting in Line 7 emits all fixed separators before $t_e(e)$.

Possibly, we emit another slice separator at $t_e(e)$ that can have any of the three types:

- If a session timed out and a sliding or tumbling window edge lies at $t_e(e)$, we emit a combined separator at $t_e(e)$ in Line 14.
- If only a sliding or tumbling window edge lies at $t_e(e)$, we emit a fixed separator in Line 15.
- If only a session timed out, there will be a flexible separator at $t_e(e)$ in Line 19.

Finally, after emitting all edge separators up to $t_e(e)$, we forward e to the Slice Manager in Line 22.

Algorithm 1 Stream Slicing with Scotty.

State:

    next_edge : Next edge of a tumbling or sliding window.
    session_timeout : Timeout for session window.
    gap : Minimum session window gap.

  1: **upon event** e **do**
  2:    **if** e **is** in-order tuple **then**
  3:        *// Session window slicing*
  4:        $flex\_end \leftarrow (t_e(e) \geq session\_timeout)$;
  5:        $session\_timeout \leftarrow t_e(e) + gap$;
  6:        *// Tumbling and Sliding window slicing*
  7:        **while** $t_e(e) > next\_edge$ **do**
  8:            **emit** fixed separator with time *next_edge*;
  9:            *next_edge* $\leftarrow$ next window edge after *next_edge*;
10:        **end while**
11:        *// Emit remaining separator if needed*
12:        **if** $next\_edge == t_e(e)$ **then**
13:            **if** $flex\_end$
14:                **then emit** combined separator with time $t_e(e)$;
15:                **else emit** fixed separator with time $t_e(e)$;
16:            **end if**
17:            *next_edge* $\leftarrow$ next window edge after *next_edge*;
18:        **else if** $flex\_end$ **then**
19:            **emit** flexible separator with time $t_e(e)$;
20:        **end if**
21:    **end if**
22:    **emit** e; *// Always forward e (after separators)*
23: **end**

7.3 Optimizing the Stream Slicer

We incorporate several optimizations that reduce the per-tuple complexity and make slicing highly efficient:

7.3.1 Sessions Window Fusion. We consider the session window with the smallest gap only as discussed in Section 6.1. This avoids monitoring multiple gaps concurrently.

7.3.2 Tumbling and Sliding Window Fusion. We reduce the number of tumbling window queries with query fusion whenever the lengths of tumbling windows are multiples of each other. For example, assume two tumbling windows A and B with the lengths l^A and l^B and the query start times t_s^A and t_s^B for which $l^B \geq l^A$. If $l^B \bmod l^A = 0$ and $(t_s^B - t_s^A) \bmod l^A = 0$, all edges of B coincide with an edge of A and we can slice based on A only. We apply the same fusion logic for sliding windows because we represent each sliding window with two tumbling windows (Section 7.1.2).

7.3.3 Windows Edges. We keep the timestamp of the next upcoming window edge in memory for each query (Algorithm 2). When a tuple arrives, we compare its event-time with the timestamp of the next window edge to decide if we need to insert a slice separator (Line 7 in Algorithm 1). This ensures that we compute the time of the next window edge after slice separators only, i.e., just once per slice, which allows for scaling to high throughputs and hundreds of queries because we bind the complexity to the number of slices instead of the number of tuples per time (throughput).

Algorithm 2 Handling Window Edges.

State:

   $t_{next}[\ ]$: Array with next edge time of each tumbling window.

   $l[\ ]$: Array with length of each tumbling window.

   $next\_id$: The id of the smallest value in t_{next}.

 1: *// Proceed to the next edge and return the edge timestamp.*
 2: **function** NEXT( )
 3:     $tmp \leftarrow t_{next}[next\_id]$;
 4:     **do**
 5:         $t_{next}[next\_id] \leftarrow t_{next}[next\_id] + l[next\_id]$;
 6:         $next\_id \leftarrow$ id of smallest timestamp in t_{next};
 7:     **while** $t_{next}[next\_id] == tmp$;
 8:     **return** $t_{next}[next\_id]$;
 9: **end function**

10: *// Add a new tumbling window.*
11: **function** ADD_WINDOW(t_s:start time, l_W:window length)
12:     *// Optimization: Query fusion.*
13:     **if** $\exists x : [l_W \bmod l[x] = 0 \wedge (t_s - t_{next}[x]) \bmod l[x] = 0]$
14:         **then exit;** *// All edges already covered.*
15:     **end if**
16:     *// Remove windows if the new one covers their edges.*
17:     $\forall x$ where $[l[x] \bmod l_W = 0 \wedge (t_{next}[x] - t_s) \bmod l_W = 0]$ remove $l[x]$ from l and $t_{next}[x]$ from t_{next};
18:     *// Add new window to the arrays.*
19:     append t_s to t_{next};
20:     append l_W to l;
21:     $next\_id \leftarrow$ id of smallest timestamp in t_{next};
22: **end function**

To prevent repeated computation, Scotty stores the next upcoming slice edge for each currently running query. This binds the size of the arrays used in Algorithm 2 to the number of active queries. We invoke Algorithm 2 in the Stream Slicer (Algorithm 1 in Line 9 and 17).

Algorithm State: The algorithm keeps three variables as state:

(1) $t_{next}[\ ]$ is an array of edge times. It stores the time of the next upcoming edge for each tumbling or sliding window. Note that we represent each sliding window with two tumbling windows as described in Section 7.1.2.

(2) $l[\ ]$ is an array which stores the length of each tumbling window. This allows for calculating the time of the next edge time given the time of the edge before.

(3) $next\_id$ stores the the id of the minimum value in $t_{next}[\ ]$ in order to enable fast access to the time of the next upcoming edge.

Adding Queries and Query Fusion:
We add new window queries through the function ADD_WINDOW (see Algorithm 2). The function receives the start time of the query execution t_s and the length of the tumbling window l_W as parameters. Inside the function, we first check if we can fuse the new query with another one. If there exists a query that covers all edges of the new query already, we exit the function in Line 14 without a need to save the new query. If the new query covers all edges of a query added earlier,
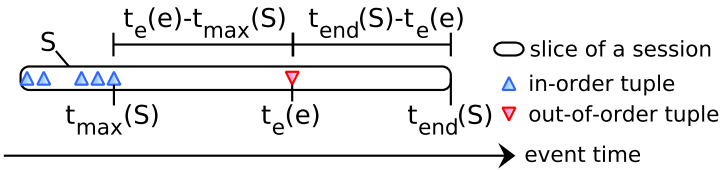
Fig. 10. Nomenclature for the discussion of out-of-order processing in the Slice Manager.

we remove the previous query in Line 17. Finally, we add the new query (i.e., the new window) to the arrays in Lines 19 to 21. Thereby, we update *next_id* accordingly if the new window causes the next edge.

Proceeding to the Next Edge:
Whenever we pass the event-time of an edge in the Stream Slicer, we invoke `NEXT_EDGE` in Algorithm 2. This function returns the time of the next edge and updates the array of stored window edges (l_{next}). Within the function, we first remember the time of the current edge. Then, we update all edges in the array that have the same edge time by adding the lengths of the respective windows. Finally, we return the event-time of the next edge.

8 The Slice Manager

In this section, we present the Slice Manager of Scotty which has three purposes:

(1) It creates new slices when receiving slice separators.
(2) It adds all tuples to the correct slice.
(3) It updates and creates past slices when processing out-of-order tuples.

The Slice Manager provides the slices to the Aggregate Store which is in charge of aggregating values for each slice.

8.1 Notations and Terminology

For the discussion of the Slice Manager, we define the following terminology in addition to the terminology introduce in Section 7.1. We visualize the terminology for the following discussion in Figure 10. Let S be a slice. We name the end time of the slice $t_{end}(S)$ and call the maximum event-time of any tuple contained in the slice $t_{max}(S)$. The event-time of a tuple e is $t_e(e)$ as before.

8.1.1 Active Slice vs. Past Slices
The Slice Manager distinguishes between the *active slice* and *past slices*. The *active slice* starts in the past but ends in the future. An event-time is in the *past* if any event processed so far has a larger event-time. An event-time is in the *future* if all events processed so far have a smaller event-time. All slices before the active slice are *past slices*.

8.1.2 Slice Types
The Slice Manager operates based on slice types instead of window specifications. This decoupling is important for scalability because it allows for adding queries without increasing the per-tuple computation effort. We distinguish three types of slices which map to the types of slice separators introduced in Section 7.1.3:

- 🔴 *Fixed Slices:* The end times of fixed slices are immutable. This guarantees that the Window Manager retains slice separations where context free windows (e.g., tumbling and sliding windows) start or end.
- 🟢 *Flexible Slices:* The end times of flexible slices can change. Thus, the Window Manager can extend flexible slices, shrink them, and fuse them with their successors. This is required to

---

Algorithm 3 Slice Management in Scotty.

State:
    s_c : currently active slice.
    gap : minimum session window gap.

1: **upon event** e **do**
2:      **if** e **is** watermark **then**
3:         forward e to the Window Manager
4:      **else if** e **is** slice separator **then**
5:         set slice type of s_c to type of e
6:         store s_c in the Aggregate Store
7:         $s_c \leftarrow$ new empty slice
8:      **else if** e **is** in-order tuple **or** $e \in s_c$ **then**
9:         add e to s_c
10:     **else if** no session windows **or** $e \in$ active session **then**
11:        add e to the slice which covers $t_e(e)$
12:     **else** *// e is out-of-order tuple and we have sessions*
13:        $S \leftarrow$ session covering $t_e(e)$ including succeeding gap
14:        **if** $t_{end}(S) - t_e(e) > gap$ **and** $t_e(e) - t_{max}(S) > gap$
15:           *// gaps on both sides of e $\Rightarrow$ new session*
16:           **then** split S at $t_e(e)$ and add e
17:        **else if** $t_{end}(S) - t_e(e) > gap$
18:           *// gap on the right of e $\Rightarrow$ right or simple insert*
19:           **then** add e to the slice which covers $t_e(e)$
20:        **else if** $t_e(e) - t_{max}(S) > gap$
21:           *// gap on the left of e $\Rightarrow$ left insert*
22:           **then** move $t_{end}(S)$ to $t_e(e)$ (excl.) and add e
23:        **else** *// no gap separates e $\Rightarrow$ fuse sessions*
24:           **then** fuse S with succeeding session and add e
25:        **end if**
26:     **end if**
27: **end**

---

      update slices when processing out-of-order tuples that change the context of context aware
      windows such as session windows.
- 🅒 *Combined Slices:* Combined slices unite characteristics of fixed and flexible slices. They
  occur when the edge of a context free window coincides with the edge of a context aware
  window. End times of combined slices are immutable. However, a combined slice also marks
  the ending of a context aware window.

8.2 The Overall Slice Management Algorithm

We show the overall algorithm of the Slice Manager in Algorithm 3. The Slice Manager is independent from watermarks. Thus, we forward watermarks to the Window Manager in Line 3. In the following subsections, we discuss how we process slice separators, how we assign tuples to slices, and how we retain correct slices when tuples arrive out-of-order.

8.2.1 Processing Slice Separators.

The Stream Slicer marks the starts of new slices with slice separators. The Slice Manager processes slice separators in three steps: First, it sets the type of the currently active slice with respect to the separator type (fixed, flexible, or combined) in Line 5 of Algorithm 3. Second, after setting the

slice type, the Slice Manager stores the slice in the Aggregate Store which manages all past slices (Line 6). Finally, the Slice Manager resets the currently active slice with a new empty slice (Line 7).

8.2.2 Adding Tuples to Slices

In-order Tuples: In-order tuples always belong to the currently active slice because their event-time cannot be in the past by definition. Thus, we add all in-order tuples to the currently active slice in Line 9 of Algorithm 3. This has low computational costs because we add tuples to their slice without a lookup operation for finding the correct slice. Moreover, the costs are independent from the number of concurrent windows and queries which is crucial for scalability.

Out-of-order Tuples: The computation effort when processing a tuple out-of-order depends on the delay of the tuple. If a tuple has a small delay but still belongs to the currently active slice, we can add the tuple to the active slice just like an in-order tuple (Line 9). If the tuple has a larger delay, we lookup the slice that covers the event-time of the tuple and add the tuple to that slice (Line 11).

8.3 Changing Slices for Out-of-order Tuples

8.3.1 Tumbling and Sliding Windows
For tumbling and sliding windows, we know the times of all window edges a priori. Thus, the Stream Slicer always initiates the correct creation of all slices for these windows. The Slice Manager ensures to retain correct slices for tumbling and sliding windows when changing past slices. In the following, we discuss how out-of-order tuples affects context aware windows on the example of session windows.

8.3.2 Session Windows
When we process session windows, an out-of-order tuple might fuse sessions or add a new session in the past (Recall the discussion of Figure 9 in Section 6.2). Thus, the Slice Manager possibly adds or changes slices in the past before inserting the out-of-order tuple.

Analogue to the Stream Slicer, the Slice Manager operates based on a single session window query - the one with the smallest gap. This is sufficient to maintain all slices for session windows with larger gaps as well (recall Section 6.1).

An out-of-order tuple either belongs to an existing session, fuses sessions, or forms a new session on its own. We show all three cases with the corresponding slice changes in Figure 9.

- *Insert in Existing Sessions:* In case the out-of-order tuple e belongs to an existing session, we insert the tuple in an existing slice. If $t_e(e)$ lies in the middle of a session (Case 1.1 in Figure 9) or extends a session at the session end (Case 1.2 in Figure 9), we insert the tuple in the slice S which covers $t_e(e)$. In both cases, the start and end times of slices remain unchanged. If e extends a session at the start of the session (Case 1.3 in Figure 9), we change $t_{end}(S)$ to $t_e(e)$ (exclusive) and add e afterwards.
- *Fusing Sessions:* Adding e can also fuse two sessions. This is the case whenever adding e shrinks the gap between sessions below the minimum session gap (Case 2 in Figure 9). Fusing sessions also combines the slices of the sessions.
- *Adding new Sessions:* If the gap on both sides of e is larger than the minimum session gap, e forms a new session on its own. In this case we split the slice which covers $t_e(e)$ (Case 3 in Figure 9). The slice before the split ends at $t_e(e)$ (exclusive). The new slice (after the split) starts at $t_e(e)$ (inclusive) and ends at the former end of the slice that we split. We can split S after $t_{max}(S)$ (i.e., within the gap) without changing the aggregate of S because the gap cannot contain any tuples.

Formal Specification: We formalize the rules for all cases depicted in Figure 9 in Algorithm 3. First, we find the session S which covers $t_e(e)$ in Line 13 (including the gap after the session). Then, we check if there are gaps larger than the minimum gap on both sides of e in Line 14 (Case 3 in Figure 9). If this is not the case, we check the gap after e in Line 17 (Cases 1.1 and 1.2 in Figure 9).
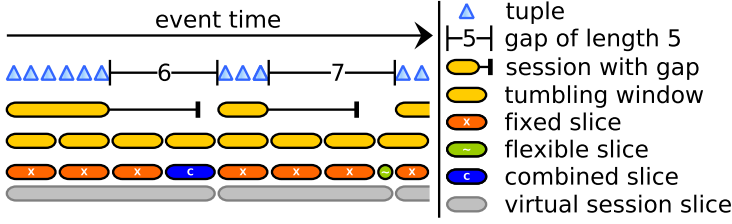
Fig. 11. Joint Processing for Multiple Queries.

(a) Splitting sessions.

(b) Fusing sessions.

(c) Left extend with $t_e(e)$ at
the session end.

(d) Left extend with
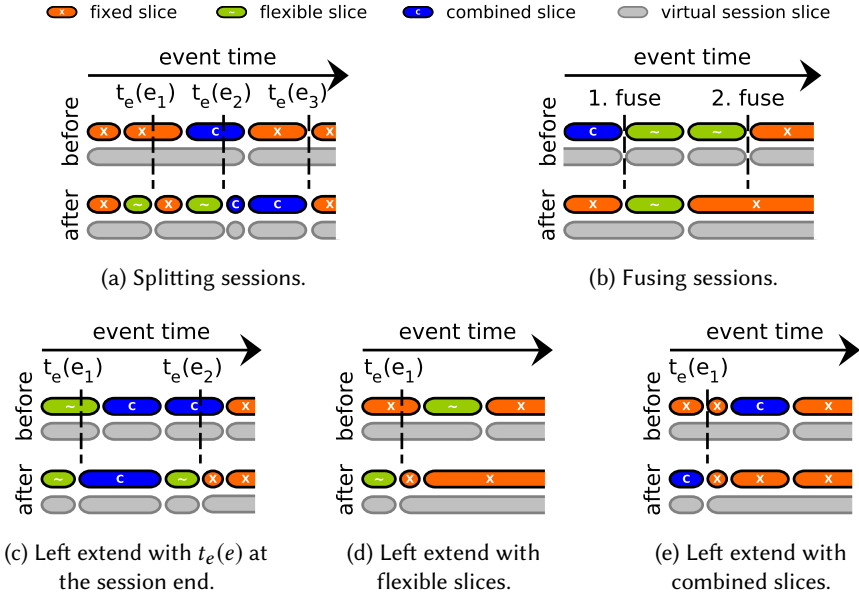flexible slices.

(e) Left extend with
combined slices.

Fig. 12. Slice Management for Out-of-order Tuples with Multiple Sliding, Tumbling, and Session Windows.

Finally, we check the gap before e in Line 20 (Case 1.3 in Figure 9). If no gap remains larger than the minimum gap which separates sessions, we fuse sessions in Line 24 (Case 2 in Figure 9).

8.4 Joint Slice Management for all Window Types

In the following, we discuss how Scotty shares partial aggregates among multiple queries involving different window types.

8.4.1 Multiple Query Example

For example, consider Figure 11. We slice the stream with respect to a minimum session window gap of four. In addition, we process a tumbling window with a length of three. This results in slices of all types. Fixed slices (⬤) mark the endings of tumbling windows and flexible slices (⬤) the endings of sessions. Combined slices (⬤) mark the coincidence of both.

In general, Algorithm 3 remains unchanged for multi query scenarios. However, S corresponds to the *session* which covers $t_e(e)$ including the gap after the session. In contrast to the example in Figure 9, there is no one-to-one mapping between sessions and slices any more. Thus, the subroutines for fusing (Line 24), splitting (Line 14), and extending sessions (Line 20) change. We now discuss these subroutines.

Algorithm 4 Splitting a Session.

Parameter:

    e : Tuple to be inserted.
    $t_e(e)$: Event-time of e.

1: $S \leftarrow$ slice which covers $t_e(e)$;
2: **if** S starts at $t_e(e)$ **then**
3:      *// Slice before S must be fixed.*
4:      change the type of the slice before S to combined;
5:      add e to S;
6: **else** *// S does not start at $t_e(e)$.*
7:      change $t_{end}(S)$ to $t_e(e)$ (excluding $t_e(e)$ from S);
8:      change type of S to $flexible$;
9:      add slice in $[t_e(e), \text{former } t_{end}(S)]$ with former type of S.
10:     add e to the new slice.
11: **end if**

We visualize all cases with examples in Figure 12 and show a formal specification in the form of an algorithm for each action. The discussion presented here refers to session windows as one example of a context aware window type. In general, the Slice Manager is agnostic to the window specifications. Thus, it is irrelevant for the Slice Manager which windows require a slice.

Virtual Session Slices: We introduce the concept of *virtual session slices* to represent sessions including their succeeding gaps. Such slices are not stored, but we keep an index with their start and end times. This allows us to find the session which covers $t_e(e)$ with negligible latency.

8.4.2 Splitting Sessions

Example: In case an out-of-order tuple e forms a new session on its own, we split a past session (Line 14 in Algorithm 3). We show the split logic in Figure 12a. We always split a session at $t_e(e)$. For e_1 and e_2, the event-times do not coincide with the start of an existing slice. Thus, we split the slice covering $t_e(e)$. The slice before $t_e(e)$ is flexible. The slice after $t_e(e)$ inherits the slice type of the slice we split ($\bullet\!\!\Rightarrow\!\bullet\bullet$; $\bullet\!\!\Rightarrow\!\bullet\bullet$). For e_3, $t_e(e_3)$ already matches the start of a slice. Thus we do not need to split a slice. However the slice before e_3 becomes the end of a session and its type changes to combined ($\bullet\!\!\Rightarrow\!\bullet$).

Algorithm: We show the logic for splitting sessions in Algorithm 4. The algorithm receives the out-of-order tuple e and the event-time $t_e(e)$ of that tuple as parameters.

First, we find the slice which covers $t_e(e)$ and name it S (Line 1). If S starts at $t_e(e)$, there is a slice separation at $t_e(e)$ already and we do not need to split slices. However, we need to split the session by changing the slice types. The slice before S must be a fixed slice because it is in the middle of a session where we cannot have combined or flexible slices. When we split the session, the slice before S becomes the last slice of a session. Thus, we change the type of the slice before S to combined (Line 4) and add e to S (Line 5).

If S does not start at $t_e(e)$, we need to split S at $t_e(e)$. Therefore, we first shrink S by changing $t_{end}(S)$ to $t_e(e)$ (Line 7). Then, we change the type of S to flexible, because S marks the end of a session now (Line 8). Finally, we add a new slice in the gap between $t_e(e)$ and the former end of S (Line 9) and add e to this new slice (Line 10). Thereby, the new slice inherits the former type of S.

Algorithm 5 Fusing Sessions.

Parameter:
    e : Tuple to be inserted.
    $t_e(e)$: Event-time of e.
    S : The session which is fused with its succeeding session.

1: $S_L \leftarrow$ last slice in S;
2: $S_F \leftarrow$ first slice in the session after S;
3: **if** S_L is a combined slice **then**
4:     change type of S_L to $fixed$;
5: **else** // S_L *is a flexible slice.* $\Rightarrow$ *remove flexible separator*
6:     fuse S_L and S_F keeping the type of S_F;
7: **end if**
8: add e to the slice covering $t_e(e)$;


8.4.3 Fusing Sessions

Example: When an out-of-order tuple fuses two sessions (Line 24 in Algorithm 3), we check the slice type of the last slice in the earlier session. The last slice of a session must be combined or flexible. If the slice is combined, we cannot fuse it with its succeeding slice because it also marks the ending of a sliding or tumbling window. Thus, we change the slice type from combined to fixed which removes the marker for the session end (1. fuse in Figure 12b; (c)⇒(x)). If the slice is flexible, we fuse it with its successor. The fused slice inherits its type from the later slice (2. fuse in Figure 12b; (flex)(x)⇒(x)).

Algorithm: We formalize the logic for fusing sessions in Algorithm 5. The algorithm receives the out-of-order tuple e, the event-time $t_e(e)$ of that tuple, and the session S (which is fused with its succeeding session) as parameters.

As first step, we find the last slice in the session S and name it S_L (Line 1). Then, we find the first slice in the session after S and name it S_F (Line 2). In order to fuse S with its succeeding session, we fuse S_L with S_F. If S_L is a combined slice, we cannot remove the separation between S_L and S_F because S_L marks the start or end of a tumbling or sliding window. In this case, we fuse sessions by changing the slice type from combined to fixed (Line 4).

If S_L is not combined, it must be flexible. In this case, we fuse S_L with S_F (i.e., we merge them to one partial aggregate). Thereby, we keep the slice type of S_F (Line 6). Finally, we add e to the slice which covers $t_e(e)$ (Line 8). Note that e does not necessarily belong to S_L or S_F. $t_e(e)$ possibly falls in another slice preceding S_L or succeeding S_F.

8.4.4 Extending Sessions

Example: When we extend a session at its start (Line 20 in Algorithm 3), we check if the slice which covers $t_e(e)$ is the last slice of a session. If so, we check if the slice is combined or flexible. If it is flexible, we can change its end time to $t_e(e)$ (excluding) without changing the slice type (e_1 in Figure 12c). If the slice is combined, we cannot move its end and, thus, split it at $t_e(e)$ in a flexible and a fixed slice (e_2 in Figure 12c; (c)⇒(flex)(x)).

In Figure 12d and 12e, we consider the case were we extend a session at its start, but $t_e(e)$ is not within the last slice of a session. In this case, we extend the session in two steps. First, we remove the former end of the session before the one we extend. Second, we mark the new end at $t_e(e)$ (exclusive). In order to remove the former end of the session which covers $t_e(e)$, we require the last slice of that session. If the last slice is flexible, we fuse it with its succeeding slice (Figure 12d). Thereby, the fused slice inherits the type of the later slice ((flex)(x)⇒(x)). If the last slice is combined,

---

Algorithm 6 Extending a Session at its Start.

Parameter:

    e : Tuple to be inserted.

    $t_e(e)$: Event-time of e.

    S : The session which covers $t_e(e)$.

1: $S_e \leftarrow$ slice which covers $t_e(e)$;
2: $S_L \leftarrow$ last slice in S;
3: *// Simple case: No fixed slices are involved.*
4: **if** $S_e == S_L$ **and** S_e is *flexible* **then**
5:      change $t_{end}(S_e)$ to $t_e(e)$ (exclusive);
6:      add e to the slice succeeding S_e;
7: **else if** $S_e == S_L$ **then** *// S_e is combined.*
8:      change $t_{end}(S_e)$ to $t_e(e)$ (exclusive);
9:      add *fixed* slice in $[t_e(e),$former $t_{end}(S_e)]$;
10:     add e to the new slice;
11: **else** *// If S_e and S_L are not the same slice, S_e is fixed.*
12:     *// Remove former end of S.*
13:     **if** S_L is *flexible* **then**
14:        fuse S_L with succeeding slice (keep type of successor);
15:     **else** *// S_L is combined.*
16:        change type of S_L to *fixed*;
17:     **end if**
18:     *// Add new end of S.*
19:     **if** S_e starts at $t_e(e)$ **then** *// change slice type.*
20:        change type of the slice before S_e to *combined*;
21:        add e to S_e;
22:     **else** *// split slice.*
23:        change $t_{end}(S_e)$ to $t_e(e)$ (exclusive);
24:        change type of S_e to *flexible*;
25:        add slice (former type of S_e) in $[t_e(e),$former $t_{end}(S_e)]$;
26:        add e to the new slice;
27:     **end if**
28: **end if**

---

we cannot fuse it and just change its type to fixed (Figure 12e; ⬤⇒⬤). In order to set the new end of the session, we require the slice which covers $t_e(e)$. If this slice starts at $t_e(e)$, we mark the end of the session by setting the type of the preceding slice to combined (⬤⇒⬤). Otherwise, we split the slice which covers $t_e(e)$. The slice before the split is flexible and the one after the split inherits the type of the slice we split (Figure 12d; ⬤⇒⬤⬤).

Algorithm: We show the logic for extending a session in Algorithm 6. The algorithm receives the out-of-order tuple e, the event-time $t_e(e)$ of that tuple, and the session S which covers $t_e(e)$ as parameters. Note that S corresponds to the virtual session slice covering $t_e(e)$ before adding e. Thus, S is the session before the one we extend.

As first step, we find the slice in the session S which covers $t_e(e)$ and name it S_e (Line 1). Then, we find the last slice in S and name it S_L (Line 2).

We check if S_e and S_L are the same slice. If so, we act depending on the type of S_e (i.e., the type of S_L). Since we look at the last slice of a session, the slice can be flexible or combined. If it is flexible, we move the slice separation to extend the session after S at its start. Therefore, we change $t_{end}(S_e)$ to $t_e(e)$ (exclusive) in Line 5. This automatically shifts the start of the session after S to

$t_e(e)$ (inclusive). After extending the session, we add e in Line 6. If S_e is combined, we cannot shift the slice separation. Thus, we need to split S_e at $t_e(e)$. Therefore, we change $t_{end}(S_e)$ to $t_e(e)$ in Line 8. Then, we add a new fixed slice between $t_e(e)$ and the former end of S_e in Line 9. Finally, we add e to the new slice in Line 10.

If S_e and S_L are not the same slice, S_e must be a fixed slice that ends within the gap between two sessions. In this case, we extend the session after S in two steps: First, we remove the former end of S. Then, we add the new earlier end of S and, thereby, extend the session after S.

In order to remove the former end of S, we either fuse S_L with its succeeding slice in Line 14 (if S_L is flexible) or change the type of S_L to fixed in Line 16 (if S_L is combined).

When we add the new end of S at $t_e(e)$, we first check if $t_e(e)$ coincides with a slice separation already. If so, we do not need to split slices. We just mark the end of the session S by changing the type of the slice that ends at $t_e(e)$ from fixed to combined (Line 20) and add e to S_e (Line 21).

If no slice starts at $t_e(e)$, we split S_e at $t_e(e)$. Therefore, we change the end of S_e to $t_e(e)$ (exclusive) in Line 23 and add a new slice between $t_e(e)$ (inclusive) and the former end of S_e (Line 25). Finally, we add e to the new slice (Line 26).

9 Programming Examples

In this section, we show several examples which illustrate how one can use Scotty in different stream processing systems and how one can extend Scotty with new aggregation functions and window types.

9.1 Scotty in Different Stream Processing Systems

Based on the General Stream Slicing approach, we provide Scotty as a general purpose window operator implemented in Java. Scotty is independent of a specific stream processing system. In principle, every stream processing system can embed Scotty as an operator, which supports stateful user-defined processing functions in Java. Furthermore, it is important to note that the correctness guarantees of Scotty, depend on the deliver guarantees of the underlying system. Thus, exactly-once delivery semantics are required to guarantee correct processing results in a distributed stream processing system. Such delivery guarantees are regularly provided by stream processing systems: For example, Flink implements a solution based on asynchronous checkpoints [14, 15]. In case of failures, the operator state will be set back to a recent complete checkpoint and the stream will be replayed starting from that checkpoint. As a result, each tuple is reflected exactly-once in the operator state which yields correct results. Kafka Streams implements a tuple tracking based on change capture messages and special Kafka topics called *changelog topics* and *offset topics* [71]. In case of failures, operator state can be recovered from the messages stored under these topics. Storm implements a message tracking based on acknowledgments and allows for replaying a stream through its reliability API in case of failures [61].

For event-time processing, Scotty can either utilize a system provided timestamp and watermark or derive the event time from a tuple field. In both cases, the timestamp becomes a field of the tuple based on which tuples are assigned to slices and windows by Scotty. In this paper, we limit the discussion to Scotty to allow for discussing algorithms and implementations in detail. However, we provide pointers to related works in Section 11, which address the issue of distributed setups where no global (synchronized) timestamp service can be relied on.

Listing 1 shows the interface of the Scotty window operator, which defines two functions. `ProcessElement()` consumes a new record from the data stream with a corresponding timestamp (either, event time, or processing time). For each invocation, Scotty assigns the record to a slice, adapts the slices if necessary, and updates its corresponding partial aggregate in the aggregation store. `ProcessWatermark()` consumes a watermark timestamp and triggers all windows that

Listing 1. Scotty Window Operator interface

```
1   public class ScottyWindowOperator<InputType>{
2     //Processes a new record with a specific timestamp.
3     public void processElement(InputType element, long ts) {...}
4     //Processes a watermark and provide results for all windows that are triggered since the last watermark.
5     public List<AggregateWindow> processWatermark(long watermarkTs) {...}
6   }
```

Listing 2. Scotty Embedding in Apache Beam

```
1    PCollection<KV<Integer, Integer>> stream = ...
2
3    KeyedScottyWindowOperator<Integer, Integer> scottyWindowDoFn =
4          new KeyedScottyWindowOperator<Integer, Integer>(0, new Sum());
5
6    //Adding three window definition for Tumbling, Sliding and Senssion windows.
7    scottyWindowDoFn.addWindow(new TumblingWindow(WindowMeasure.Time, 5000));
8    scottyWindowDoFn.addWindow(new SlidingWindow(WindowMeasure.Time, 2000, 1000));
9    scottyWindowDoFn.addWindow(new SessionWindow(WindowMeasure.Time, 2000));
10   //Apply Scotty Windowing
11   PCollection<WindowResults> windowAggregates = stream.apply(ParDo.of(scottyWindowDoFn));
```

ended between the last received watermark timestamp and the current watermark timestamp. To support out-of-order events, Scotty delays the termination of windows by a fixed allowed lateness. In the future, we plan to integrate window refinements as defined by the Dataflow Model [1] and adaptive watermark delays [7] to minimize the windowing latency.

To integrate Scotty with a stream processing system, Scotty has to become an (user-defined) operator in the respective system which can then be used just like any other operator. To this end, one has to implement a connector class. Such a class is usually a wrapper class instantiating Scotty and implementing (or extending) the operator class of the stream processing system. Such a class is usually implemented as an interface or abstract class in Java. The complexity of implementing a connector depends on the respective stream processing system and its requirements for implementing a custom operator. First, one needs to implement the logic for passing input tuples to Scotty and results back to the system. Second, it is required to connect the state of Scotty to the state backup of the respective systems, to be able to recover from node failures. To ease the connection to different state backends, we made the entire Scotty window operator serializable. A careful study and implementation of state backups for individual systems can significantly reduce the overhead caused by state backups upon checkpoints and caused by state recovery upon failures.

The Scotty open-source project provides connectors for Apache Flink [16], Apache Storm [61], Apache Beam [1], Apache Samza [48], Apache Kafka Streams [40], and Apache Spark Continous Processing [60, 75]. These connectors implement Flink's `KeyedProcessFunction`, Storm's `BaseBasicBolt`, Beam's `DoFn`, Samza's `StreamTask`, Kafka's `Processor`, and Spark's `FlatMapFunction` respectively. Using these connectors, Scotty can directly replace the native window operator of these stream processing systems. Listing 2 presents an Apache Beam stream processing pipeline using the Scotty operator. This example, demonstrates the definition of a window aggregation across multiple different concurrent windows. In line 3 to 9 we create a window operator, which performs a sum aggregation on three window definitions (i.e., a tumbling-, a sliding-, and a session-Window). Finally, the Scotty window operator is directly embedded into the Apache Beam pipeline as any native operator. All connectors expose the same Scotty API through their `addWindow` methods. To use Scotty, users first configure the Scotty operator using the same commands independent of the streaming system. Then, users add the Scotty operator to their processing pipeline just like any other operator of the respective system. We provide a demo for each system in the respective connector directory in the Scotty open source project.

Listing 3. Interface for User-defined Window Aggregations

```
1   public interface AggregateFunction <InputType , PartialAggregateType , FinalAggregateType > {
2     // Transforms a tuple to a partial aggregate .
3     PartialAggregateType lift ( InputType inputTuple );
4     // Computes the combined aggregate from partial aggregates .
5     PartialAggregateType combine ( PartialAggregateType pAgg1 , PartialAggregateType pAgg2 );
6     // Transforms a partial aggregate to a final aggregate .
7     FinalAggregateType lower ( PartialAggregateType pAgg );
8   }
```

9.2 Extending Scotty with custom Window Aggregations and Types

The Scotty window operator provides several extension points, which allow users to define own aggregation functions and window types. In principle, these extension points follow the description in Section 5.4. In the following we describe how users can use these extension points programmatically.

Window Aggregations: For the definition of a user-defined window aggregation function, Scotty provides the interface shown in Listing 3. Each aggregation function definition consists of a `lift`, `combine`, and `lower` function. Additionally, Scotty provides two interfaces to declare if an aggregation function is commutative or invertible. This allows for performing the different slicing decisions for arbitrary aggregation functions as defined in Section 5. Depending on the aggregation function, the `PartialAggregationType` stores different data. For a sum aggregation, it only stores a single number as the current partial window sum. In contrast, for a holistic aggregation, we must store all records assigned to a slice. Depending on the concrete implementation, this can be a simple buffer of all records, a tree structure to improve efficiency, or a compressed format to save memory.

Window Types: For the definition of user-defined window types, Scotty provides three interfaces, which correspond to the three abstract window types in Scotty (i.e., context-free, forward context-free, and forward context-aware). For forward context-free and forward context-aware windows, the user has to implement a window context. The window context can maintain arbitrary information about the window boundaries and content. For example, a session window maintains a list of all currently active windows. For each processed record, Scotty updates this window context and checks if it should fuse two windows, or it has to insert a new window. Consequently, it adapts the underlying slices according to the rules described in Section 6. Finally, Scotty uses the window context to trigger the windows if the watermark exceeds the window end. Beside the common window types (i.e., tumbling-window, sliding-window, and session-window), Scotty provides implementations for further window types, such as Fixed-band Window [51], Punctuation-based Windows [68], and Slide-By-Tuple Windows [44].

10 Evaluation

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

10.1 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 [19, 69]. We determine latencies with the JMH benchmarking suite [49]. JMH provides precise latency measurements on JVM-based systems. We use the *ObjectSizeCalculator* of Nashorn to determine memory footprints [50].

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 [59]. For the buckets technique, we use the implementation of Apache Flink [16]. For tuple buffers, we use an implementation based on a ring buffer array. We also include Pairs [42] and Cutty [17] as specialized slicing techniques where possible.

Data. We replay real-world sensor data from a football match [47] and from manufacturing machines [36]. 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 [26]. We add 5 gaps per minute to separate sessions. This is representative for the ball possession moving from one player to another[2]. 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 that is built for the football data we use [67]. If not indicated differently, we use the sum aggregation in Sections 10.2 and Section 10.3. In Section 10.4, we use the M4 aggregation technique [37] 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 10.3.2 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 four parts. First, we compare stream slicing and alternative approaches with respect to their throughput, latency, and memory footprint (Section 10.2). Second, we study the impact of each workload characteristic introduced in Section 4 (Section 10.3). Third, we integrate general slicing in Apache Flink and show the performance gain for a concrete application (Section 10.4). Fourth, we compare Scotty with the built-in solutions for window aggregation on different systems (Section 10.5). Sections 10.2 and 10.3 focus on the performance per operator instance, Section 10.4 studies the parallelization, and Section 10.5 studies different systems.

10.2 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 10.2.1. Our goal is to understand the performance of stream slicing compared to alternative techniques, including specialized slicing techniques. In Section 10.2.2, we evaluate how the throughput changes in the

---

[2]The DEBS 2013 Grand Challenge defines *ball possession* as follows: *"A player (and thereby his respective team) can obtain the ball whenever the ball is in his proximity and he hits it. A ball is in proximity of the player when it is less than one meter away from him. The distance of one meter applies to the distance between the sensor within the ball and any of the two sensors in the player's shin guards. A ball is hit whenever its acceleration or velocity peaks. A ball will stay in the possession of a given player until another player hits it, the ball leaves the field, or the game is stopped. Specifically, a ball may leave the player's proximity and will still remain in his possession."* [47]
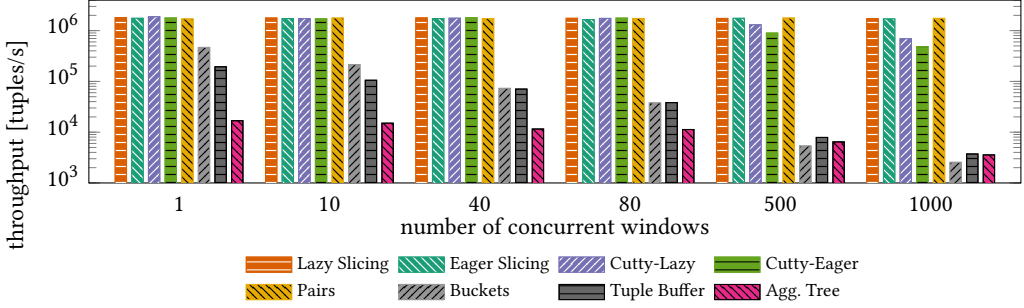
Fig. 13. In-order Processing with Context Free Windows.

presence of out-of-order tuples and context-aware windows. In Section 10.2.3, we evaluate the
memory footprint and in Section 10.2.4 the latency of different techniques.

10.2.1 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
that facilitate plotting line charts at different zoom levels (Application of Section 10.3). We chose
Pairs [42] and Cutty [17] 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 13. All three slicing techniques process millions of tuples
per second and scale to large numbers of concurrent windows.

Buckets achieve orders of magnitude less throughput than slicing techniques and do 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.

10.2.2 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
($l_g = 1$sec.) 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 14. Slicing techniques achieve an order of magnitude higher
throughput than alternative techniques that 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 that cause updates in the aggregate tree. Buckets show the same performance
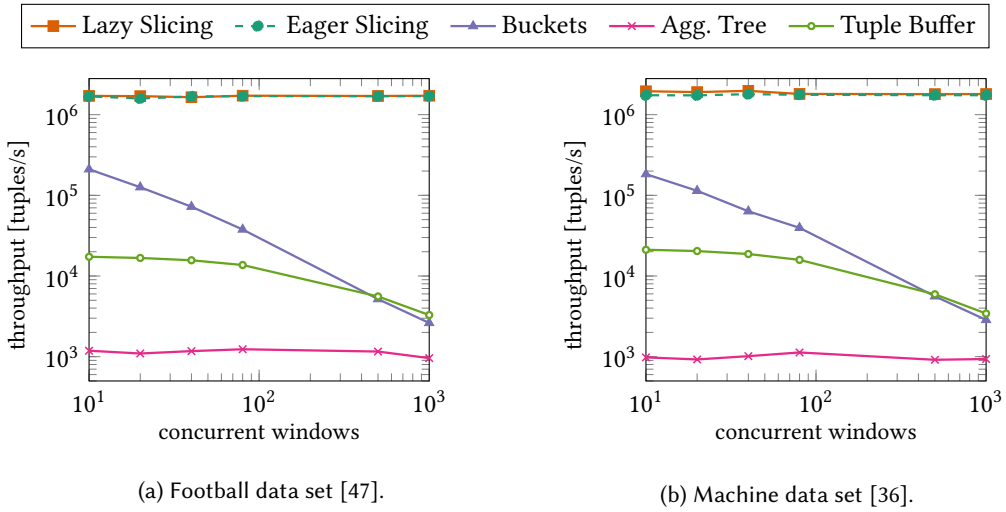
Fig. 14. Increasing the number of concurrent windows including 20% out-of-order tuples and session windows.

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 avoids 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.

10.2.3 Memory Consumption We now study the memory consumption of different techniques with four plots: In Figures 15a and 15c, 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 15b and 15d, 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 15a and 15b show the memory consumption for time-based windows, which do not require us 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.
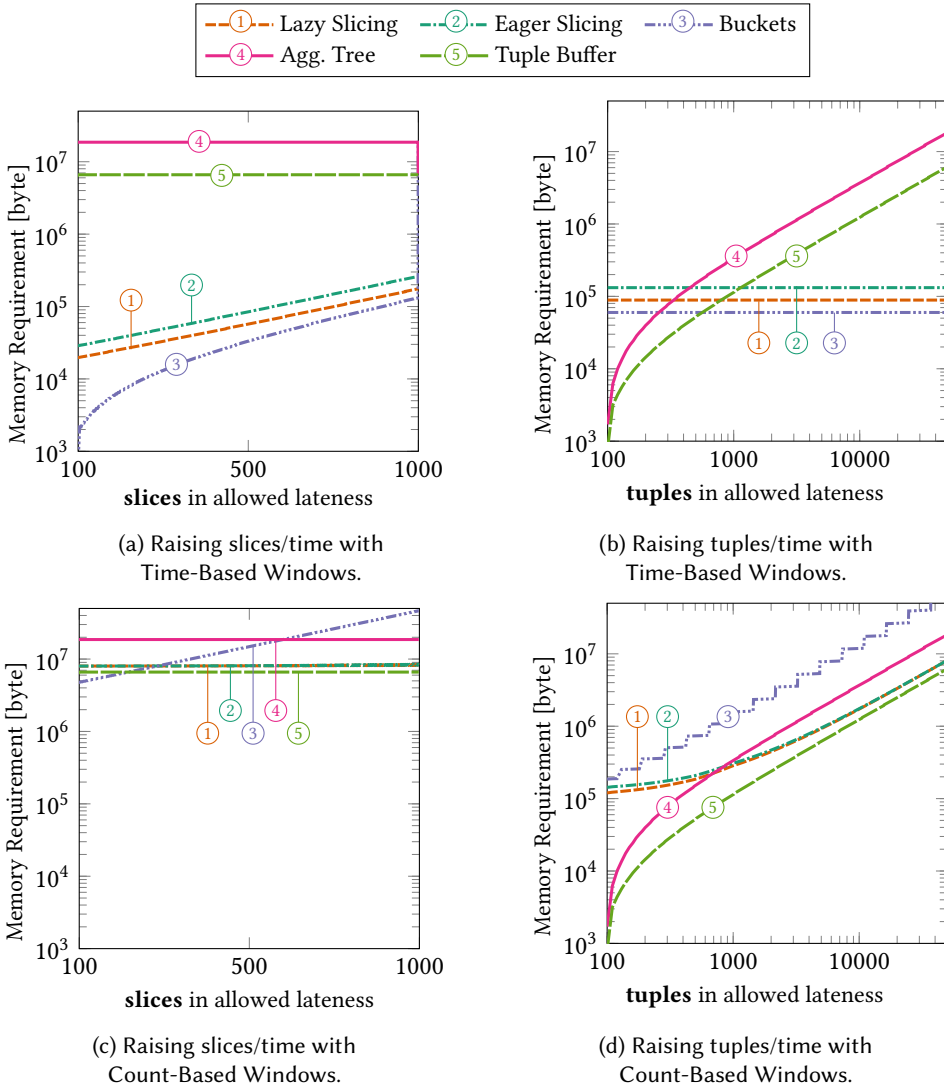
Fig. 15. Memory Experiments with Out-of-order Streams.

Results for Count-Based Windows. Figures 15c and 15d show the memory consumption for count-based windows, which require individual tuples to be stored. The experiment setup is the same as in Figures 15a and 15b. 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 15d). 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 [70]. 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 15c). Each bucket stores all tuples it contains which leads to duplicated tuples for overlapping buckets.

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

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

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

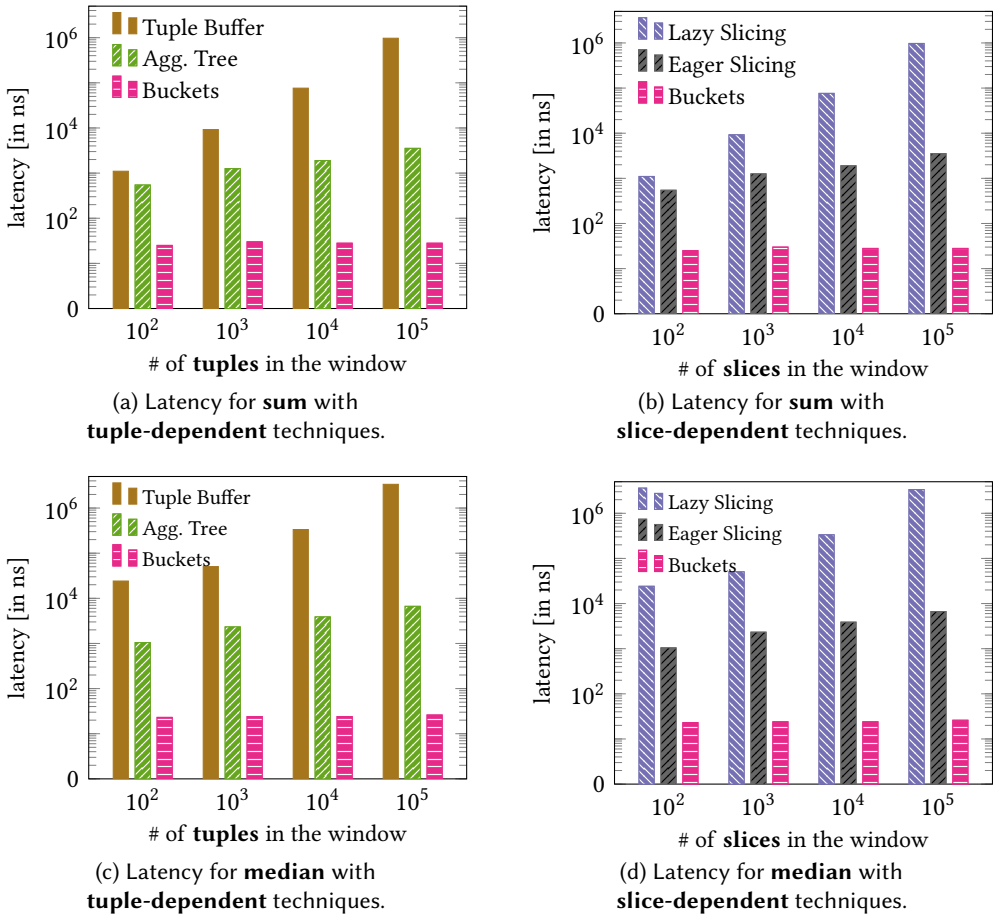(d) Latency for **median** with
slice-dependent techniques.

Fig. 16. Output Latency of Aggregate Stores.

Summary. When we can drop individual tuples and store partial aggregates only (Figure 15a
and 15b), 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 15c and 15d),
storing tuples dominates the memory consumption.

10.2.4 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 16,
we show the latency for different situations.

Distributive and Algebraic Aggregation. For the sum aggregation (Figure 16a), 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
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.

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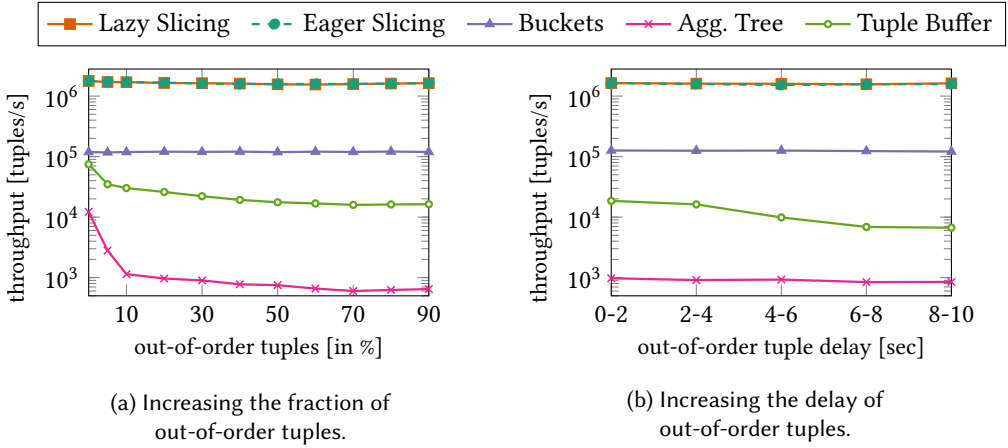
(b) Increasing the delay of
out-of-order tuples.

Fig. 17. Impact of Stream Order on the Throughput.

Holistic Aggregation. The latencies for the holistic median aggregation (Figure 16c) 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.

10.3 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.

10.3.1 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 17). We use the same setup as for the throughput experiments in Section 10.2.2 with 20 concurrent windows.

Out-of-order Performance. In Figure 17a, 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 that require updates. Aggregate Trees on tuples have a much larger number of tree levels because they store tuples instead of slices as leaf nodes.
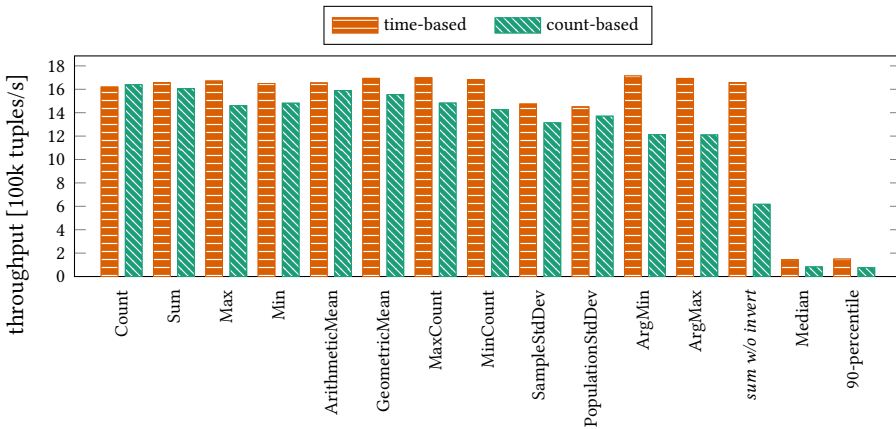
Fig. 18. Impact of Aggregation Types on Throughput.

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 re-computation of aggregates. Eager Slicing seldom faces this issue (see Section 10.2.2).

Delay Robustness. In Figure 17b, 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.

10.3.2 *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 18. We differentiate time-based and count-based windows to show the impact of invertibility. We implement the same aggregation functions as Tangwongsang et al. [59]. 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 that 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.

(a) Football data set [47].   (b) Machine data set [36].
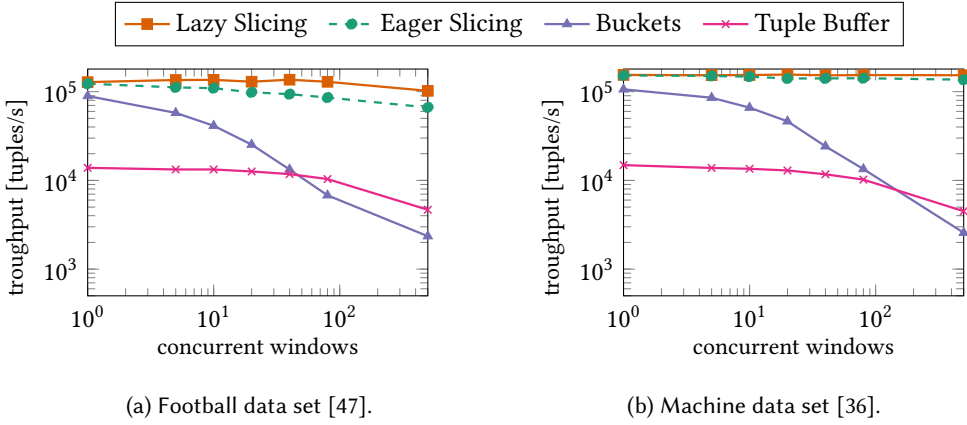
Fig. 19. Throughput for Median Aggregation.

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 18). 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 that always requires a recomputation when removing tuples.

Impact of Holistic Aggregations. In Figure 18, we observe that holistic aggregations have a much lower throughput than algebraic and distributive aggregations. In Figure 19, 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.

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.
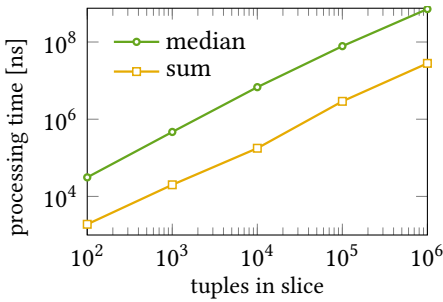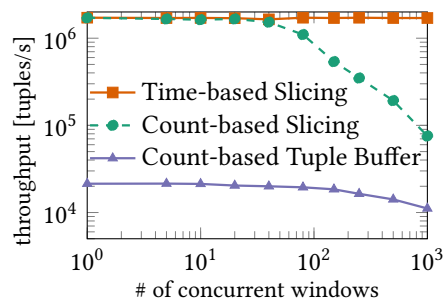
Fig. 20. Time for Recomputing Aggregates.

Fig. 21. Impact of Different Window Measures.

10.3.3 *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 20). 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.

10.3.4 *Impact of Window Measures* We compare different window measures in Figure 21. 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 10.2.2. The throughput for arbitrary advancing measures is the same as for time-based measures because they are processed identically [17].

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.
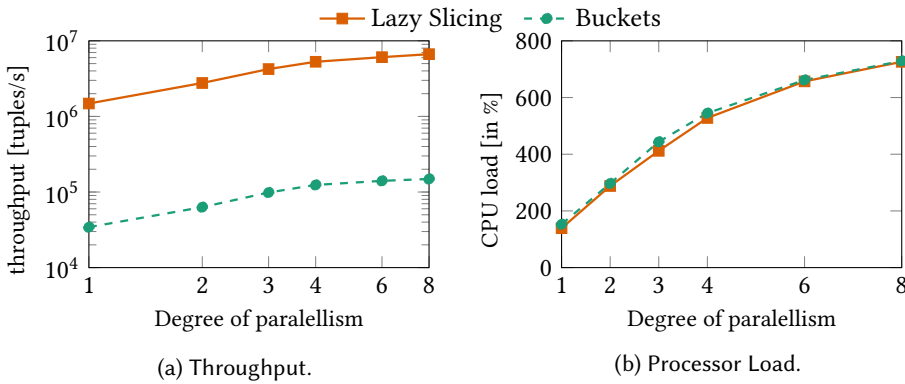
(a) Throughput.

(b) Processor Load.

Fig. 22. Parallelizing the workload of a live-visualization dashboard (80 concurrent win. per operator instance).

10.4 Parallel Stream Slicing

In this experiment, we study stream slicing on the example of our dashboard application [67] which uses the M4 aggregation [37]. 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 22, 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 22a). 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 22b). Slicing achieves an order of magnitude higher throughput than buckets, because it prevents assigning tuples to multiple buckets (cf. Section 10.2.1). 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.

10.5 Scotty in Different Stream Processing Systems

We evaluate the throughput of window aggregation on different stream processing systems in Figure 23. Since Apache Beam provides an API only, we evaluate the Beam API on Flink and Samza. We keep data, queries, and metrics as introduced in Section 10.2.1. We take the throughput of Scotty with one concurrent window as baseline (100% throughput) and compare it to the throughput of the respective built-in techniques. We show how each technique scales when increasing the number of concurrent windows.

We stick to relative numbers because our experiments are conducted with different hardware and software setups that impact the absolute throughput. For example, Samza requires to read data from Apache Kafka as message broker, which requires to also run Apache Zookeeper. The overhead of these two additional systems contributes to a lower throughput of Samza compared to Flink. The Beam API added a heavy overhead, which was analyzed in detail by Hesse et al. [29]. This leads to much lower throughput compared to the native APIs of Flink and Samza. We have a well maintained and tuned setup of Flink which contributes to Flink having the highest throughput in all experiments. In this paper, we focus on comparing techniques for window aggregation rather than streaming systems. Karimov at al. provide a solid comparison of systems, which addresses and carefully analyzes effects like the ones stated above [38].
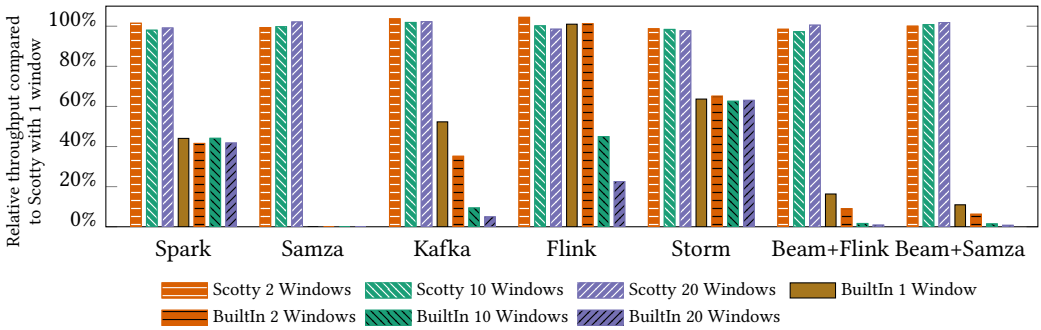
Fig. 23. Throughput of Scotty compared to built-in operators on different systems.

Results. Scotty outperforms the respective built-in techniques on all evaluated systems. The results also confirm Scotty's scalability to large numbers of concurrent windows which we observed in previous experiments. For a single window, only Flink's built-in solution (based on Buckets) achieves similar throughput than Scotty. However, the throughput of Buckets decreases drastically when we add more concurrent windows, which is also reflected in the results for Beam on Flink, Beam on Samza, and Kafka Streams. The built-in techniques of Spark and Storm did not exhibit such scalability issues. We do not show results for the built-in technique of Samza, because it did not produce correct results and our request to the mailing list was not answered.

Summary. Scotty provides higher throughput then built-in operators on all tested systems and scales to large numbers of concurrent windows. The built-in operators of Spark and Storm also scale to large numbers of concurrent windows but provide lower throughput than Scotty.

11 Related Work

In this section, we discuss additional related work that we did not cover in the previous sections. First, we summarize alternative approaches for optimizing window aggregations. Then, we discuss solutions that process streams in batches, before we summarize complementary techniques. Finally, we present follow-up works published after the short versions of this paper [64, 65].

Optimizing Window Aggregations. Our general slicing techniques utilizes features of existing techniques such as on-the-fly slicing [42], incremental aggregation [59], window grouping [27, 28], and user-defined windows [17]. However, general stream slicing offers a unique combination of generality and performance. One can extend other slicing techniques based on this paper to reach similar generality and performance. Existing slicing techniques such as Pairs [42] and Panes [43] are limited to tumbling and sliding windows. Cutty can process user-defined window types, but does not support out-of-order processing [17]. Several publications optimize sliding window aggregations focusing on different aspects such as incremental aggregation [10, 22, 59] or worst-case constant time aggregation [57]. 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 [31]. 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. Our solutions can be integrated in open source streaming systems such as Apache Flink [16], Spark [75], and Storm [61].

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 [8,

39, 74]. For example, D-Streams [74] processes mini-batches of data, which are combined to windows. This requires the *slide step* and *range* of sliding windows to be multiples of the batch size. SABER introduces *window fragments* to decouple *slide* and *range* of sliding windows from the batch size [39]. 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 [8]. 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 [27, 28, 53]. 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 [34]. 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 [41]. 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 [54, 55]. Trill [18] 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 time.

Follow-Up Works. The general stream slicing approach and Scotty have been recognized by several follow-up works. Benson et al. introduced a distributed window aggregation approach using an extended version of the Scotty library [9]. Their solutions extends Scotty with in-network aggregation and pre-aggregation at remote data sources. The NebulaStream platform for data management in the internet-of-things adopted general stream slicing to enable efficient stream processing on small devices [76]. Hirzel et al. proposed FiBA, a highly efficient sliding window aggregation algorithm that optimally handles streams of varying degrees of out-of-orderness [58]. They conclude that this algorithm could serve as a more efficient aggregation store in Scotty to combine the benefits of stream slicing with faster final aggregation. Furthermore, general stream slicing was attributed as a promising approach to further improve the processing efficiency of modern scale-up stream processing system [25, 77, 78].

Stream processing systems regularly receive tuples with event timestamps that have been assigned at the spots where the respective events have been recorded. For example, in a distributed processing setting such as the Internet of Things, timestamps may be assigned by sensor nodes such as smartphones, machines, or connected cars. In such a distributed processing setting, where no global (synchronized) timestamp service can be relied on, no central stream processing system can guarantee correctness, because events with the same timestamp may be recorded at different times due to unsynchronized clocks. We have addressed this issue with the SENSE system for gathering sensor data tuples with guaranteed time coherence [35, 63, 66]. We have further integrated SENSE and Scotty into an end-to-end stream processing pipeline [62].

12   Conclusion

Stream slicing is a technique for streaming window aggregation which provides high throughputs and low latencies with a small memory footprint. 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. We implemented our general stream slicing technique in *Scotty*, an open-source operator that provides efficient and general sliding-window aggregation for stream processing systems under

the Apache 2.0 license. Scotty also allows for implementing user-defined aggregation functions and window types without changing the core operations of the technique.

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. In general, stream slicing is beneficial whenever a slice contains at least a few records (≈ 10). Thus, it is not beneficial for count-based windows with a very small slide step (less than 10 records). We observed that the throughput scales linearly with the number of processing cores and that latency is in the order of microseconds. Techniques based on window buckets provide lower latency, but exhibit drastically lower throughput due to repeated computations for concurrent windows.

References

[1] Tyler Akidau, Robert Bradshaw, et al. 2015. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing. *PVLDB* 8, 12 (2015), 1792–1803.

[2] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, et al. 2014. The Stratosphere platform for big data analytics. *VLDB Journal* 23, 6 (2014), 939–964.

[3] Apache Apex. 2018. Enterprise-grade unified stream and batch processing engine. https://apex.apache.org/.

[4] Apache Beam. 2018. An advanced unified programming model. https://beam.apache.org/ (project website).

[5] Arvind Arasu and Jennifer Widom. 2004. Resource sharing in continuous sliding-window aggregates. *Proceedings of the International Conference on Very Large Data Bases (PVLDB)* (2004), 336–347.

[6] Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, et al. 2018. Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark. In *ACM SIGMOD*. 601–613.

[7] Ahmed Awad, Jonas Traub, and Sherif Sakr. 2019. Adaptive Watermarks: A Concept Drift-based Approach for Predicting Event-Time Progress in Data Streams. In *EDBT*.

[8] Cagri Balkesen and Nesime Tatbul. 2011. Scalable data partitioning techniques for parallel sliding window processing over data streams. In *International Workshop on Data Management for Sensor Networks (DMSN)*.

[9] Lawrence Benson, Philipp M. Grulich, Steffen Zeuch, Volker Markl, and Tilmann Rabl. 2020. Disco: Efficient Distributed Window Aggregation. In *EDBT*.

[10] Pramod Bhatotia, Umut A Acar, Flavio P Junqueira, and Rodrigo Rodrigues. 2014. Slider: Incremental sliding window analytics. In *Proceedings of the International Middleware Conference*. ACM, 61–72.

[11] Brice Bingman. 2018. Poor performance with Sliding Time Windows. In *Flink Jira Issues (issues.apache.org/jira/browse/FLINK-6990)*.

[12] Irina Botan, Roozbeh Derakhshan, Nihal Dindar, Laura Haas, Renée J. Miller, and Nesime Tatbul. 2010. SECRET: A Model for Analysis of the Execution Semantics of Stream Processing Systems. *PVLDB* 3, 1-2 (2010), 232–243.

[13] Paris Carbone. 2018. *Scalable and Reliable Data Stream Processing*. Ph.D. Dissertation. KTH Stockholm.

[14] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. 2017. State management in Apache Flink®: consistent stateful distributed stream processing. *PVLDB* 10, 12 (2017), 1718–1729.

[15] Paris Carbone, Gyula Fóra, Stephan Ewen, Seif Haridi, and Kostas Tzoumas. 2015. Lightweight asynchronous snapshots for distributed dataflows. *arXiv* (2015).

[16] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink: Stream and batch processing in a single engine. *IEEE Data Engineering Bulletin* 38, 4 (2015), 28–38.

[17] Paris Carbone, Jonas Traub, Asterios Katsifodimos, Seif Haridi, and Volker Markl. 2016. Cutty: Aggregate Sharing for User-Defined Windows. In *ACM CIKM*. 1201–1210.

[18] Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, Robert DeLine, Danyel Fisher, John C Platt, James F Terwilliger, and John Wernsing. 2014. Trill: A High-performance Incremental Query Processor for Diverse Analytics. *Proceedings of the International Conference on Very Large Data Bases (PVLDB)* 8, 4 (2014), 401–412.

[19] Sanket Chintapalli, Derek Dagit, Bobby Evans, et al. 2016. Benchmarking streaming computation engines: Storm, Flink and Spark streaming. In *IEEE IPDPSW*. 1789–1792.

[20] Xenofontas Dimitropoulos, Paul Hurley, Andreas Kind, and Marc Ph Stoecklin. 2009. On the 95-percentile billing method. In *Passive and Active Network Measurement (PAM)*. Springer, 207–216.

[21] Buğra Gedik. 2014. Generic windowing support for extensible stream processing systems. *Software: Practice and Experience (SPE)* 44, 9 (2014), 1105–1128.

[22] Thanaa M Ghanem, Moustafa A Hammad, Mohamed F Mokbel, Walid G Aref, and Ahmed K Elmagarmid. 2007. Incremental evaluation of sliding-window queries over data streams. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 19, 1 (2007), 57–72.

[23] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. 1997. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery (DMKDFD)* 1, 1 (1997), 29–53.

[24] Michael Grossniklaus, David Maier, James Miller, Sharmadha Moorthy, and Kristin Tufte. 2016. Frames: data-driven windows. In *Proceedings of the ACM International Conference on Distributed and Event-based Systems (DEBS)*.

[25] Philipp M. Grulich, Sebastian Breß, Jonas Traub, Tilmann Rabl, Janis von Bleichert, Zongxiong Chen, Steffen Zeuch, and Volker Markl. 2020. Grizzly: Efficient Stream Processing Through Adaptive Query Compilation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM New York, NY, USA.

[26] Philipp M. Grulich, Jonas Traub, Sebastian Breß, Asterios Katsifodimos, Tilmann Rabl, and Volker Markl. 2019. Generating Reproducible Out-of-Order Data Streams. In *ACM DEBS*. 256–257.

[27] Shenoda Guirguis, Mohamed A Sharaf, Panos K Chrysanthis, and Alexandros Labrinidis. 2011. Optimized processing of multiple aggregate continuous queries. In *CIKM*. ACM, 1515–1524.

[28] Shenoda Guirguis, Mohamed A Sharaf, Panos K Chrysanthis, and Alexandros Labrinidis. 2012. Three-level processing of multiple aggregate continuous queries. In *IEEE International Conference on Data Engineering (ICDE)*. 929–940.

[29] Guenter Hesse, Christoph Matthies, Kelvin Glass, Johannes Huegle, and Matthias Uflacker. 2019. Quantitative Impact Evaluation of an Abstraction Layer for Data Stream Processing Systems. In *ICDCS*. IEEE, 1381–1392.

[30] Martin Hirzel, Henrique Andrade, Buğra Gedik, Vibhore Kumar, Giuliano Losa, Howard Nasgaard, Robert Soulé, and Kun-Lung Wu. 2009. SPL stream processing language specification. *IBM Research Report* (2009).

[31] Martin Hirzel, Scott Schneider, and Kanat Tangwongsan. 2017. Sliding-Window Aggregation Algorithms: Tutorial. In *Proceedings of the ACM International Conference on Distributed and Event-based Systems (DEBS)*. 11–14.

[32] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. 2014. A Catalog of Stream Processing Optimizations. *Comput. Surveys* 46, 4 (2014), 46.

[33] Kartik Hosanagar, John Chuang, Ramayya Krishnan, and Michael D Smith. 2008. Service adoption and pricing of content delivery network (CDN) services. *Management Science* 54, 9 (2008), 1579–1593.

[34] Ryan Huebsch, Minos Garofalakis, Joseph M Hellerstein, and Ion Stoica. 2007. Sharing aggregate computation for distributed queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 485–496.

[35] Julius Hülsmann, Jonas Traub, and Volker Markl. 2020. Demand-based sensor data gathering with multi-query optimization. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2801–2804.

[36] Zbigniew Jerzak, Thomas Heinze, Matthias Fehr, Daniel Gröber, Raik Hartung, and Nenad Stojanovic. 2012. The DEBS 2012 grand challenge. In *DEBS*. 393–398.

[37] Uwe Jugel, Zbigniew Jerzak, Gregor Hackenbroich, and Volker Markl. 2014. M4: a visualization-oriented time series data aggregation. *Proceedings of the International Conference on Very Large Data Bases (PVLDB)* 7, 10 (2014), 797–808.

[38] Jeyhun Karimov, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen, and Volker Markl. 2018. Benchmarking Distributed Stream Processing Engines. In *IEEE ICDE*.

[39] Alexandros Koliousis, Matthias Weidlich, Raul Castro Fernandez, Alexander L Wolf, Paolo Costa, and Peter Pietzuch. 2016. SABER: Window-based hybrid stream processing for heterogeneous architectures. In *ACM SIGMOD*. 555–569.

[40] Jay Kreps. 2016. Introducing Kafka Streams: Stream Processing Made Simple. *Confluent Blog* (2016). https://www.confluent.io/blog/introducing-kafka-streams-stream-processing-made-simple/.

[41] Sailesh Krishnamurthy, Michael J Franklin, Jeffrey Davis, Daniel Farina, Pasha Golovko, Alan Li, and Neil Thombre. 2010. Continuous analytics over discontinuous streams. In *ACM SIGMOD*. 1081–1092.

[42] Sailesh Krishnamurthy, Chung Wu, and Michael Franklin. 2006. On-the-fly sharing for streamed aggregation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 623–634.

[43] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A Tucker. 2005. No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. *SIGMOD Record* 34, 1 (2005), 39–44.

[44] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A Tucker. 2005. Semantics and evaluation techniques for window aggregates in data streams. In *ACM SIGMOD*. 311–322.

[45] Jin Li, Kristin Tufte, David Maier, and Vassilis Papadimos. 2008. AdaptWID: An adaptive, memory-efficient window aggregation implementation. *IEEE Internet Computing* 12, 6 (2008), 22–29.

[46] Jin Li, Kristin Tufte, Vladislav Shkapenyuk, Vassilis Papadimos, Theodore Johnson, and David Maier. 2008. Out-of-order processing: A new architecture for high-performance stream systems. *PVLDB* 1, 1 (2008), 274–288.

[47] Christopher Mutschler, Holger Ziekow, and Zbigniew Jerzak. 2013. The DEBS 2013 grand challenge. In *Proceedings of the ACM International Conference on Distributed and Event-based Systems (DEBS)*. 289–294.

[48] Shadi A Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H Campbell. 2017. Samza: stateful scalable stream processing at LinkedIn. *PVLDB* 10, 12 (2017), 1634–1645.

[49] OpenJDK. 2018. JMH Benchmarking Suite Project Website. http://openjdk.java.net/projects/code-tools/jmh/.

[50] OpenJDK. 2018. Nashorn Project, ObjectSizeCalculator. http://openjdk.java.net/projects/nashorn/.

[51] Kostas Patroumpas et al. 2006. Window specification over data streams. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*.

[52] David Salomon. 2007. *Variable-length Codes for Data Compression.* Springer.

[53] Anatoli U Shein, Panos K Chrysanthis, and Alexandros Labrinidis. 2015. F1: Accelerating the Optimization of Aggregate Continuous Queries. In *CIKM*. 1151–1160.

[54] Anatoli U Shein, Panos K Chrysanthis, and Alexandros Labrinidis. 2017. Flatfit: Accelerated incremental sliding-window aggregation for real-time analytics. In *International Conference on Scientific and Statistical Database Management.*

[55] Anatoli U Shein, Panos K Chrysanthis, and Alexandros Labrinidis. 2018. SlickDeque: High Throughput and Low Latency Incremental Sliding-Window Aggregation. In *EDBT*.

[56] Leo Syinchwun. 2016. Lightweight Event Time Window. In *Flink Jira Issues (issues.apache.org/jira/browse/FLINK-5387)*.

[57] Kanat Tangwongsan, Martin Hirzel, and Scott Schneider. 2017. Low-Latency Sliding-Window Aggregation in Worst-Case Constant Time. In *DEBS*. 66–77.

[58] Kanat Tangwongsan, Martin Hirzel, and Scott Schneider. 2019. Optimal and general out-of-order sliding-window aggregation. *Proceedings of the VLDB Endowment* 12, 10 (2019), 1167–1180.

[59] Kanat Tangwongsan, Martin Hirzel, Scott Schneider, and Kun-Lung Wu. 2015. General incremental sliding-window aggregation. *Proceedings of the International Conference on Very Large Data Bases (PVLDB)* 8, 7 (2015), 702–713.

[60] Joseph Torres, Michael Armbrust, Tathagata Das, and Shixiong Zhu. 2018. Introducing Low-latency Continuous Processing Mode in Structured Streaming in Apache Spark 2.3. *Databricks Blog* (2018).

[61] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. 2014. Storm@twitter. In *ACM SIGMOD*. 147–156.

[62] Jonas Traub. 2019. Demand-based data stream gathering, processing, and transmission. *TU Berlin, PhD Thesis* (2019).

[63] Jonas Traub, Sebastian Breß, Tilmann Rabl, Asterios Katsifodimos, and Volker Markl. 2017. Optimized On-Demand Data Streaming from Sensor Nodes. *Proceedings of the 2017 Symposium on Cloud Computing (SoCC)* (2017), 586–597.

[64] Jonas Traub, Philipp M. Grulich, Alejandro Rodriguez Cuellar, Sebastian Breß, Asterios Katsifodimos, Tilmann Rabl, and Volker Markl. 2018. Scotty: Efficient Window Aggregation for out-of-order Stream Processing. In *IEEE ICDE*.

[65] Jonas Traub, Philipp M. Grulich, Alejandro Rodriguez Cuellar, Sebastian Breß, Asterios Katsifodimos, Tilmann Rabl, and Volker Markl. 2019. Efficient Window Aggregation with General Stream Slicing. In *EDBT*.

[66] Jonas Traub, Julius Hülsmann, Sebastian Breß, Tilmann Rabl, and Volker Markl. 2019. SENSE: Scalable Data Acquisition from Distributed Sensors with Guaranteed Time Coherence. (2019). https://arxiv.org/abs/1912.04648.

[67] Jonas Traub, Nikolaas Steenbergen, Philipp M. Grulich, Tilmann Rabl, and Volker Markl. 2017. I2: Interactive Real-Time Visualization for Streaming Data. In *EDBT*. 526–529.

[68] Peter A. Tucker, David Maier, Tim Sheard, and Leonidas Fegaras. 2003. Exploiting punctuation semantics in continuous data streams. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 15, 3 (2003), 555–568.

[69] Kostas Tzoumas et al. 2015. High-throughput, low-latency, and exactly-once stream processing with Apache Flink. data-artisans.com/blog/high-throughput-low-latency-and-exactly-once-stream-processing-with-apache-flink.

[70] Mikhail Vorontsov. 2013. Memory consumption of popular Java data types - part 2. *Java Performance Tuning Guide* (2013). http://java-performance.info/memory-consumption-of-java-data-types-2/.

[71] Guozhang Wang. 2017. Enabling Exactly-Once in Kafka Streams. *Confluent Blog* (2017). https://www.confluent.io/blog/enabling-exactly-once-kafka-streams/.

[72] Jark Wu. 2017. Improve performance of Sliding Time Window with pane optimization. In *Flink Jira Issues (issues.apache.org/jira/browse/FLINK-7001)*.

[73] Yuan Yu, Pradeep Kumar Gunda, and Michael Isard. 2009. Distributed aggregation for data-parallel computing: interfaces and implementations. In *ACM SIGOPS*. 247–260.

[74] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion Stoica. 2012. Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters. *USENIX conf. on Hot Topics in Cloud Ccomputing* 12 (2012).

[75] Matei Zaharia, Reynold S Xin, Patrick Wendell, et al. 2016. Apache Spark: A unified engine for big data processing. *Commun. ACM* 59, 11 (2016), 56–65.

[76] Steffen Zeuch, Ankit Chaudhary, Bonaventura Del Monte, et al. 2020. The NebulaStream Platform: Data and Application Management for the Internet of Things. In *Conference on Innovative Data Systems Research (CIDR)*.

[77] Steffen Zeuch, Bonaventura Del Monte, Jeyhun Karimov, Clemens Lutz, Manuel Renz, Jonas Traub, Sebastian Breß, Tilmann Rabl, and Volker Markl. 2019. Analyzing Efficient Stream Processing on Modern Hardware. In *PVLDB*.

[78] Shuhao Zhang, Feng Zhang, Yingjun Wu, Bingsheng He, and Paul Johns. 2020. Hardware-Conscious Stream Processing: A Survey. *ACM SIGMOD Record* 48, 4 (2020), 18–29.