# Slicing techniques for temporal aggregation in spanning event streams

Aurélie Suzanne, Guillaume Raschia, José Martinez, Damien Tassetti



## To cite this version:

Aurélie Suzanne, Guillaume Raschia, José Martinez, Damien Tassetti. Slicing techniques for temporal aggregation in spanning event streams. Information and Computation, 2021, pp.104807. 10.1016/j.ic.2021.104807 . hal-03403161

## HAL Id: hal-03403161
## https://hal.science/hal-03403161

Submitted on 5 Jan 2024

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution - NonCommercial 4.0 International License

Version of Record: https://www.sciencedirect.com/science/article/pii/S0890540121001231
Manuscript_45d7807254ba2ba3f8b65252707cc288

# Slicing Techniques for Temporal Aggregation in Spanning Event Streams

Aurélie Suzanne[a,b], Guillaume Raschia[a], José Martinez[a], Damien Tassetti[a]

[a] *LS2N; Université de Nantes, France*
[b] *Expandium, 15 Boulevard Marcel Paul, 44800 Saint-Herblain, France*

**Abstract**

Slicing is a popular approach to perform aggregation in streaming systems. It allows sharing computation costs among overlapping windows. However those systems are limited to point events. In this paper, we address the temporal aggregate computation issue in streams where events come with a duration, denoted as spanning event streams. After a short review of the new constraints ensued by event lifespan in a temporal sliding-window context, we propose a new structure for dealing with slices in such an environment, and prove that our technique is both correct and effective to deal with such spanning events. We then further extend this technique to compensate for the new constraints induced by the duration of spanning events with a multi-level structure able to reduce insertion costs.

*Keywords:* Data Stream, Spanning Events, Temporal Aggregates, Sliding Windows

## 1. Introduction

Windows have become a pillar of streaming systems. By keeping only the most recent data, they transform infinite flows of data into finite data sets, allowing aggregate functions. These aggregates continuously summarize the data, providing useful insights on the data at a low memory cost. Sliding windows advance across time, and, in many cases, two successive windows share events, leading to redundancy in computation between consecutive or intersecting windows. This redundancy can be avoided with slicing techniques that allow to pre-compute aggregates on sub-parts of the windows which can then be shared.

Up to now, these optimizations were limited to instantaneous events only, i.e., points in time, thereafter denoted as Point Event Streams (PES), and Spanning Event Streams (SES), where events are not assigned to a single point in time but rather to a time interval, have been overlooked. Those events also

*Email addresses:* aurelie.suzanne@ls2n.fr (Aurélie Suzanne), guillaume.raschia@ls2n.fr (Guillaume Raschia), jose.martinez@ls2n.fr (José Martinez), damien.tassetti@etu.univ-nantes.fr (Damien Tassetti)

*Preprint submitted to Information and Computation* *October 5, 2021*

© 2021 published by Elsevier. This manuscript is made available under the CC BY NC user license
https://creativecommons.org/licenses/by-nc/4.0/

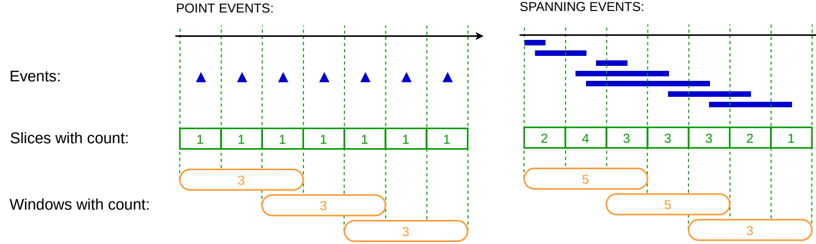



Figure 1: Slices and windows with point vs. spanning events

need to be handled efficiently, as they are used in many applications, e.g., monitoring systems like telecommunications or transportation. As an example, in a telecommunication network, antennas transmit spanning events like phone calls and monitoring them is crucial. A common metric would retrieve, every five minutes, the number of devices connected during the last fifteen minutes. With PES we would need to choose between the connection and disconnection of a device to an antenna as an indicator. With SES we can fully use the phone call interval, from connection to disconnection, hence improving the accuracy of the analysis. In that case, events with device connection, disconnection, or ongoing calls, can all three be handled correctly.

**Problem Statement.** In this communication we focus on the problem of efficient query processing for sliding windows dealing with SES. For this purpose we extend current streaming systems and their slicing techniques to SES. This extension cannot be done straightforwardly, as lifespan of events incurs side effects and an overhead cost on slice computation. Indeed, one spanning event may cover several slices, which implies that aggregates, such as the count of events, cannot be deduced from the partial aggregates in the covering slices. Figure 1 illustrates this problem. With spanning events on the right, the first window (orange bar) contains five events (blue lines), while the count of events we can deduce from the covering slices (green rectangles) is nine. With point events (blue triangles on the left), the direct count of events and the count from the slices coincide.

**Contributions.** In this paper, we propose a novel slicing technique designed for SES. Our technique relies on a new slice structure, adapted to the sensibility to duplication of aggregate functions. From that slice data model, we develop an event insertion workflow that takes care of events and slice bounds. We also propose a slice combination function to ensure that no duplication occurs at window release. As long events need to be inserted in potentially many slices, we further extend our technique to a multi-level data model which allows merging adjacent slices and reduce the number of insertions. Those aggregate computation techniques are agnostic to the way slices are created, such that one may use the best fitted stream slicer for a given use case.

**Road Map.** In order to do so, Section 2 deals with background definitions to extend streams and windows for spanning events. It subsequently gives key

concepts to introduce an extension of the PES slice model required to support spanning events in Section 3. Section 4 presents an efficient generalization of that model. In Section 5 we expose and study algorithms to perform temporal aggregate computations in SES thanks to our slice data model. We experiment slicing algorithms supporting our model and its extension in Section 6, compared to state-of-the-art aggregation techniques. Then, Section 7 reviews prior works in the data stream and temporal database processing fields. We conclude in Section 8.

## 2. Preliminaries

### *2.1. Timeline and Time Intervals*

Time is represented as an infinite, totally ordered, discrete set $(\mathbb{T}, \prec_{\mathbb{T}})$, where each time point $c$ is called a *chronon* [1]. (Discrete) Intervals are expressed with a lower and an upper bound, as pairs $(\ell, u) \in \mathbb{T} \times \mathbb{T}$ with $\ell \prec_{\mathbb{T}} u$. By convention, a time interval is written as a left-closed–right-opened interval $[\ell, u)$. We denote by $\mathbb{I} \subset \mathbb{T} \times \mathbb{T}$ the set of time intervals. For any $t \in \mathbb{I}$, $\ell(t) \in t$ and $u(t) \notin t$ are respectively the lower and upper bounds of the interval $t$. A chronon $c$ can be represented by the interval $[c, c+1)$.

Two intervals can be compared with the thirteen Allen's predicates [2]. We introduce three predicates as a combination of Allen's base predicates, which will prove useful hereafter. They are illustrated in Figure 2. Their corresponding formal definitions are as follows:

- $\mathrm{P}_{\cap}(a, b) := \ell(a) < u(b) \wedge \ell(b) < u(a)$, i.e., time intervals $a$ and $b$ have at least one chronon in common;
- $\mathrm{P}_{\dashv}(a, b) := \ell(b) < u(a) \leq u(b)$, i.e., time interval $a$ ends in $b$, an asymmetric relation;
- $\mathrm{P}_{\rightarrow}(a, b) := \ell(a) < u(b) < u(a)$, i.e., $a$ overlaps and goes beyond $b$, asymmetric too.

It is worth noting that $\mathrm{P}_{\cap} = \mathrm{P}_{\dashv} \vee \mathrm{P}_{\rightarrow}$ and $\mathrm{P}_{\dashv} \wedge \mathrm{P}_{\rightarrow} = \perp$. In other words, $\forall a, b \in \mathbb{I}$, if $a$ overlaps $b$, $a$ either finishes inside $b$ time range or goes beyond.

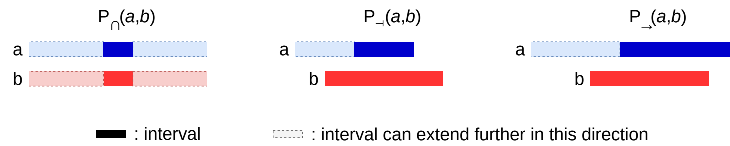



Figure 2: The three interval comparison predicates used in this paper

*2.2. Spanning Event Stream*

Within our SES framework, we consider that each event comes with a time interval, adding the notion of lifespan to events. Instantaneous events can still be modeled with a single-chronon interval. We consider that events are received after their ending.

**Definition 1 (Spanning Event Stream).** *A Spanning Event Stream $S$ is as follows, where $\Omega$ corresponds to any set of values, whether structured or not, that brings the contents of each event $e \in S$, and $\mathsf{t}(e) = t$ is a time interval that gives the lifespan of the event $e$:*

$$S = (e_i)_{i \in \mathbb{N}} \text{ with } e_i = (x, t) \in \Omega \times \mathbb{I}$$

In the following, the generic function $\mathsf{t}(.)$ will be used to denote the time component of any temporal data structure.

The order of events in the stream obeys the constraint: $\forall (e, e') \in S^2,\ e < e' \Leftrightarrow u(\mathsf{t}(e)) \prec_{\mathbb{T}} u(\mathsf{t}(e'))$, which means that events are ordered by their end time. In this paper, only on-time events are considered. This implies that events are received as soon as their upper bound is reached, i.e., at time $u(\mathsf{t}(e))$. The set of streams is denoted by $\mathcal{S}$.

*2.3. Aggregate Functions*

Streaming systems require continuous queries to process the data. As data load often makes it impossible to process data individually by an end-user application, a common solution is to use aggregates. Many aggregate functions exist, which can often be studied by categories rather than individually. We propose two classifications, based on their properties for slices and spanning events. Table 1, gives an overview of the most popular aggregate functions [3], with their associated classification.

One can distinguish several **algebraic properties** [4, 5, 6, 7, 8] such as: *distributive*: a stream can be split into sub-streams and there exist some functions to compute the final aggregate from sub-aggregates, e.g., a count can be computed from a set of sub-counts; *algebraic*: an aggregate can be computed from a list of distributive aggregates, e.g., a mean can be computed from sum's and corresponding count's sub-aggregates; *holistic*: some functions do not belong to any of the above categories, e.g., median. No constant upper bound on storage applies for the last category of functions, which yields to using specifically tailored algorithms. For this reason, holistic functions are not considered in this paper.

One can also distinguish among **accumulative properties** [9]: *cumulative*: an aggregate is an accumulation of all the events, e.g., count adds one for each event; *selective*: an aggregate keeps only one event, in its original form, e.g., max keeps only the maximum value. Cumulative functions are sensitive to event duplicates that can happen as a consequence of working with SES. Therefore, we shall study these categories separately.

Table 1: Classification of the most popular aggregate functions

| | Aggregate function | Algebraic prop. | Accumulative prop. |
|---|---|---|---|
| **sum-like** | count, sum | distributive | cumulative |
| | mean, standard-deviation | algebraic | cumulative |
| **max-like** | max, min | distributive | selective |
| | argmax, argmin | algebraic | selective |
| | maxCount, minCount | algebraic | cumulative |
| **collect-like** | collect, concatenate (string) | holistic | cumulative |
| | ith-youngest | holistic | selective |
| **median-like** | median, percentile | holistic | cumulative |
| | ith-smallest | holistic | selective |

*2.4. Temporal Sliding Window in* SES

Aggregate functions are blocking operators that require window definition in order to be performed on data streams. Windows split the infinite stream into finite sub-streams from which events can actually be aggregated. Sliding windows have the particularity to advance with time independently from the stream. They are associated to time intervals that are defined by two parameters: the size or range $\omega$, and the step $\beta$ which determines how fast the window advances in time. Overlaps can happen in such windows as soon as $\omega > \beta$, e.g., a window of size $\omega = 15$ minutes advancing each $\beta = 5$ minutes. Altogether, they provide an infinite list of sliding windows over the data stream $S$.

**Definition 2 (Sliding Window Family).** *A Sliding Window Family $W_S$ is as follows, where $S_{w_i}$ is a finite sub-stream of $S$ containing the events that occurred in the window $w_i$, and $t_i$ is the time interval of $w_i$, also denoted $\mathsf{t}(w_i)$:*

$$W_S = (w_i)_{i \in \mathbb{N}} \text{ with } w_i = (S_{w_i}, t_i) \in \mathcal{S} \times \mathbb{I}$$

At run-time, window life-cycle goes through several steps: *window creation* is triggered when the window lower bound is reached, and *window release* is triggered as soon as the upper bound is reached. Between these two triggers, the window accumulates all the incoming events of interest. At window release, the system computes the final aggregate.

Following this general workflow, SES however, requires to investigate further in order to handle temporal sliding windows. Firstly, it is worth noting that window creation is not impacted by spanning events as the bounds of the window, $\ell(\mathsf{t}(w))$ and $u(\mathsf{t}(w))$, are independent from the stream content. But an incoming event is both assigned to the current window and in several past windows depending on its lifespan. Indeed, event assignment to a window is decided according to the intersection predicate $\mathrm{P}_\cap$, as defined in Section 2.1. Thus, triggering a window release immediately at closing time would yield to missing events for this window because they are still ongoing and they will be caught only in the future.

To overcome this problem, we introduce a Time-To-Postpone (TTP) parameter. Its role is to delay the window release, with a trigger now occurring at the

window upper bound plus the TTP. Of course, this value needs to be chosen very carefully as long-standing events could still arrive after the TTP. Various methods exist to define this value, from a fixed user-defined value to an evolving value continuously learned by the system. Accurate definition of the TTP is out of the scope of this communication, since our main focus is the aggregate computation. Hence we consider the basic setting with a constant TTP value that is expected to be larger than the largest event. With this additional delay for window release, any event is now taken into account in active windows.

## 3. Slices

### 3.1. Point Event Slices

Slicing techniques divide windows into non-overlapping slices which keep events in the form of continuously updated sub-aggregates. These sub-aggregates are then combined in order to compute the final window aggregate. Advantages of slices are numerous: (i) they limit memory usage by requiring only one aggregate per slice instead of buffering all the events, and (ii) they reduce spikes in the system at window release since only partial aggregates need to be computed, and (iii) they allow for sharing computation among windows [10, 11, 12].

**Definition 3 (PES Slice Data Model).** *We define a series of slices $\Gamma_{W,S}$ depending on a window family $W$ and a data stream $S$ as follows, where $\phi \in \Phi$ is an internal slice structure that stores the partial aggregate value, and $t = \mathsf{t}(\gamma_i)$ is the time interval of the slice $\gamma_i$:*

$$\Gamma_{W,S} = (\gamma_i)_{i \in \mathbb{N}} \text{ with } \gamma_i = (\phi, t) \in \Phi \times \mathbb{I}$$

The internal slice structure $\phi$ depends on the aggregate function, e.g., sum would store a sum for each slice, whereas mean would store a sum and a count. A list of internal structures for common aggregate functions can be found in [3].

Slices obey two properties:

**P1** $\forall (i,j) \in \mathbb{N}^2, i \neq j \to \neg \mathrm{P}_\cap(\mathsf{t}(\gamma_i), \mathsf{t}(\gamma_j))$: two slices cannot overlap;

**P2** $\forall i \in \mathbb{N}, u(\mathsf{t}(\gamma_i)) = \ell(\mathsf{t}(\gamma_{i+1}))$: two successive slices meet, in Allen's meaning.

Those properties ensure that $\Gamma_{W,S}$ is a time partitioning of the stream $S$, w.r.t. a family window $W$. The exact conditions on slice bounds are given by the slicing policy that we are going to discuss in Section 5.

To use the slices, we adopt the incremental aggregation method introduced by Tangwongsan et al. [3] and reused in [13]. This approach is based on the three functions: lift, combine and lower. They are illustrated in Figure 3, and informally described as follows:

- lift : $S \to \Phi$, prepares events for an ongoing insertion in slices. It is triggered when an event arrives in the system, and it transforms the event to fit the internal slice structure.

- combine : $\Phi^2 \to \Phi$, merges two slices into a new one. This operation is used both at event insertion and at window release, as shown on Figure 3.
- lower : $\Phi \to$ Agg, computes the final aggregate from the internal structure of a slice, in order to actually release a window.

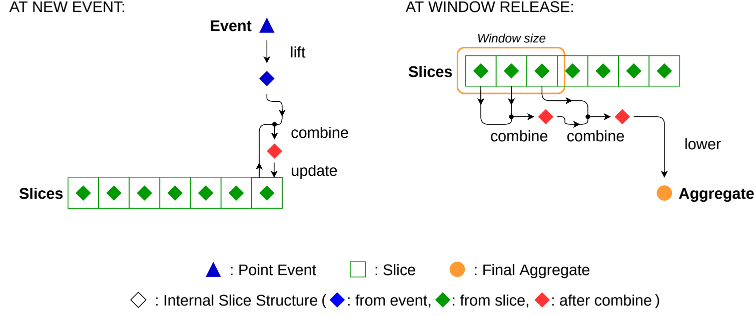



Figure 3: Usage of functions lift, lower and combine to insert events and release aggregates in PES.

As an example, we want to know, every 5 minutes, the number of device disconnections and the maximum call duration for an antenna for the past 15 minutes. For such a query, the slice structure would consist in partial counts and max. A typical scenario is as follows:

- The **initial state** of the internal structure contains seven **5-minute slices**, where $n$ represents the partial counts, and max the partial maximums:

| time | 0 | 5 | 10 | 15 | 20 | 25 | 30 | 35 |
|---|---|---|---|---|---|---|---|---|
| n | 8 | 19 | 15 | 18 | 14 | 12 | 16 | |
| max | 20 | 63 | 19 | 33 | 12 | 47 | 14 | |

- A **new event** with a call duration of 18 minutes arrives at **time 34**, then:
  1. the event is transformed by lift, giving $(n = 1, \max = 18)$;
  2. this lifted event is combine'd with the latest slice, as illustrated below.

| time | 0 | 5 | 10 | 15 | 20 | 25 | 30 | 35 |
|---|---|---|---|---|---|---|---|---|
| n | 8 | 19 | 15 | 18 | 14 | 12 | 17 | |
| max | 20 | 63 | 19 | 33 | 12 | 47 | 18 | |

event 34

- At **window release**, we compute a window of size **15 minutes** considering a large TTP value of 20 minutes. There are two steps to process the $[0, 15)$ window:
  1. incrementally combine the first three slices. A first combine is applied on the first two slices and gives $(n = 8 + 19 = 27, \max = \max(20, 63) = 63)$. Then, a second combine on the previous result and third slice results in $(n = 27{+}15 = 42, \max = \max(63, 19) = 63)$;
  2. apply lower to output a count of 42 and a maximum of 63.

### *3.2. Spanning Event Slices*

With spanning events, one event can find itself in several slices as shown on Figure 1, which is obviously not an issue for point events since slices are non-overlapping. Hence, we have to adapt the above slicing technique to SES. Firstly, it requires to update several slices during the insertion, at the combine level. However, this quickly leads to a duplication problem that must be leveraged. As their sensitivity to duplication varies, we shall study selective and cumulative aggregate functions separately.

#### *3.2.1. Selective Aggregate Functions*

The former is the simpler. By definition, selective functions, e.g., max, only retains one event by slice, thus duplication is not a problem. We can compute selective aggregate functions with the same slice model than the one for PES, following an almost identical workflow. Figure 4 illustrates the slicing workflow for selective aggregate functions. We can see that only the combine step at the insertion of a new event is modified. Indeed, *all the slices that have a non-empty intersection with the event* need to be updated, rather than the latest one only.

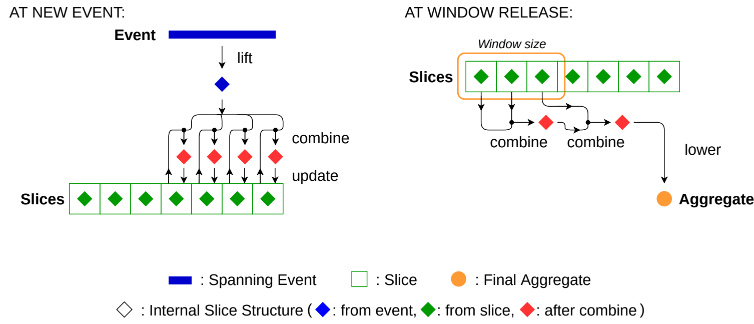



Figure 4: Usage of functions lift, lower, and combine to insert events and release aggregates in SES with selective aggregate functions.

Back to our phone call example, one focuses on the selective function, i.e., the maximum call duration, as follows:

- The **initial state** contains slices with their local maximum values:

| time | 0 | 5 | 10 | 15 | 20 | 25 | 30 | 35 |
|---|---|---|---|---|---|---|---|---|
| max | 20 | 63 | 19 | 33 | 12 | 47 | 14 | |

- A **new event** arrives, say an **18-minute** phone call at **time 34**:
  1. The event is lift'ed into (18);
  2. This lifted event is combine'd to *each related slice*, the last four slices here.

| time | 0 | 5 | 10 | 15 | 20 | 25 | 30 | 35 |
|---|---|---|---|---|---|---|---|---|
| max | 20 | 63 | 19 | 33 | 18 | 47 | 18 | |
| event | | | | 16 | | | 34 | |

- At **window release**, the steps to process the $[0, 15)$ window are:
    1. incrementally combine the first three slices. This gives $(\max(20, 63) = 63)$ after the first combine, then $(\max(63, 19) = 63)$ in the second round;
    2. apply lower to output a maximum of 63.

*3.2.2. Cumulative Aggregate Functions*

Cumulative functions accumulate the data; hence they are sensitive to event duplication among slices. To compensate for this problem, we extend the internal slice structure, as shown in the following definition.

**Definition 4 (SES Slice Data Model).** *We define a series of slices $\Gamma_{W,S}$ adapted for SES cumulative aggregate functions as follows, where both $\phi \in \Phi$ and $\varphi \in \Phi$ are internal slice structures that stores the partial aggregate value, and $t = \mathsf{t}(\gamma_i)$ is the time interval of the slice $\gamma_i$:*

$$\Gamma_{W,S} = (\gamma_i)_{i \in \mathbb{N}} \text{ with } \gamma_i = (\phi, \varphi, t) \in \Phi^2 \times \mathbb{I}$$

We propose the $(\phi, \varphi)-$structure to separate events that end in the slice $(\phi)$, defined with $\mathrm{P}_{\dashv}(\mathsf{t}(e), \mathsf{t}(\gamma))$, from events that continue after the slice $(\varphi)$, defined by $\mathrm{P}_{\rightarrow}(\mathsf{t}(e), \mathsf{t}(\gamma))$. To show that our extension performs the expected computation we use the properties of our defined interval comparison predicates, which states that an event overlapping a slice interval either finishes inside its time range or goes beyond.

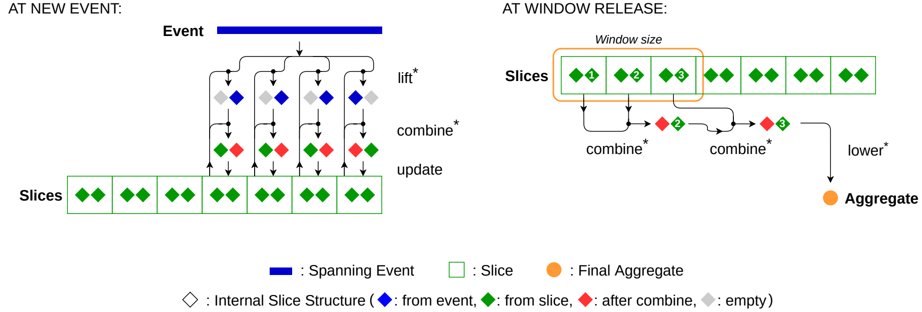



Figure 5: Usage of functions lift*, lower* and combine* to insert events and release aggregates in SES with cumulative aggregate functions. The internal structure is split to separate the events which end in the slice $\phi$ on the left, and the events which end after the slice $\varphi$ on the right.

The aggregation process uses modified versions of the lift, combine, and lower operators as described in Section 3.1. This is illustrated in Figure 5. A formal definition of these modified versions is given in Table 2.

The new functions operate in the following way:

Table 2: Extension ($*-$form) of slice operators to the $(\phi,\varphi)-$structure for $\mathsf{SES}$.

$$\begin{aligned}&\mathsf{lift}^*(e:S,t:\mathbb{I})\rightarrow(\phi,\varphi,t):\Gamma_{W,S}\\&\quad\phi=\mathsf{lift}(e)\text{ if }\mathrm{P}_{\dashv}(\mathsf{t}(e),t)\text{ else }0_{\mathsf{Agg}}\\&\quad\varphi=\mathsf{lift}(e)\text{ if }\mathrm{P}_{\rightarrow}(\mathsf{t}(e),t)\text{ else }0_{\mathsf{Agg}}\end{aligned}$$

$$\begin{aligned}&\mathsf{lower}^*((\phi,\varphi,_):\Gamma_{W,S})\rightarrow y:\mathsf{Agg}\\&\quad y=\mathsf{lower}(\mathsf{combine}(\phi,\varphi))\end{aligned}$$

$$\begin{aligned}&\mathsf{combine}^*((\phi_a,\varphi_a,a):\Gamma_{W,S},(\phi_b,\varphi_b,b):\Gamma_{W,S})\rightarrow(\phi,\varphi,a\cup b):\Gamma_{W,S}\\&\quad\text{assert}\quad u(a)=\ell(b)\text{ or }u(b)=\ell(a)\text{ or }a=b\\&\quad\phi=\mathsf{combine}(\phi_a,\phi_b)\\&\quad\varphi=\mathsf{combine}(\varphi_a,\varphi_b)\text{ if }a=b\text{ else }\varphi_{\max\{a,b\}}\end{aligned}$$

- $\mathsf{lift}^* : S,\mathbb{I}\rightarrow\Gamma_{W,S}$: promotes each event as a $(\phi,\varphi)$ slice item. Choosing among the slice part to contribute to is made thanks to $\mathrm{P}_{\dashv}$ and $\mathrm{P}_{\rightarrow}$ conditions, respectively for $\phi$ and $\varphi$. Each event is eligible to only one of them, and the non-eligible part is let empty. Basically, as one can see on Figure 5, the event contributes to the $\phi$ part of the most recent slice, and to the $\varphi$ part of all other intersecting slices. This implies that the $\mathsf{lift}^*$ operation depends on the interval of the slice, and should be computed for each slice;
- $\mathsf{combine}^* : \Gamma^2_{W,S}\rightarrow\Gamma_{W,S}$: behaves differently depending on the moment it is triggered. When $\mathsf{combine}^*$ is triggered *at events insertion*, it will rely on the raw $\mathsf{combine}$ operator from [3] to update as much $\phi$ as $\varphi$. We can, however, note, as shown in Figure 5, that only one of them is updated as the event cannot contribute to both at the same time during the $\mathsf{lift}^*$ step. When $\mathsf{combine}^*$ is triggered *at window release* , it ignores the $\varphi$ part of the oldest slice to prevent event duplication, since an event in $\varphi$ necessarily contributes to the next slice, either in $\phi$ or $\varphi$. Hence updating only the most recent $\varphi$ ensures neither to duplicate the event nor to forget it. This behavior can be seen on Figure 5 where, at each $\mathsf{combine}^*$, only the $\varphi$ of the most recent slice is considered.
- $\mathsf{lower}^* : \Gamma_{W,S}\rightarrow\mathsf{Agg}$: merges the distinct parts $\phi$ and $\varphi$ to provide the exact aggregate value.

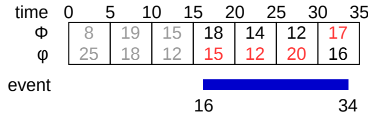
As neither event duplication nor omission are possible with the $(\phi,\varphi)-$structure, we claim that all popular cumulative aggregate functions can be used with this new structure.

We continue our example with the cumulative function part, and use these new $\mathsf{lift}^*$, $\mathsf{combine}^*$ and $\mathsf{lower}^*$ functions to count the number of devices connected to an antenna with spanning events.

- In the **initial state**, $\phi$ and $\varphi$ both represent partial counts.

| time | 0 | 5 | 10 | 15 | 20 | 25 | 30 | 35 |
|---|---|---|---|---|---|---|---|---|
| Φ | 8 | 19 | 15 | 18 | 14 | 12 | 16 | |
| φ | 25 | 18 | 12 | 14 | 11 | 19 | 16 | |

- When a **new event** arrives, the **18-minute** phone call at **time 34**:
  1. the event is transformed with lift* into ($\phi = 1, \varphi = 0$) for the most recent slice, whereas it gives ($\phi = 0, \varphi = 1$) for the three previous slices;
  2. this lifted event is then combined with each related slice, which corresponds to the latest four slices.




- At **window release**, we process the $[0, 15)$ window as follows:
  1. use combine* twice for the first three slices. This gives ($\phi = 8 + 19 = 27, \varphi = 18$) after the first combine*, then ($\phi = 27 + 15 = 42, \varphi = 12$) after the second;
  2. apply lower to output a (correct) count of $42 + 12 = 54$.

## 4. Multi-Level Slices

Streaming systems are dealing with extremely high ingestion rates, and hence one of the main concerns for our system is the insertion cost. Furthermore, we saw that spanning events can overlap with several slices, which implies multiple slice insertions for a single event. In this section, we propose a generalization of the slice model with a *multi-level slice structure* allowing managing long-standing events within a unique slice instead of several ones.

### *4.1. Multi-Level Model*

We first assume that one can reduce insertion cost by controlling the number of slices in which we insert events. The proposed structure is meant to work with both selective and cumulative aggregate functions. Hence, in this section, we will not distinguish between those two types of aggregate functions.

**Definition 5 (SES Multi-Level Slice Data Model).** *Given a height $d \in \mathbb{N}$, one creates $d$ levels by slice, each of them containing a structure similar to the one of Definition 4, i.e., it has a time interval $t \in \mathbb{I}$ and a $(\phi, \varphi) \in \Phi^2$ structure. We define a series of multi-level slices $\Gamma_{W,S}$ as follows:*

$$\Gamma_{W,S} = ((\gamma_i^k)_{1 \leq k \leq d})_{i \in \mathbb{N}} \textit{ with } \gamma_i^k = (\phi, \varphi, t) \in \Phi^2 \times \mathbb{I}$$

It is worth noticing that selective functions only require a $\phi \in \Phi$ structure. This generalized slice model is schematized on Figure 6.

In Definition 5, $\gamma_i^k$ is the slice structure at the level $k$ for the slice $\gamma_i$, which we further denote as *slice level*; $\mathtt{t}(\gamma_i^k) = t$ is the time interval of that level. When the height $d$ is equal to one, it is strictly equivalent to the Definition 4. Therefore, Definition 5 provides a generalized form of the spanning event slice structure.

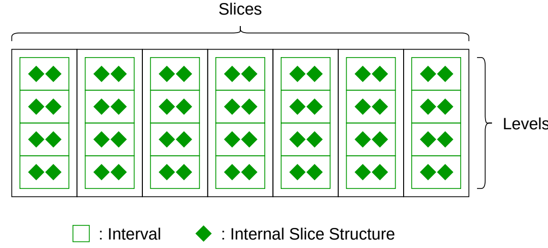



Figure 6: Outline of our multi-level slices data model with a height of 4

### 4.2. Time Intervals of Upper Slice Levels

The base level ($k = 1$) of each slice defines the actual time interval $\mathtt{t}(\gamma^1) = t$. Those slice ranges are non-overlapping and they are given by the slicing technique (see Section 5), as the "flat" slice model.

Intervals for levels $k > 1$ are created as a composition of raw slice intervals, as follows:

$$\forall i \in \mathbb{N}, \forall k \in [\![2, d]\!], \mathtt{t}(\gamma_i^k) = \bigcup_{j=i-k+1}^{i} \mathtt{t}(\gamma_j^1) \quad (1)$$

For instance, level 2 covers the current and the previous slices, and so on, until level $d$ which will contain the $d - 1$ previous slices. An example is given by slice 7 on Figure 7.

In Equation 1, one considers $\mathtt{t}(\gamma_j^1) = 0_{\mathbb{I}}$ for $j \leq 0$, to fix the equation at the very beginning of the slice timeline (when $i < k$).

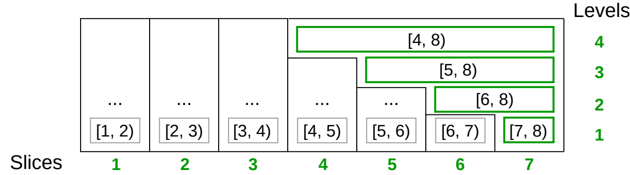



Figure 7: Time intervals of levels 1 to 4 of slice 7 in the multi-level slice structure with slices of size 1 and a height of 4.

This implies that for levels higher than 1 the constraint of non-overlapping does not hold. To avoid any redundancy at event insertion, we define some additional properties:

**P3** $\forall i \in \mathbb{N}, \forall p, q \in [\![1, d]\!], p \neq q, S_{\gamma_i^p} \cap S_{\gamma_i^q} = \emptyset$: an event can contribute to at most one level of a given slice;

**P4** $\forall i, j \in \mathbb{N}, \forall p, q \in [\![1, d]\!], i \neq j, \mathrm{P}_{\cap}(\mathsf{t}(\gamma_i^p), \mathsf{t}(\gamma_j^q)) \rightarrow S_{\gamma_i^p} \cap S_{\gamma_j^q} = \emptyset$: an event cannot contribute to any slice level which overlaps with a slice level where it has been inserted.

In the above properties, $S_{\gamma_i^k}$ denotes the sub-stream of events from $S$ that contribute to the level $k$ of the slice $\gamma_i$.

*4.3. Working with the Multi-Level Slice Model*

In the multi-level technique, we use the $(\phi, \varphi)$-structure in the same way as with cumulative functions for SES (see Section 3.2.2). This allows to separate events according to their end time. Then, for an event to contribute to a slice, whatever the level, it must intersect with the raw slice interval $\mathsf{t}(\gamma^1)$. Then, to satisfy property **P4**, insertion into past slices may skip contiguous slices and jump directly to non-overlapping ones.

To release a window $w$, we combine all the slice levels $\gamma_i^k$ which overlaps with the window, i.e., such that $\mathrm{P}_{\cap}(\mathsf{t}(w), \mathsf{t}(\gamma_i^k)) \neq \emptyset$.

As for any level $k > 1$, slices are overlapping with previous slices, we also need to include upper levels for slices where $u(\mathsf{t}(w)) \leq \ell(\mathsf{t}(\gamma_i^1)) \wedge u(\mathsf{t}(w)) > \ell(\mathsf{t}(\gamma_i^k)))$. Note that those slices are not used for release in the case where the height is equal to one, as in Section 3.2.2. For all slice levels ending with or after the window, i.e., where $u(\mathsf{t}(w)) \leq u(\mathsf{t}(\gamma_i^k)))$, property **P4** ensures us that the event contributing to $\varphi$ are not contributing to any other later slice. Thus we need to change the combine* function, as exposed in Table 3. This new function takes into account that when $u(\mathsf{t}(w)) \leq u(\mathsf{t}(\gamma_i^k))$: (i) we need to combine both $\phi$ and $\varphi$ for slice levels, (ii) the interval output from the combine function cannot have an upper bound higher than $u(\mathsf{t}(w))$. This update implies that combine* is now asymmetric with parameters order impacting the final result. The left part (a) contains the future final result, while the right part (b) contains the slice level to add to the result.

Table 3: Rewrite of the combine* operator for the SES multi-level slice model.

$$
\begin{aligned}
&\mathsf{combine}^*((\phi_a, \varphi_a, a) : \Gamma_{W,S}, (\phi_b, \varphi_b, b) : \Gamma_{W,S}) \rightarrow (\phi, \varphi, t) : \Gamma_{W,S} \\
&\quad \phi = \mathsf{combine}(\phi_a, \phi_b) \\
&\quad \varphi = \mathsf{combine}(\varphi_a, \varphi_b) \text{ if } u(a) \leq u(b) \text{ else } \varphi_b \\
&\quad t = a \cup b \text{ if } u(a) = \ell(b) \text{ else } a
\end{aligned}
$$

*4.3.1. Example*

Let's use an example to help understanding the multi-level slice model and the way it works. To this end, we follow on our running example and count

the number of devices connected to an antenna. It is then a scenario with a cumulative function. For a selective function, as with "flat" slices, only one internal structure would be required, the rest would work similarly. In order to skip the initialization phase, where higher levels cannot be created as there is no previous slices, we consider slices starting at time 50. Slices are defined every 5 minutes and they have 3 levels.

- The **initial state** is made of six **5-minute slices** with **3 levels** each. A slice level has a time interval (in gray) and a partial count, which is divided between its $\phi$ part on the top and its $\varphi$ part on the bottom:

| | | | | | | |
|---|---|---|---|---|---|---|
| 3 | [40, 55)<br>4<br>3 | [45, 60)<br>5<br>6 | [50, 65)<br>5<br>7 | [55, 70)<br>4<br>3 | [60, 75)<br>4<br>7 | [65, 80)<br>5<br>6 |
| 2 | [45, 55)<br>5<br>2 | [50, 60)<br>7<br>4 | [55, 65)<br>4<br>5 | [60, 70)<br>3<br>4 | [65, 75)<br>5<br>4 | [70, 80)<br>4<br>3 |
| 1 | [50, 55)<br>5<br>6 | [55, 60)<br>3<br>5 | [60, 65)<br>6<br>9 | [65, 70)<br>5<br>7 | [70, 75)<br>5<br>4 | [75, 80)<br>5<br>5 |

- When a **new event** arrives, a phone call with the duration of **18 minutes** at **time 79**, giving a valid time of $[61, 79)$, one observes that it covers 4 slices:

  1. the event is inserted into the slice where it ends, here $[75, 80)$, at the highest level, which is $k = 3$ corresponding to range $[65, 80)$. The event is transformed with $\mathsf{lift}^*$ into $(\phi = 1, \varphi = 0)$. Then, it is inserted with $\mathsf{combine}^*$, updating the slice level from $(\phi = 5, \varphi = 6)$ to $(\phi = 6, \varphi = 6)$.
  2. As the beginning of the event is older than 65, we look at the first previous slice which does not overlap the current slice level, i.e., $[60, 65)$. Since the event starts in that slice level, one has to $\mathsf{lift}^*$ it into $(\phi = 0, \varphi = 1)$ and $\mathsf{combine}^*$ with the raw slice level to get $(\phi = 6, \varphi = 10)$.

| | | | | | | |
|---|---|---|---|---|---|---|
| 3 | [40, 55)<br>4<br>3 | [45, 60)<br>5<br>6 | [50, 65)<br>5<br>7 | [55, 70)<br>4<br>3 | [60, 75)<br>4<br>7 | [65, 80)<br>6<br>6 |
| 2 | [45, 55)<br>5<br>2 | [50, 60)<br>7<br>4 | [55, 65)<br>4<br>5 | [60, 70)<br>3<br>4 | [65, 75)<br>5<br>4 | [70, 80)<br>4<br>3 |
| 1 | [50, 55)<br>5<br>6 | [55, 60)<br>3<br>5 | [60, 65)<br>6<br>10 | [65, 70)<br>5<br>7 | [70, 75)<br>5<br>4 | [75, 80)<br>5<br>5 |

event: 61 — 79

- At **window release**, we process the $[50, 65)$ window as follows:

  1. successively $\mathsf{combine}^*$ all slice levels which overlap with the window. To this end, when the slice level ends within or after the window

interval, one incorporates both the $\phi$ and $\varphi$ parts; otherwise one uses only the $\phi$ part. Activated values are shown below (in red). The resulting structure is $(\phi = 55, \varphi = 36)$.

| | | | | | | |
|---|---|---|---|---|---|---|
| 3 | [40, 55) 4 3 | [45, 60) 5 6 | [50, 65) 5 7 | [55, 70) 4 3 | [60, 75) 4 7 | [65, 80) 6 6 |
| 2 | [45, 55) 5 2 | [50, 60) 7 4 | [55, 65) 4 5 | [60, 70) 3 4 | [65, 75) 5 4 | [70, 80) 4 3 |
| 1 | [50, 55) 5 6 | [55, 60) 3 5 | [60, 65) 6 10 | [65, 70) 5 7 | [70, 75) 5 4 | [75, 80) 5 5 |

2. then, apply lower$^*$ to output a (correct) count of 91.

## 5. Stream Slicer

### *5.1. Candidate Frameworks to Meet the SES Requirements*

Along with the multi-level slice model, a method to both insert events in slices and release window aggregates from those slices was proposed. Then, one needs a system that is able to create such slices from the window parameters and the data stream. For sliding windows, several such systems already exist to address PES aggregation. One of the SES requirements is that past windows can be updated, and hence window start and end bounds must coincide with slices bounds to be able to recreate past windows. For this purpose, the algorithms Panes [11] and Pairs [14] are good candidates, while Cutty[10] is unsuitable as it marks only start bounds of windows and not end bounds. Panes [11] partitions the stream into constant size slices, equal to $\gcd(\beta, \omega)$. At the contrary, Pairs [14] creates at most two slices per step. When $\omega \mod \beta = 0$ both methods are equivalent, while Panes generates twice as many slices as Pairs when it is not the case. To reduce insertion and release costs [12], the goal of a stream slicer is to produce as few slices as possible, thus Pairs is more appropriate. Scotty [12] produces slices for each new window start or end, which makes the method equivalent to Pairs. Hence, we shall use the Pairs technique in this paper.

### *5.2. Slicing Algorithms*

#### *5.2.1. Single-Level Slicing*

We consider cumulative aggregate functions, and each entering event is straightforwardly lifted to the $(\phi, \varphi)$-structure, as defined in Section 3.2.

We use the Pairs technique to separate the input stream into slices. It creates up to two slices per step where the first slice is of size $|\mathsf{t}(\gamma_1)| = \omega \mod \beta$ (denoted $\varepsilon$ in Algorithm 2)and the second one of size $|\mathsf{t}(\gamma_2)| = \beta - |\mathsf{t}(\gamma_1)|$. This leads to $n_\beta = 2$ slices per step if $\omega \mod \beta > 0$, 1 otherwise, and $n_\omega = 2\lfloor \omega/\beta \rfloor + 1$ slices per window if $\omega \mod \beta > 0$, $\omega/\beta$ otherwise.

The slice-based SES aggregation process, coined SE-Slicing and exposed in Algorithm 1, uses an "event-at-a-time" execution model. In this algorithm, one considers $\tau \in \mathbb{T}$ as the clock, i.e., an infinite time counter starting from

$0_{\mathbb{T}}$. The $n_\omega$ and $n_\beta$ values are initialized (line 3 - *nb_slice*) with the above formulas. The conditions for the window start and end times (resp. lines 5 and 7) are performed with a $\mathbb{T}-$mark incremented by $\beta$ each time it is reached. $\delta$ corresponds to the TTP and delays window release. *read_stream*$(S, \tau)$ (line 9) retrieves the event $e$ at current time $\tau$ if it exists, nothing otherwise. *add_slices* (line 6) is detailed in Algorithm 2. It creates the missing slices for a new window, covering the most recent $\beta$ time range. *insert_event* on line 10 is explained in Algorithm 3. It starts from the most recent slice, scans backward, and stops as soon as it reaches a non-intersecting slice. The last operation, *release_window* on line 8 is given in Algorithm 4. It combines $n_\omega$ slices, corresponding to all the slices in a window, and then it lowers the result to release a final aggregate. It also deletes the $n_\beta$ oldest slices (Algorithm 4, line 4), since they are no more supporting any open window.

**Algorithm 1:** SE-Slicing

```
   input  : S ∈ 𝒮, ω ∈ ℕ, β ∈ ℕ, δ ∈ ℕ
1  τ : 𝕋 ← 0_𝕋
2  Γ : List⟨(Φ, Φ, 𝕀)⟩ as Slices ← ()
3  n_ω, n_β : ℕ² ← nb_slice(ω, β)
4  while True do
5  |  if window_begins_at(τ) then
6  |  |  add_slices(Γ, τ, ω, β)
7  |  if window_ends_at(τ − δ) then
8  |  |  release_window(Γ, n_ω, n_β)
9  |  if e ← read_stream(S, τ) then
10 |  |  insert_event(Γ, e)
11 |  τ ← τ + 1
```

**Algorithm 2:** add_slices

```
   input  : Γ ∈ Slices,
            τ ∈ 𝕋, ω ∈ ℕ, β ∈ ℕ
1  ε : ℕ ← ω mod β
2  if ε > 0 then
3  |  add (0, 0, [τ, τ + ε)) to Γ
4  add (0, 0, [τ + ε, τ + β)) to Γ
```

**Algorithm 3:** insert_event

```
   input  : Γ ∈ Slices, e ∈ S
1  i : ℕ ← |Γ| − 1
2  while P_∩(t(e), t(Γ[i])) ∧ i ≥ 0 do
3  |  Γ[i] ←
   |    combine*(Γ[i], lift*(e, t(Γ[i])))
4  |  i ← i − 1
```

**Algorithm 4:** release_window

```
   input  : Γ ∈ Slices,
            n_ω ∈ ℕ, n_β ∈ ℕ
1  γ : (Φ, Φ, 𝕀) ← (0, 0, t(Γ[0]))
2  for i ∈ [0, n_ω) do
3  |  γ ← combine*(γ, Γ[i])
4  delete slice 0 to n_β −1 from Γ
5  print lower*(γ)
```

*5.2.2. Multi-Level Slicing*

Multi-level slicing has a very similar algorithmic schema to SE-Slicing. Hence, it may reuse the main loop defined in the Algorithm 1, with two slight updates: (i) the height $d \in \mathbb{N}$ is a new parameter, and (ii) the Slices type becomes $\text{List}\langle(\Phi, \Phi, \mathbb{I})^d\rangle$ to match the multi-level slice structure of Definition 5. One denote by SE-ML-Slicing the resulting algorithm. Despite its similarity with SE-Slicing, the inner functions `add_slices`, `insert_event` and `release_window` need to be re-implemented.

As for the SE-Slicing technique, Pairs is the preferred framework to add new slices (see Algorithm 6). During the process, multi-level slice structures are allocated and initialized as stated in Algorithm 5. Multi-level slice structures

are fed with past raw slice levels (line 5). At the very beginning, we cannot fill all the levels and stop as soon as there is no previous slice (line 4). Hence, the first slice contains only one level, the second only two, and so on, until we reach $i \geq d$.

**Algorithm 5:** `create_slice`

```
  input : Γ ∈ Slices, d ∈ ℕ, t ∈ 𝕀
1 γ : (Φ, Φ, 𝕀)^d
2 γ[0] ← (0, 0, t)
3 for k ∈ [1, d) do
4     if |Γ| > k then
5         γ[k] ← (0, 0, [ℓ(t(Γ[|Γ| − k][0])), u(t)))
6 add γ to Γ
```

**Algorithm 6:** `add_slices`

```
  input : Γ ∈ Slices,
          τ ∈ 𝕋, ω ∈ ℕ, β ∈ ℕ, d ∈ ℕ
1 ε : ℕ ← ω mod β
2 if ε > 0 then
3     create_slice(Γ, d, [τ, τ + ε))
4 create_slice(Γ, d, [τ + ε, τ + β))
```

To insert an event, Algorithm 7 computes the number of slices $n_\gamma$ an event intersects with (lines 2-7). Then, a specific pathway is run to insert the event in as few slice levels as possible (lines 8-14).

**Algorithm 7:** `insert_event`

```
   input : Γ ∈ Slices, d ∈ ℕ,
           e ∈ S, n_β ∈ ℕ
1  i : ℕ ← |Γ| − 1
2  r : ℕ ←
       |t(e)| + u(t(Γ[i][0])) − u(t(e))
3  n_γ : ℕ ← ⌊r/β⌋ × n_β
4  if r mod β > |t(Γ[i][0])| then
5      n_γ ← n_γ + 2
6  else if r mod β > 0 then
7      n_γ ← n_γ + 1
8  while i ≥ 0 ∧ n_γ > 0 do
9      if n_γ ≥ d then
10         combine*(Γ[i][d − 1], lift*(e, t(Γ[i][0])))
11     else
12         combine*(Γ[i][n_γ − 1], lift*(e, t(Γ[i][0)))
13     i ← i − d
14     n_γ ← n_γ − d
```

**Algorithm 8:** `release_window`

```
  input : Γ ∈ Slices,
          n_ω ∈ ℕ, n_β ∈ ℕ, d ∈ ℕ
1 γ : (Φ, Φ, 𝕀) ← (0, 0, t(Γ[0][0]))
2 for i ∈ [0, n_ω) do
3     for k ∈ [0, d) do
4         γ ← combine*(γ, Γ[i][k])
5 for k ∈ [1, d) do
6     for i ∈ [0, min(k, |Γ| − n_ω))
       do
7         γ ←
            combine*(γ, Γ[n_ω + i][k])
8 delete slice 0 to n_β − 1 from Γ
9 print lower*(γ)
```

Writing into slices starts from the most recent slice, and inserts in the highest possible level. If it needs to insert into more slices than the height ($n_\gamma > d$), then it inserts into level $d$ (lines 9-10). As other slices still need to be written, it goes backward in the $\Gamma$ slice list, skipping the ones which are already covered. The next eligible slice $\gamma_j$ ends at the beginning of the slice level $\gamma_i^d$ in which the event was inserted, such as $\ell(\mathsf{t}(\gamma_i^d)) = u(\mathsf{t}(\gamma_j^1))$. This slice hop from $i$ to $j$ is determined by the height $d$ of the structure, and actually, the destination slice number is $j = i - d$ (line 13). The algorithm continues inserting the event in past

slices (with possible hops) until it covers the slice level $\gamma^1_{i-n_\gamma}$. The successive steps to insert an event are illustrated on Figure 8. The insertion starts on the right with the most recent slice and then, it goes back in time. The event spans over 7 slices and the height is 4; hence the first insertion is performed at the highest level (level 4) of the most recent slice (slice 8) which covers 4 slices (from 5 to 8). Then, a second insertion arises at level 3 of slice $8-4=4$ in order to cover the 3 remaining slices (from 2 to 4). The event is therefore always inserted in a minimum number of slices thanks to the highest level insertion policy. It is important to note that this multi-level slice structure requires a specific path to insert events into slice levels (in particular for the hoping phase).

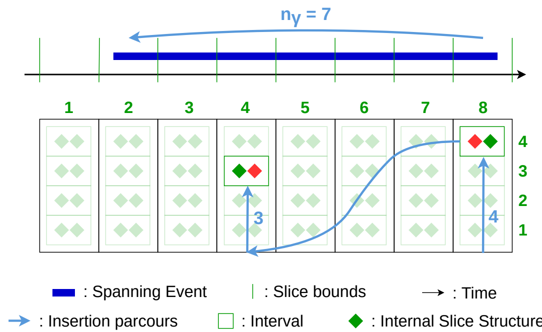



Figure 8: Insertion of an event in the multi-level slice structure. Only the required slice levels are enabled. The red diamonds represent parts of the internal structures that are updated.

Release of a window traverses all intersecting slice levels with the current window $w$, as shown in Algorithm 8. To this end, it first scans all levels of slices $\gamma_i$ such that their raw level $\gamma^1_i$ overlaps with the window $(\mathrm{P}_\cap(\mathsf{t}(w), \mathsf{t}(\gamma^1_i))$ (lines 2-4). Then it looks for more recent slices, from $\gamma_{i_{\max}+1}$ to $\gamma_{i_{\max}+d-1}$, since at least one of their levels, for each of those recent slices, overlaps with the window (lines 5-7). Figure 9 shows all the internal structures activated at window release when $d=4$. Since the window covers the first three slices, then it first combines all those 12 slice levels. Then it combines with 3 levels of slices 4, 2 levels of slices 5 and 1 level of slice 6. Note that when there TTP is small we might read fewer slices because all slices would not yet be created (line 6). For example, if we release a window of size 3 when we are currently writing slice 5, we would only read internal slice structure of slices 1 to 5 and not 1 to 6 as expected (see Figure 9).

### 5.3. *Time Complexity Analysis*

#### 5.3.1. *Single-Level Slices*

The slicing technique creates one slice per step $\beta$ when $\omega \mod \beta = 0$. Initially the cost per window for adding slices is the number of slices per window, $\omega/\beta$. Nonetheless, because the slices are shared among windows, the cost of adding slices is shared too, one slice being used in $\omega/\beta$ windows. It yields to an amortized cost of 1 per window for slice addition.

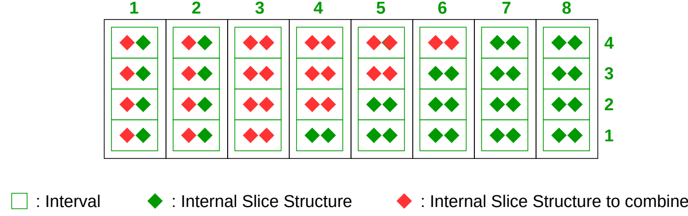



Figure 9: Internal structures to consider when releasing a window of size 3 with a multi-level slice model of height 4. Release starts with the oldest slices shown on the left.

Event insertion in SES has initially a worst-case complexity in $\omega/\beta \cdot N$ because all slices could receive all events. However, we assume that most of the time, event size is smaller than the window size. Hence the event does not need to be inserted into all slices of a window. Therefore we introduce the average event size $\mu_e$ to analyze the number of impacted slices. The complexity becomes then $\lceil \mu_e/\beta \rceil \cdot N$ Again, slice sharing allows to reduce the cost, which becomes $\lceil \mu_e/\omega \rceil \cdot N$ with an upper bound in $N$. The best-case complexity is in $\beta/\omega \cdot N$ when each event is inserted into only one slice. Hence the behavior of events insertion varies depending on the size of the events.

The final aggregate computation has a complexity depending only on window parameters. The cost of window release is the number $\omega/\beta$ of slices per window, i.e., a constant value.

Finally, we can note that, when $\omega \mod \beta > 0$, two slices are created per step and all the above cost values are multiplied by two. Indeed we create twice more slices, for the same event size we would hence insert in twice more slices and we would also double the number of slices that are required for a window release.

*5.3.2. Multi-Level Slices*

The time cost for adding one slice is dependent on the parameter $d$ of the SE-ML-Slicing algorithm. If $\omega \mod \beta = 0$, we create one slice per step, and the cost per step is $d$. As slices are shared among windows, the amortized cost per window still remains $d$.

Let focus on the insertion operation. When $\omega \mod \beta = 0$, and given a height $d$, the size of the highest-level time interval is $d \times \beta$. Hence, an event is inserted in at most $\lceil \mu_e/d\beta \rceil$ slice levels. The cost to insert $N$ events, amortized for a window considering the sharing, is then $\lceil \mu_e/d\omega \rceil \cdot N$. In the best case, the cost is $\beta/\omega \cdot N$ as we insert each event in only one slice. In the worst case, the cost is $1/d \cdot N$ when the event is inserted in every slice. It is worth noticing that the larger the height, the smaller the cost of insertion.

Finally, the cost for releasing a window is the cost for reading all covering slices and more precisely, all the slice levels. This cost can be decomposed into several parts. We read all levels for all slices up to $n_\omega$, and the cost is $d \times (\omega/\beta)$. We then read upper levels for slices from $n_\omega$ to $n_\omega + d - 1$, that yields to a cost of $d \times (d-1)/2$. To sum up, the cost to release a window is $d \times (\omega/\beta + (d-1)/2)$.

When $\omega \mod \beta > 0$ we create twice more slices per step. Hence, we double all previous costs, as expected.

*5.3.3. Summary*

Table 4 gives a summary of all time costs for slicing techniques, with the window cost on top and event cost below. For SE-Slicing and SE-ML-Slicing, window addition and deletion cost correspond to addition and deletion of slices. As it is the most common case, we consider the case where $\omega \mod \beta = 0$.

Table 4 also presents the cost of a baseline streaming approach *Buckets* [13], and a state-of-the-art temporal database approach *Plane Sweeping* [9]. *Buckets* allocates one bucket per window, which can be split into two different methods. Tuple Buckets stores all the events intersecting a window to its associated bucket, in their original form. On the contrary, the Aggregate Buckets method stores the events in a pre-aggregated form for each window. For both tuple-based and aggregate-based methods, the events are therefore processed independently for each bucket of every non-closed window they are in. *Plane Sweeping* uses two endpoint indexes, one for events and one for windows. Those indexes store one entry for each starting and ending bounds. As this technique is fitted for temporal databases, when computing the results it normally contains all events and windows, and indexes are read concurrently, up to their end. In our streaming system, indexes are read partially at window release, and this up to the upper bound of the window. *Plane Sweeping* is shown in its cumulative (Cumul-Sweeping) and selective version (Select-Sweeping).

Table 4: Complexity overview (time cost per window), w.r.t. $N$, the number of events in a window, $\mu_e$ the average size of an event, $\omega$ and $\beta$ the window parameters, and the height $d$ of the slices. The precise cost is used as much as possible and we consider a uniform distribution of events.

| Window cost | add | release | delete |
|---|---|---|---|
| Tuples-Buckets | $1$ | $N$ | $1$ |
| Agg-Buckets | $1$ | $1$ | $1$ |
| Cumul-Sweeping | $2\log((2\delta+\omega)/\beta)$ | $2\beta/\omega \cdot N + 2$ | $2$ |
| Select-Sweeping | $2\log((2\delta+\omega)/\beta)$ | $\log(\delta N/2\omega)\cdot(\beta/\omega \cdot N + 1)$ | $2$ |
| SE-Slicing | $1$ | $\omega/\beta$ | $1$ |
| SE-ML-Slicing | $d$ | $d(\omega/\beta + d - 1)$ | $1$ |

| Event cost | insert | delete |
|---|---|---|
| Tuples-Buckets | $N$ | $0$ |
| Agg-Buckets | $N$ | $0$ |
| Cumul-Sweeping | $\log((2\delta+\beta)/\omega \cdot N)\cdot 2\beta/\omega \cdot N$ | $2\beta/\omega \cdot N$ |
| Select-Sweeping | $\log((2\delta+\beta)/\omega \cdot N)\cdot 2\beta/\omega \cdot N$ | $2\beta/\omega \cdot N$ |
| SE-Slicing | $\lceil \mu_e/\omega \rceil \cdot N$ | $0$ |
| SE-ML-Slicing | $\lceil \mu_e/d\omega \rceil \cdot N$ | $0$ |

Overall SE-Slicing greatly improves insertion cost for events smaller than the range compared to the Buckets methods. At release, SE-Slicing has a better cost

than Tuple Buckets, while having slightly higher cost than Aggregate Buckets. As a result SE-Slicing promises much better insertion cost for windows with large ranges (and in particular when event size is small). Compared to Sweeping, SE-Slicing has a better window addition and release, as long as the number of events is high enough. Event insertion depends on the characteristics of the window and events, for long event size and small window size and step Sweeping has better insertion cost than SE-Slicing. However, as soon as the step is at least greater than half the event size SE-Slicing has a better complexity. We should also note that Sweeping is the only technique not presenting linear deletion cost for events, as the event index should be emptied.

SE-ML-Slicing reduces insertion cost compared to SE-Slicing. But cost for slice creation and window release are slightly increased. Those new costs depend on the *height* of the multi-level slice structure. *Height* value should hence be chosen accordingly with the planned size for events. The larger the height the higher the computation cost at release, but the higher the gain at the insertion for long-standing events. A small height on the contrary gives complexities close to those of SE-Slicing. And a height of 1 would strictly match the SE-Slicing numbers.

### *5.4. Space Complexity*

The space complexity per window is greatly improved by SE-Slicing technique compared to Tuple Buckets which buffers all the events and hence has a space complexity in $\mathcal{O}(N)$, and to the Sweeping technique that keeps every event and window twice in indexes with a space requirement of $(2\delta+\omega)/\beta+(2\delta+\beta)/\omega \cdot N$. In contrast, slicing technique buffers one pre-aggregate only for each slice, then it has a constant space requirement of $\lceil \omega/\beta \rceil$, even reduced to 1 in amortized complexity thanks to slice sharing among many windows. Aggregate Buckets achieve the same complexity, with the same advantage of bounded memory.

SE-ML-Slicing on the other side comes at a higher cost as it stores all slice levels. The cost is $\lceil \omega/\beta d \rceil$ and it falls down to $d$ for a single window.

## 6. Experiments

### *6.1. Experimental Setup*

This series of experiments intends to show the performance improvements with SE-Slicing and SE-ML-Slicing techniques compared to baseline Buckets and state-of-the art Sweeping approaches.

**Data Set.** We used two data sets. Firstly, a *synthetic data set* where each event size is determined by a random number generated with a normal distribution ($\mu$ is given as average events size, $\sigma = 10$). The system creates a non-delayed stream with one event per chronon, totaling 2M events. Next, the *SS7 data set* replays a real-world telephony network with one minute of anonymized data containing 3.2M events. Each event contains 119 fields from which we extract the start and stop times to generate event intervals. In all

experiments, we use $\delta$ to represent the TTP, given as an input parameter of the streaming system.

**Aggregates.** For each window we compute three aggregates: two cumulative functions, namely count and sum, and one selective function, max.

**Setup.** All experiments were executed on an 8-core Intel® Xeon® Silver 4110 CPU @ 2.10 GHz with 126 GB of RAM under Linux Debian 10.

**Implementation.** Implementation has been done in modern C++. Algorithms for the slicing method SE-Slicing are shown in Section 5.2.1 for count and sum. Max uses a similar algorithm, with a non-duplicated slice structure. Algorithms for the multi-level slicing method SE-ML-Slicing are shown in Section 5.2.2. For SE-ML-Slicing we use a circular array to store slices, which allows to define the array size once and then reuse previous slices instead of creating new ones. For the Tuple Buckets techniques, we only store event pointers in the buckets, so that memory overhead is reduced. The Aggregate Buckets technique uses the same partial implementation as SE-Slicing and SE-ML-Slicing. Sweeping technique uses a std::map for indexes and stores event pointers as id of events.

**Metrics.** To monitor the algorithms, we consider three dimensions. The throughput gives the amount of time needed to process a certain number of events. Throughput in these experiments is achieved by letting the program absorb as many events as it can. Then, we monitor CPU time and the maximum memory footprint via `psutil`, a Python cross-platform tool for retrieving information on running processes and system resources. It delivers CPU time spent by a given process, as well as the memory footprint in real time. The memory footprint indicator kept is the maximum memory used per experiment.

**Protocol.** There are three facets to the experiments. First we analyze in Section 6.2 how the different techniques react to a change in event size, and thus stress the insertion part. Then, we study the impact of varying window parameters and in particular increasing the window ranges and the number of slices, in Section 6.3. Finally, we observe the behavior of the algorithms in a more realistic setting thanks to the SS7 data set in Section 6.4.

### *6.2. Impact of Event Size*

Event size is at the heart of spanning events. Thus, maintaining an acceptable throughput with increasing events size is a key challenge for SES streaming systems.

**Single-Level Slices.** As expected from the complexity review in Section 5.3, and as illustrated in Figure 10, large event size results in a decrease of the throughput for all methods, except Sweeping which has a constant throughput no matter the window or event size. For all window sizes, SE-Slicing performs better than Buckets and Sweeping. Furthermore, the smaller the event, the better the gain with SE-Slicing. Increasing window size also results in better performance for SE-Slicing compared to the other techniques, which can be explained by the need to release windows less often. However, for small windows we can see that SE-Slicing starts to be a clear concurrent to SE-Slicing. CPU and maximum memory experiments follow similar tendencies than throughput.

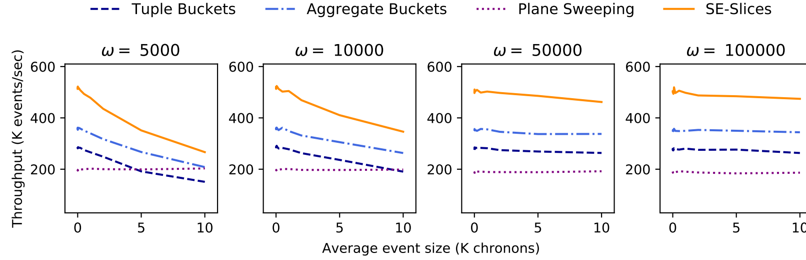



Figure 10: Comparing slices and bucket techniques for varying event size and for different ranges ($\beta = \omega/5$, $\delta = 15000$)

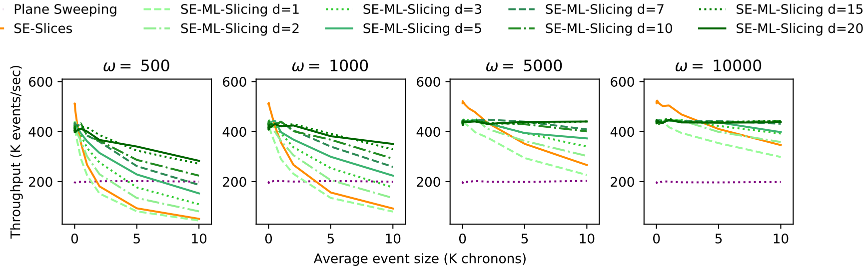



Figure 11: Comparing $d$-slices techniques with raw slices and sweeping for varying event size and for different ranges ($\beta = \omega/5$, $\delta = 15000$)

Regarding CPU, all techniques stay roughly in the same order of magnitude, although SE-Slicing is slightly lower than the others, and with the exception of Tuple Buckets which skyrockets when windows are small and the data stream has long-standing events. Maximum memory used by both SE-Slicing and Aggregate Buckets are quite equivalent, while Tuple Buckets and Sweeping tend to use more memory. Tuple Buckets memory consumption grows with event size, while Sweeping shows a constant consumption in all use cases.

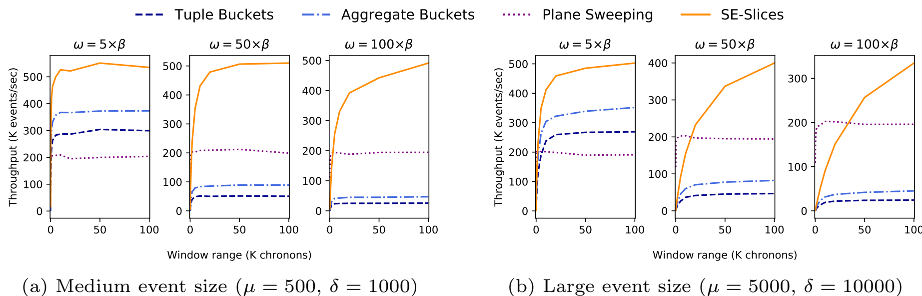
**Multi-Level Slices.** On the complexity analysis we saw that SE-ML-Slicing reduces insertion cost and thus is fitted for use cases where the ratio between event size and window size is large. Thus we study its behavior on smaller windows than previously. As shown on Figure 11, for every height greater than 1, throughput of SE-ML-Slicing outperforms SE-Slicing for all event sizes. When SE-Slicing under-performs compared to Sweeping, SE-ML-Slicing is able to take over and provide better throughput, in particular with large height values. This gain increases with event size, as expected from the complexity review, and validates the motivation for creating this multi-level technique. Furthermore, for this series of experiments the larger the height and the faster the system. For larger window sizes ($\omega = 5000$ and $\omega = 10000$), the system reaches its limit at 550k events/sec. There, heights 15 and 20 have similar results and do not suffer any slow down with increasing event size. Hence we consider that, for those

window sizes, increasing the height would not improve further the performance of the system. Regarding the CPU usage, the higher the height, and the less required CPU. Memory usage in both techniques is roughly the same, except for small windows where SE-Slicing saves more memory. This is due to the high slice management cost of SE-ML-Slicing.

*6.3. Impact of Window Parameters*

With sliding windows, the smaller the step compared to the range and the more the windows overlap with each other. As a consequence, lots of windows are opened at the same time, that yields to many aggregate computations, one per step. Hence, it is interesting to check how the system behaves for increasing window sizes, but also for large overlaps.

**Single-Level Slices.** SE-Slicing shows an increase in performance for all step sizes when $\omega \mod \beta = 0$ (see Figure 12a). In particular, a significant improvement compared to Buckets occurs when the step gets smaller compared to the window size (as long as they are not too close to one), which gives SE-Slicing a definitive advantage with overlapping windows. This also explains the poor performance of both Buckets techniques, which cannot share aggregates, when $\omega = 500 \times \beta$. Sweeping presents the particularity to have a constant throughput.This technique has a clear advantage on SE-Slicing for small window sizes, in particular with long events (see Figure 12b). This setting however would require events longer than the window size, which is very unlikely to happen in real-life use cases. When $\omega \mod \beta > 0$, performance improvement is smaller due to the overhead cost of the two slices per step. But slice techniques still perform better than Buckets, and perform similarly with the Sweeping technique. From this series of experiments, we can conclude that SE-Slicing must be the preferred approach to mitigate overlapping windows in real-life settings, i.e., as long as the ratio between window size and event length stays larger than one.

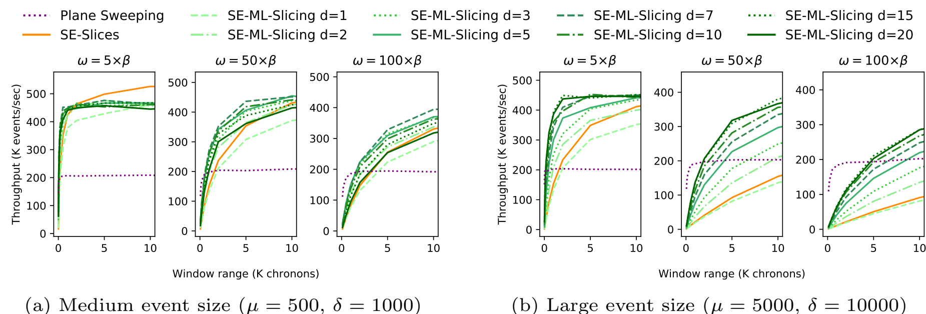



(a) Medium event size ($\mu = 500$, $\delta = 1000$)

(b) Large event size ($\mu = 5000$, $\delta = 10000$)

Figure 12: Throughput depending on the range $\omega$ with varying steps $\beta$ and event size

**Multi-Level Slices.** To address the performance issue of SE-Slicing for small windows and long events, SE-ML-Slicing seems particularly well-suited. We then focus here on the behavior of SE-ML-Slicing in the window range $[1, 10\,000]$ chronons. With an event size of $\mu = 500$, for varying window range and multiple steps, SE-ML-Slicing performs initially worse than SE-Slicing (see




(a) Medium event size ($\mu$ = 500, $\delta$ = 1000)

(b) Large event size ($\mu$ = 5000, $\delta$ = 10000)

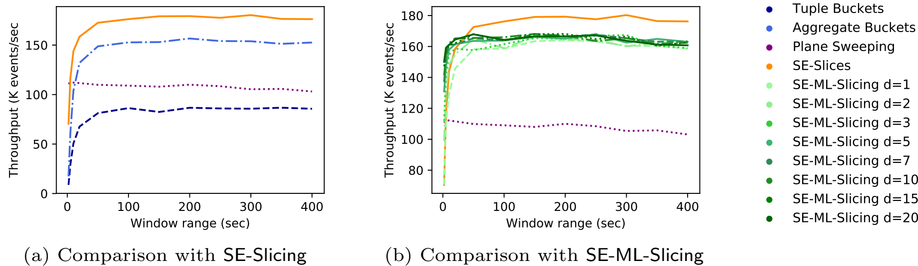
Figure 13: Throughput depending on the range $\omega$ with varying steps $\beta$ and event size

Figure 13a). As soon as $\omega \geq 50 \times \beta$, it performs better than SE-Slicing for all heights greater than 1. This threshold phenomenon is due to the overhead cost of the multi-level structure. For all overlap factors both SE-Slicing and SE-ML-Slicing perform better than Sweeping, with however a slow down for high overlap factors. When $\mu = 5\,000$, SE-ML-Slicing clearly out-performs SE-Slicing for all heights greater than 1 (see Figure 13b). We can note that the larger the window range and the better the improvement with SE-ML-Slicing. Furthermore, increasing the event size results in a huge slowdown for windows with a high overlap factor. It makes the SE-ML-Slicing technique highly beneficial. In this case the competition with Sweeping is tighter, and for overlapping of $\beta \geq 50 \times \omega$ SE-ML-Slicing with large height it is even able to over-perform Sweeping, while SE-Slicing cannot. For high overlap factor, SE-ML-Slicing outperforms Sweeping as soon as the step is at least a hundredth the size of the event, where most of the time it is much larger than that. We can also note that in all conditions, increasing the height $d$ increases the performance of the system, until we reach an optimal height for any given event size. With $\omega = 50 \times \beta$ for instance, this optimal height is 7 when event size $\mu = 500$, and it is 15 when event size is $\mu = 5\,000$. Changing the overlap factor also changes those optimal heights. This motivates the need to appropriately choose the height of the SE-ML-Slicing structure accordingly with the event size and the window parameters.

### *6.4. SS7 Data Set*

Finally, we want to validate the performance of slicing techniques with a data set coming from real-life use cases.

**Single-Level Slices.** With real-life data, and for all window sizes, SE-Slicing performs at least 40% better than Tuple Buckets, 10% better than Aggregate Buckets, and 30% better than Sweeping (see Figure 14a). Once again, CPU and memory consumption of SE-Slicing and Aggregate Buckets are similar, while Tuple Buckets is more resources demanding. In particular Tuple Buckets almost double memory consumption for small windows where events need to be duplicated many times. On the other side SE-Slicing, Aggregate Buckets and Sweeping have a constant memory consumption for almost all window




(a) Comparison with SE-Slicing

(b) Comparison with SE-ML-Slicing

Figure 14: Impact of the range on real data ($\omega/\beta = 5$, $\mu = 16$ seconds, $\delta = 2.5$ hours)

sizes. However, Sweeping is slightly more CPU demanding than SE-Slicing.Those behaviors admit an exception where we observe a peak in CPU and memory consumption for small window size in all techniques except Sweeping, which also results in a decrease of the throughput (see the 0-20 area of window range on both Figures 14a and 14b).

**Multi-Level Slices.** With the SS7 data set, SE-ML-Slicing struggles to outperform SE-Slicing. When the window range is small, SE-ML-Slicing shows better performance, in particular for large heights (see Figure 14b). When increasing window range, the performance of SE-ML-Slicing stays a bit behind the one of SE-Slicing. This behavior can be explained with the data set which contains mainly small events (average event size is 16 seconds), and hence the multi-slice overhead at release time is not compensated by a faster insertion. CPU consumption of SE-ML-Slicing is a bit higher than SE-Slicing, while memory consumption varies between runs, at a level again slightly higher than SE-Slicing. All heights have similar CPU and memory consumption.

### *6.5. Summary*

In summary, there is a significant improvement in using the slicing technique compared to Buckets and Sweeping. Even though SE-Slicing and SE-ML-Slicing outperformed both Buckets techniques in all situations, this is particularly significant for large windows and high-overlapping ratio. Advantage compared to Sweeping follows the inverse rule. As the Sweeping technique presents a constant throughput with all windows and events size, it starts to be competitive when the window range and event size are comparable, particularly for high overlap ratios. The SE-Slicing technique is well-suited for small events, while SE-ML-Slicing, with its specialized structure, performs better for long-standing events. The latter technique, which was meant to decrease insertion complexity, also does the job pretty well for increasing window size. However increasing height comes at a cost that is critical when the ratio between the window range and step increases. Hence, this multi-level technique should be used only when events are large enough, that is to say at least as large as the slices, in order to take advantage of the multi-level structure. Figure 15 highlights the best techniques w.r.t. a 2D parameter space. Each point gives the technique that outperformed

all the others in the experiments for a given setting. The three hand-drawn areas highlight the preferred technique to use in each area. The choice between the two slicing techniques for SES and Sweepingcan thus be motivated by the expected event size and window parameters. It is definitely worth noticing that most of the real-life use cases have a small overlap factor and a high event ratio, hence privileging the SE-Slicing technique.

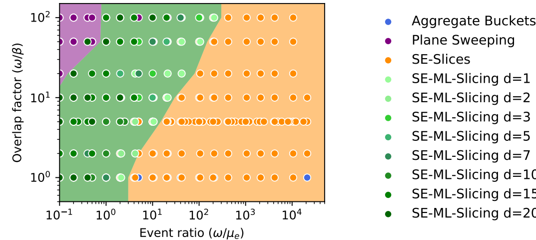



Figure 15: Best-fitted technique depending on event size and window parameters

## 7. Related Work

Many techniques have been proposed to improve the performance of sliding windows on PES systems: *buffers*, *buckets*, *aggregate trees*, *stacks*, *slices*, and their compositions [13]. Baseline techniques keep all the events and compute the aggregate at release time: *buffers* just do that, whereas *buckets* [15] distribute events into subsets (e.g., one per window). *Buffer* technique can deal with out-of-order events by reordering them before processing [16]. However, to be adapted to SES, it would require a continuous reordering as well as costly deletions of past events, which incurs a low throughput. *Buckets* are especially used for out-of-order processing [17]. As they keep events separately for each currently opened windows, *buckets* techniques can be easily adapted to deal with spanning events, and provide a baseline for streaming technique as used in this paper. With overlapping windows, both *buffer* and *buckets* methods lead to redundancy in computation as well as to spikes in the system when aggregates are released. The *buckets* technique, however, comes in two flavors: Tuple Buckets keeps all the events in their original form, while Aggregate Buckets [13] keeps only one partial aggregate per window, thus reducing release spikes and providing a better throughput as shown in the experiments. *Aggregate trees* such as FlatFAT [3] and FiBA [16] store partial aggregates on top of events in a hierarchical data structure. *Stack* such as TwoStacks [18] and DABA [18] store events in a stack requiring in-order events. Both *aggregate trees* and *stack* structures use the timestamp of events to order the structure and assign events to windows. This can be easily achieved with time point storage, but is not fitted to spanning events as intervals start and end bound follow different ordering. Finally, *slices* technique can be further improved with final aggregation techniques, which define how to merge slice sub-aggregates. Instead of iterating over all the slices,

those techniques use aggregate trees or indexes, e.g., B-Int [4], FlatFIT [19], and SlickDeque [8]. This allows to mitigate the bottlenecks when computing final results at window release. As they continuously update all aggregates, insertion of spanning events would yield to large re-updates of the structure, and the duplication in slices still needs to be handled.

Other techniques from the temporal databases area can also be cited. In temporal databases, few work has been done regarding sliding windows, which is denoted as fixed intervals queries [9]. Currently the state-of-the-art technique is proposed by Piatov and Helmer with a *plane sweeping* algorithm [9]. This technique keeps both starting and ending bounds of events and windows in separate indexes which are then read concurrently while filling global counters on events started and ended. This technique can be extended to streaming systems as shown in this paper, providing good results only when events are longer than windows. SB-trees [20] store hierarchy of intervals along with partial aggregates. However, the B-tree structure is hard to maintain in a streaming system dealing with high ingestion rate. Indeed, after each insertion, rebalancing is run to keep the structure as packed as possible. Furthermore, this technique requires a separate tree for each aggregation function, increasing drastically its cost in real-life use cases with multiple aggregates at a time. Finally, deletion is handled on an event basis, which requires to buffer events and then, cannot be used with selective functions. TMDA-FI [21] proposes a different approach where a table is built from window ranges and their partial aggregate values. Those values are computed incrementally as the input relation is scanned. When applied to streaming systems, this technique is similar to the Aggregate Buckets one.

## 8. Conclusion

This article addresses the issue of partial aggregate sharing among overlapping windows to *spanning event* streams (SES for short). Dealing with spanning events brings new constraints, since events overlap the ongoing window as much as past windows. Concerning slicing techniques, SES aggregation implies that adjacent slices may be assigned the very same events. Hence, operations sensitive to duplication would provide inaccurate results, and common slicing techniques cannot be used straightforwardly.

Therefore, we extended the slice model and algorithms according to properties of the aggregate functions. When functions are insensitive to event duplicates, we can reuse the model and workflow of PES, with a slight difference, however: at event insertion, we update all the intersecting slices instead of only the last one. When functions do have this sensitivity, we duplicate the structure to separate events that ends in the slice from the ones that continue afterwards.

We then optimized further this technique with the goal to reduce event insertion cost. Indeed, the duration of spanning events implies multiple insertion in slices, which could be reduced with the use of larger slices. To this end, we created a multi-level slice model which allows the insertion of events in larger intervals than the ones from the raw slices. This multi-level model has a similar

behavior than the raw slice model, with the addition of a specific path to insert events.

As expected from complexity analysis, slicing techniques with spanning events are computationally more costly than with point events, but they stay, on average, lower than the buckets techniques, and in particular the tuple buckets one. More precisely, experiments show that the use of slices with spanning event results in significant improvements in throughput, in all use cases with varying events and window size as much as varying overlapping ratio. For long-standing events, slicing techniques can be further improved by the use of multi-level slices. Multi-level slices have higher or equivalent results compared to raw slices, depending on the ratio between slices and event size. Furthermore, the height of the multi-level slices should be chosen accordingly with the event size, range and step to be sure not to underperform. However, when the size of the window is too small compared to event length or too large compared to its step the slicing techniques become less efficient than plane sweeping from the temporal database area. Those settings are, however, very unlikely to happen in real life conditions.

To conclude, our raw slice model is suitable for all techniques which purpose is to partially aggregate spanning events. It can easily be implemented in more complex structures, such like multi-level slices, and could be used in final aggregation optimization techniques. Eventually, a study of the impact of out-of-order events would be of great interest as a follow-on this line of research.

## References


[1] M. H. Böhlen, A. Dignös, J. Gamper, C. S. Jensen, Temporal Data Management : An Overview, in: eBISS'17, Vol. 324, 2017, pp. 51–83.

[2] J. F. Allen, Maintaining knowledge about temporal intervals, Communications of the ACM 26 (11) (1983) 832–843.

[3] K. Tangwongsan, M. Hirzel, S. Schneider, K.-L. Wu, General incremental sliding-window aggregation, PVLDB'15 8 (7) (2015) 702–713.

[4] A. Arasu, J. Widom, Resource Sharing in Continuous Sliding-Window Aggregates, VLDB'04 30 (2004) 336–347.

[5] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, H. Pirahesh, Data Cube : A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals, Data Mining and Knowledge Discovery 1 (1) (1997) 29–53.

[6] H. G. Kim, M. H. Kim, A review of window query processing for data streams, Journal of Computing Science and Engineering 7 (4) (2013) 220–230.

[7] S. Krishnamurthy, M. J. Franklin, J. Davis, D. Farina, P. Golovko, A. Li, N. Thombre, Continuous analytics over discontinuous streams, in: SIGMOD'10, 2010, pp. 1081–1092.

[8] A. U. Shein, P. K. Chrysanthis, A. Labrinidis, SlickDeque: High Throughput and Low Latency Incremental Sliding-Window Aggregation., EDBT'18 (2018) 397–408.

[9] D. Piatov, S. Helmer, Sweeping-based temporal aggregation, SSTD'17: Advances in Spatial and Temporal Databases LNCS 10411 (2017) 125–144.

[10] P. Carbone, J. Traub, A. Katsifodimos, S. Haridi, V. Markl, Cutty: Aggregate Sharing for User-Defined Windows, in: CIKM'16, 2016, pp. 1201–1210.

[11] J. Li, D. Maier, K. Tufte, V. Papadimos, P. A. Tucker, No Pane, No Gain: Efficient Evaluation of Sliding-Window Aggregates over Data Streams, ACM SIGMOD Record 34 (1) (2005) 39–44.

[12] J. Traub, P. M. Grulich, A. Rodriguez Cuellar, S. Bress, A. Katsifodimos, T. Rabl, V. Markl, Scotty: Efficient Window Aggregation for Out-of-Order Stream Processing, in: ICDE'18, 2018, pp. 1300–1303.

[13] J. Traub, P. Grulich, A. R. Cuéllar, S. Breß, A. Katsifodimos, T. Rabl, V. Markl, Efficient Window Aggregation with General Stream Slicing, in: EDBT'19, 2019, pp. 97–108.

[14] S. Krishnamurthy, C. Wu, M. Franklin, On-the-fly sharing for streamed aggregation, in: SIGMOD'06, 2006, pp. 623–634.

[15] J. Li, K. Tufte, D. Maier, V. Papadimos, AdaptWID: An Adaptive, Memory-Efficient Window Aggregation Implementation, IEEE Internet Computing 12 (6) (2008) 22–29.

[16] K. Tangwongsan, M. Hirzel, S. Schneider, Optimal and general out-of-order sliding-window aggregation, PVLDB'19 12 (10) (2019) 1167–1180.

[17] J. Li, K. Tufte, V. Shkapenyuk, V. Papadimos, T. Johnson, D. Maier, Out-of-order processing: a new architecture for high-performance stream systems, PVLDB'08 1 (1) (2008) 274–288.

[18] K. Tangwongsan, M. Hirzel, S. Schneider, Low-Latency Sliding-Window Aggregation in Worst-Case Constant Time, DEBS'17 (2017) 66–77.

[19] A. U. Shein, P. K. Chrysanthis, A. Labrinidis, FlatFIT: Accelerated incremental sliding-window aggregation for real-time analytics, SSDBM'17 (2017) 1–12.

[20] J. Yang, J. Widom, Incremental computation and maintenance of temporal aggregates, The VLDB Journal 12 (3) (2003) 262–283.

[21] M. H. Böhlen, J. Gamper, C. S. Jensen, Multi-dimensional Aggregation for Temporal Data, in: EDBT'06, 2006, pp. 257–275.

**Declaration of interests**

☑ The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

☐The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: