# Chill: Runtime Reconfiguration of Windowed Stream Aggregates with Semantic Guarantees

Jingyu Liu
Chalmers University of Technology
and University of Gothenburg
Gothenburg, Sweden
jingyu.liu@chalmers.se

Vincenzo Gulisano
Chalmers University of Technology
and University of Gothenburg
Gothenburg, Sweden
vincenzo.gulisano@chalmers.se

## Abstract

Stream Aggregates are a key building block in edge-to-cloud data pipelines used to summarize data over *windows*, time intervals defined by their *advance* (the frequency of output) and *size* (the interval length). In heterogeneous deployments such as smart grids and vehicular networks, runtime changes in workload and resources require practitioners to rebalance freshness, cost, and semantic guarantees.

A limitation of existing solutions is that, once the window advance and size are chosen, they stay fixed for the entire execution. To overcome this limitation, Chill enables dynamic adjustment of both parameters. Analysts can specify a set of acceptable advances and sizes, selecting them at runtime to trade performance and guarantees through weaker (at-most-once) or stronger (exactly-once) reconfiguration semantics. We present alternatives for both timestamp-ordered and out-of-order execution models. We also empirically show that Chill can match the performance of fixed-configuration baselines while supporting reconfigurations in negligible time.

## CCS Concepts

• **Information systems** → **Data management systems**; • **Theory of computation** → **Streaming models**.

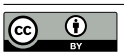
## Keywords

Stream Processing, Stream Aggregates, Semantic Equivalence

**ACM Reference Format:**
Jingyu Liu and Vincenzo Gulisano. 2026. Chill: Runtime Reconfiguration of Windowed Stream Aggregates with Semantic Guarantees. In *The 20th ACM International Conference on Distributed and Event-based Systems (DEBS '26), June 23–26, 2026, Lisbon, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3809481.3812621

## 1 Introduction

Stream Aggregates (or simply Aggregates) are a key building block in edge-to-cloud data pipelines, where data is continuously sensed (e.g., in smart grids [19] and vehicular networks [16, 20]) and transformed into insights. Executed by Stream Processing Engines (SPEs) [5, 9, 11], Aggregates can be deployed at different points in the pipeline, depending on the Aggregates' computational cost and the resources available at each node. They summarize data over *windows* and periodically produce results that enable timely analysis.




*DEBS '26, Lisbon, Portugal*

ACM ISBN 979-8-4007-2693-4/2026/06
https://doi.org/10.1145/3809481.3812621

In practical edge-to-cloud deployments, both workload and available resources vary over time. Practitioners therefore need to continuously rebalance output freshness, computational cost, and semantic guarantees, rather than committing to a single static aggregation configuration.

*Motivation.* Aggregates' computational cost depends on both summarization parameters (e.g., window span, aggregation function) and time-evolving data characteristics (e.g., volume, distribution). Many approaches have been proposed to reduce such costs – distribution [1], parallelization [13], wavelets [8], sketches [18], sampling [6], compression [17], or shedding [22] – often combined depending on whether CPU or memory is the bottleneck. Still, these solutions face challenges in edge-to-cloud settings: scaling out may be infeasible at the edge, and approximation may sacrifice accuracy in ways analysts cannot always control.

Notably, existing solutions typically assume a fixed window configuration at runtime (i.e., a window with given advance and size; see § 2), and they steer costs either by manipulating the input tuples (e.g., discarding them through load shedding [22]) or by modifying the aggregation state (e.g., using approximate structures such as sketches [18]). We propose a different approach: using the window configuration itself as a runtime control knob. More precisely, we ask: *Can computational costs be kept under control by letting analysts specify a family of acceptable window configurations (with varying computational demands), and by efficiently switching among them with well-defined guarantees?*

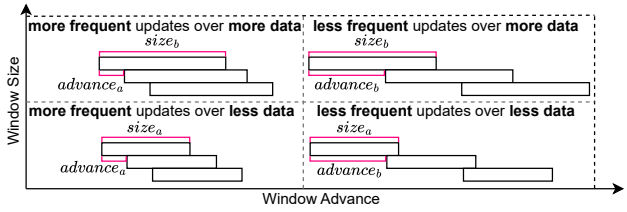



**Figure 1: Computation cost under different advances and sizes.**

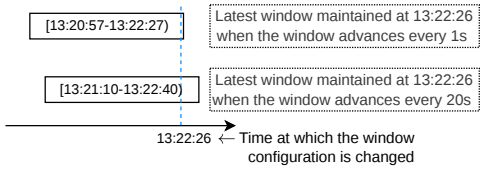
*Contributions.* We make the following contributions with a focus on practical deployment. By building on pane-based aggregation [4] (see § 2), where non-overlapping window segments are maintained independently and merged when producing outputs, we design algorithms that allow analysts to define Aggregates across multiple window advance and size configurations, and to switch efficiently among them at runtime. A larger advance yields less frequent and thus less costly outputs, while a smaller size summarizes data over shorter intervals at a lower cost, as shown in Figure 1.

As mentioned in our example, a configuration change might imply that the previously maintained windows no longer align with

the new configuration. We show this observation enables further trade-offs between guarantees and performance upon configuration changes, and introduce strategies that either guarantee *at-most-once* semantics, where tuple contributions are counted at most once, or stronger *exactly-once* semantics, where every tuple is always processed with respect to the correct window configuration.

**Example - Part 1**

Consider an analyst counting per-vehicle reports in a road traffic monitoring application. She is interested in two reporting setups: per-vehicle reports every second over the last minute and a half, or reports every 20 seconds over the same last minute and a half. Depending on whether finer- or coarser-grained results are needed, she may switch between these two configurations at runtime. With Chill, she can define an Aggregate supporting two window advances (1 s and 20 s) and a window size of 90 seconds, and switch between these configurations at runtime. As illustrated below, such a switch might cause misalignment between previously maintained windows and the newly selected configuration, since their boundaries no longer coincide. For instance, at 13:22:26, a window under a 20 s advance would span 13:21:10-13:22:40, whereas switching to a 1 s advance implies a window spanning 13:20:57-13:22:27.




We also implement prototypes tailored for Aggregates that are fed only timestamp-sorted input streams (e.g., data arriving in increasing time order), as well as Aggregates that support the processing of out-of-order streams. Using common benchmark data, synthetic stress workloads, and state-of-the-art SPEs like Apache Flink [9] (or simply Flink in the remainder), we show our algorithms enable reconfiguration with negligible overhead while sustaining performance comparable to fixed-configuration baselines. We also evaluate how enlarging the acceptable window configuration space affects performance, further showing the trade-offs between at-most-once and exactly-once semantics. All our code is publicly available at https://doi.org/10.5281/zenodo.20280981.

*Organization.* § 2 covers preliminaries. § 3 introduces our system model and problem definition. § 4 details our algorithms. § 5 covers the empirical assessment of our approach. § 6 reviews related work. § 7 concludes and suggests future directions. All symbols/abbreviations are found in Appendix A.

## 2 Preliminaries

A *stream* $S$ is an unbounded sequence of *tuples* [2]. Each tuple $t$ carries a timestamp $t.\tau$ and a payload $t.\phi$. Stream processing *queries* are Directed Acyclic Graphs (DAGs) composed of *sources*, *operators*, and *sinks*. Operators are either *stateless* – they do not maintain a state that evolves with the tuples they process – or *stateful*. For a source tuple $t$, $t.\tau$ is the *event time* set by the source when the event occurred, expressed in time units from a given epoch that progress in SPE-specific $\delta$ increments (e.g., milliseconds [9]).

When a query is deployed within an SPE, multiple copies of each operator can be instantiated to analyze different portions of its input stream(s). Commonly, such instances run in shared-nothing environments [9, 11] where each instance has a dedicated state exclusively managed by one thread. The evolution of this state – namely, the raw or aggregated tuples it contains, how they are stored in the thread's data structures, and the output tuples generated from such a state – depends solely on the tuples processed and produced by that thread.

A stateful *Aggregate* $A$ operates on *time-based windows* (or simply windows) defined by:

**Window Advance ($wa$) and Size ($ws$):** Defining the event-time intervals $[mwa, mwa+ws)$, with $m \in \mathbb{N}$, $wa > 0$, and $ws > 0$, covered by $A$ (right boundary exclusive). Each interval is a window *instance* $\gamma$; $\gamma.l$ and $\gamma.r$ denote its left and right boundaries, respectively. *Tumbling* windows occur when $wa = ws$ (a tuple falls into exactly one instance). *Sliding* windows occur when $wa < ws$ (a tuple may fall into multiple instances).

**Key-by Function $f_K(t)$:** Extracting a key from tuple $t$ and maintaining dedicated $\gamma$s for tuples sharing the same key. Although optional in SPEs [9, 11], from a model perspective, $f_K$ can always be defined, returning the same value for all tuples unless specified otherwise.

From an operational perspective, $wa$ controls output frequency and thus computational cost, while $ws$ controls the amount of history summarized and thus state footprint.

As aforementioned, when deployed, a user-defined Aggregate is typically instantiated as multiple Aggregate instances that can execute in parallel and across distributed nodes. In shared-nothing environments [9, 11], each instance owns dedicated local memory/state and processes a portion of the key space of the input stream.

*Execution Strategies and API.* $A$ keeps per-$\gamma$ state based on its execution strategy [10], where single-window, multi-window, and pane-based (also referred to as slicing [23]) are among the main strategies discussed in the literature.

*(i) Single-window.* $A$ keeps a single accumulator per active $\gamma$ and key and defines methods `add(t)`, `get()`, and `evict(t)`: `add` updates the accumulator for $\gamma$, `get` returns its current result, and `evict` removes $t$ from $\gamma$ once $t$ no longer falls within $\gamma$ after $\gamma$'s shift forward.

*(ii) Multi-window.* $A$ keeps one accumulator per *overlapping* $\gamma$ and key. Upon reception of $t$, `add(t)` is invoked on all overlapping $\gamma$s into which $t$ falls. Method `get()` reads each $\gamma$'s result. Each $\gamma$ is discarded immediately after its result is produced.

*(iii) Pane-based (the approach we use).* Panes, whose instances we denote as $p$s, are consecutive, non-overlapping intervals that may vary in length. Each $\gamma$ spans a set of consecutive panes, the first starting at $\gamma.l$ and the last ending at $\gamma.r$. By definition, $p^e.r = p^{e+1}.l$. Each tuple $t$ falls into a unique pane. $A$ maintains one partial aggregate per $p$ and exposes `add(t)`, to update $p$, and `get()`, to retrieve $p$'s partial aggregates. When producing outputs, `merge()` merges the partial aggregates of all panes covering the same window to compute the final output, as shown in Figure 2 in connection with the example that follows.

*Correctness.* In state-of-the-art SPEs [5, 9], when an output tuple $t_o$ is produced for a window instance $\gamma$, $A$ sets $t_o.\tau = \gamma.r - \delta$ (the maximum event time within $\gamma$). Hence, output event times take values of the form $nwa + ws - \delta$ (for integer $n$). If $A$ processes streams where event time advances at the same pace as wall-clock time, $A$ emits

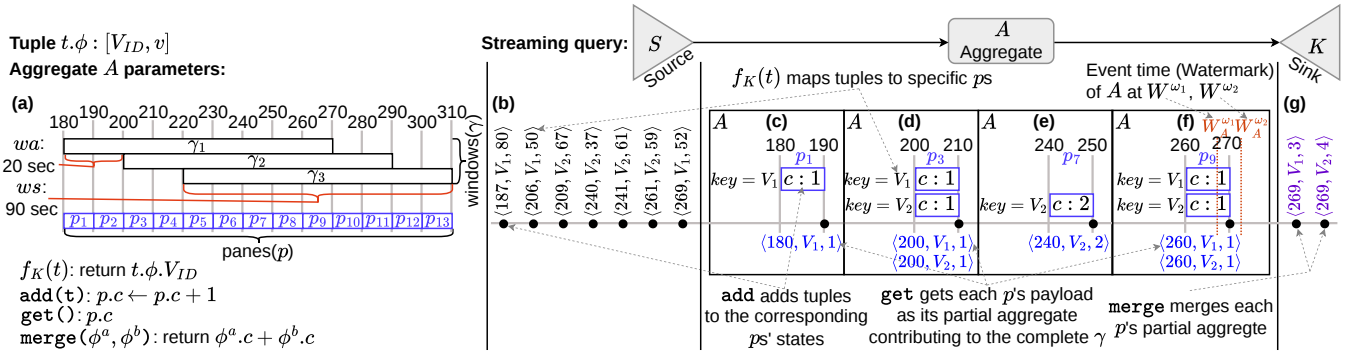



**Figure 2: Sample $A$ building on Example 1 that counts per-vehicle reports over a window with $wa$ and $ws$ of 20s and 90s, respectively. The execution excerpt shows how $A$'s functions process input tuples, maintain $A$'s $p$s, and produce output tuples by merging the partial aggregates of all $p$s within the same $\gamma$.**

one output every approximately $wa$ time units (in both event and wall-clock time).

To decide when the correct result for a given $\gamma$ can be produced, accounting for all $\gamma$'s tuples, an SPE must know that no more tuples falling into $\gamma$ are still to be processed. This can be ensured in two ways. If the SPE assumes timestamp-sorted streams [11, 14], correctness of the output of a $\gamma$ is guaranteed as soon as a tuple $t$ with $t.\tau \geq \gamma.r$ is observed. Alternatively, if the SPE supports out-of-order streams, it can rely on *watermarks* [9, 15], special tuples that signal event-time progress from each source's perspective. $W^{\omega}_A$, the watermark of $A$ at wall-clock time $\omega$, is the earliest event time any future tuple $t$ fed to $A$ can have from $\omega$ onward. Sources periodically emit watermarks as special tuples with monotonically increasing timestamps [9, 15], indicating event-time progress from each source's perspective. Upon receiving a watermark $W$, $A$ records it and updates $W^{\omega}_A$ to the minimum of the latest watermarks observed on all its input streams. When $W^{\omega}_A$ increases, $A$ outputs all windows whose right boundary is not greater than $W^{\omega}_A$ and forwards the updated $W^{\omega}_A$ downstream.

We distinguish the two cases by saying that $A$'s correctness is supported through a timestamp-based (TB) or a watermark-based (WB) implementation, respectively.

**Example - Part 2**

Building on the running example from Example-Part2, which we also include in our evaluation in § 5 with a heavier aggregation function, Figure 2 shows how a pane-based Aggregate $A$ counts, for each vehicle ID - $V_{ID}$, how many reports fall within each window when $wa$=20s and $ws$=90s. According to the pane-based execution strategy (see § 2), each $\gamma$ spans 9 10-second $p$s, as shown in Figure 2(a). Each tuple belongs to one pane only, whose local state is maintained through the `add(t)` method, while `get()`/`merge()` methods are invoked to combine the 9 pane partial aggregates into a $\gamma$'s output. In the example, $A$ relies on watermarks to determine when to output each $\gamma$. For clarity, timestamps are shown using only their seconds component; thus, the first window is depicted as covering the interval [180,270). In Figure 2(f), watermark $W^{\omega_1}_A$ approaches 270, the right boundary of the $\gamma$ whose output is to be returned, making tuples contributing to the $\gamma$ spanning [180,270) ready for output. Once the watermark $W^{\omega_2}_A$ exceeds 270, $A$ determines that no further tuples with timestamps earlier than 270 will arrive. At this point, $A$ invokes `merge()` on the partial results from 9 panes, producing the result for the $\gamma$ spanning [180,270). Upon returning the output, $\gamma$ advances by $wa$, and the corresponding pane state is updated to process subsequent tuples.

## 3 Problem Formalization

As mentioned in § 1, and illustrated in Figure 1, the frequency and cost of producing outputs are inversely proportional to $wa$, while state size and aggregation cost are proportional to $ws$. Chill aims to define an operator, $A^*$, that is parametrized by a set $\mathbb{WA}$ of acceptable $wa$ values and a set $\mathbb{WS}$ of acceptable $ws$ values. $A^*$ allows an external entity (e.g., an analyst, a practitioner, or an automated controller) to change its configuration from $(wa,ws)^i$ to $(wa,ws)^{i+1} \in \mathbb{WA} \times \mathbb{WS}$ at runtime to react to changing workload and resource conditions. When $wa$ and $ws$ change, the maintained state may no longer align with the new configuration, as illustrated in Example-Part 1. To handle this and provide explicit correctness/performance trade-offs during reconfigurations, $A^*$ can enforce weaker semantics (at-most-once, AMO) or stronger ones (exactly-once, EO). A valid solution for our problem builds on the following assumptions and must satisfy the requirements below.

*Assumptions.* As for $A$, $A^*$'s state and outputs evolution depend only on the input tuples (and watermarks, if any) it receives asynchronously from upstream sources or operators. In TB implementations, $A^*$ defines methods `process(`$t$`)` (to process $t$), `output(`$t$`)` (to output $t$), and `change((`$wa$,$ws$`))` (to trigger a configuration change). In WB implementations, it additionally defines `process(`$W$`)`, invoked whenever $A^*$'s watermark grows to a new value $W$. All methods are invoked sequentially by the thread running $A^*$. As discussed in § 2, this per-thread design applies to each thread running an Aggregate instance and therefore extends transparently to parallel and distributed executions over key-partitioned streams.

*Requirements.* We define an *epoch* as a contiguous interval of event time during which a specific $(wa,ws)$ pair is used by $A^*$.

- The first epoch starts at deployment and lasts until the (exclusive) event time scheduled at the first invocation of `change`. The second epoch begins at the latter event time (inclusive) and lasts until the time scheduled at the second `change` invocation, and so on.
- A tuple whose event time falls within an epoch must be processed using the $(wa,ws)$ of that epoch. An output with a timestamp in that epoch must reflect the same configuration.
- Tuples with the same timestamp must belong to exactly one epoch.

*Performance metrics.* When assessing $A^*$'s performance, we consider the following metrics, which capture operational efficiency and correctness behavior:

- **Throughput** the number of tuples per second ($t/s$) it sustains.
- **Latency** the per-second average wall-clock time between the ingestion of an input tuple $t_i$ triggering the production of an output tuple $t_o$ and the emission of $t_o$.
- **CPU** the per-second portion of processor time consumed during execution.
- **Panes** the per-second memory footprint given by the total number of panes it maintains.

These metrics compare $A^*$ against other baselines that support only one window configuration. When reconfigurations are performed, we also assess:

- **Reconfiguration time (RC)** the time needed to transition from $(wa,ws)^i$ to $(wa,ws)^{i+1}$.
- **Convergence time (CV)** time, measured from the event time at which a new epoch starts, until AMO resumes producing correct outputs under $(wa,ws)^{i+1}$ after the transitional period in which the non-aligning state may result in approximate outputs.

*Scope.* Regarding the scope of our contribution, please note:

- We do not address the quality of decisions made by the entity triggering reconfigurations. Our focus is solely on the algorithms and implementation supporting efficient reconfiguration.
- For notational ease, we assume that all processed tuples satisfy $t.\tau \geq ws$, so every $\gamma$ lies entirely after time 0, and that $\delta = 1$ (see § 2), as in state-of-the-art SPEs [9].

## 4 Algorithms

As mentioned in § 1, a key observation for dynamic window reconfiguration is that once $wa$ and $ws$ change, the state maintained by $A^*$ may no longer align with the new configuration. In operational settings, where reconfiguration requests can be triggered at runtime by analysts or controllers, minimizing reconfiguration time and maximizing state reuse is crucial. Maintaining state at the granularity of entire $\gamma$s provides only coarse control, whereas a pane-based strategy (see § 2) decomposes $\gamma$s into smaller "tiles" that can be efficiently reused across configurations. Because of this, $A^*$ adopts a pane-based execution strategy.

In this section, we first describe how, for each tuple, we determine the $\gamma$s it contributes to and, more importantly, the $p$ where it is stored under the chosen semantics. This tuple-to-$\gamma$/$p$ assignment is a core building block for all our solutions. We then provide an overview of the proposed solutions for AMO and EO semantics. Finally, we detail the algorithms: one general approach for TB and one for WB implementations, with only a subset of methods specialized for AMO and EO semantics. For AMO semantics, we also quantify the event-time duration after a reconfiguration during which results might differ from those of an approach where state aligns in accordance with the current epoch at all times, as in EO semantics.

*Determining a tuple's $\gamma$s and semantics-dependent $p$.* For event time $\tau$, we denote by $\Omega(\tau,(wa,ws))$ (or simply $\Omega$, when notation can be simplified) the number of $\gamma$s of advance $wa$ and size $ws$ to which a tuple $t$ so that $t.\tau = \tau$ contributes. It holds that:

$$\Omega = \begin{cases} \left\lceil \frac{ws}{wa} \right\rceil, & \text{if } (\tau \bmod wa) < (ws \bmod wa) \vee (ws \bmod wa) = 0, \\ \left\lceil \frac{ws}{wa} \right\rceil - 1, & \text{otherwise.} \end{cases} \tag{1}$$

Let $\gamma^i(\tau,(wa,ws))$ (or simply $\gamma^i$) be the $i$-th such $\gamma$, for $i \in [0,\Omega-1]$. Then:

$$\gamma^i.l = \left( \left\lfloor \frac{\tau}{wa} \right\rfloor - (\Omega-1) + i \right) wa \tag{2}$$

Let $p$ be the pane to which $t$ contributes. Since panes must align with $\gamma$ boundaries for correct aggregation, a pane starts whenever a $\gamma$ starts or ends. If $\tau_1 < \tau_2 < \ldots \leq \tau$ are all event times at which a $\gamma$ starts or ends at or before $\tau$, then $p.l = \max(\tau_1,\tau_2,\ldots)$, i.e., the latest $\gamma.l$ or $\gamma.r$ not exceeding $\tau$. Hence, $p.l$ is the greater of: (1) the left boundary of the latest $\gamma$ to which $t$ falls ($\gamma^{\Omega-1}.l$), and (2) the right boundary of the $\gamma$ immediately preceding the earliest $\gamma$ to which $t$ falls (denoted in this case as $\gamma^{-1}.r$). Thus:

$$\begin{aligned} p.l &= \max\left(\gamma^{\Omega-1}.l,\ \gamma^{-1}.r\right) \\ &= \max\left(\left\lfloor \frac{\tau}{wa} \right\rfloor wa,\ \left(\left\lfloor \frac{\tau}{wa} \right\rfloor - \Omega\right) wa + ws\right) \end{aligned} \tag{3}$$

In the following, `get_`$\gamma$`.l`$(\tau,(wa,ws))$ and `get_`$p$`.l`$(\tau,(wa,ws))$ denote the methods that compute the left boundary of a $\gamma$ or a $p$, respectively, based on Eq. 2 and Eq. 3.

*At-most-once (AMO)/exactly-once (EO) semantics and window configuration changes.* According to our problem definition (see § 3), practitioners can control performance/guarantee trade-offs not only by changing $wa$ and $ws$, but also by choosing between AMO and EO semantics. Maintaining only the $p$s required by the current – and, upon each change, retaining only those that fall within the newly selected $(wa,ws)$ parameters – introduces a finite interval during which results may diverge from those of $\gamma$s that would cover all past tuples. Nevertheless, this approach satisfies AMO semantics: each tuple is incorporated into at most one pane, either one created before the change or one created after it. It violates our epoch requirements, though (see later). In contrast, EO semantics guarantee exact results if, at all times, $p$s are maintained and updated according to all acceptable configurations and if such panes can cover the full temporal extent of the largest window size in $\mathbb{WS}$ (since a change to such a $ws$ can happen at any point during the execution of $A^*$). Our algorithms build directly on these observations. For AMO, we maintain $p$s only for the current epoch's $(wa,ws)$. When `change` is invoked, the operator schedules the new configuration and, once the change becomes effective, keeps only the previous state (i.e., $p$s) that falls entirely in $\gamma$s defined by the new configuration. For EO, we continuously maintain $p$s that can tile $\gamma$s for every $(wa,ws) \in \mathbb{WA} \times \mathbb{WS}$, and produce outputs by merging the partial aggregates of those $p$s matching the current epoch's configuration.

While the $\gamma$ to which a tuple belongs depends only on the epoch's $wa$ and $ws$, the specific $p$ varies with the chosen semantics. For AMO semantics, `get_`$p$`.l` suffices; in our algorithms, this is encapsulated in `get_AMO_`$p$`.l`. For EO semantics, the $p$ is the intersection of all panes across configurations, and $p.l$ is computed as:

$$p.l = \max_{(wa,ws) \in \mathbb{WA} \times \mathbb{WS}} p^{(wa,ws)}.l \tag{4}$$

In our algorithms, this logic is implemented by the method `get_EO_`$p$`.l`.

Since Eq. 4 yields panes as short as or shorter than those from any individual configuration, EO semantics can lead to maintaining more shorter panes and to merging more panes per output than AMO semantics. More precisely, for a current epoch with configuration $(wa,ws)$, AMO maintains panes only for that configuration, so

both the number of panes kept in memory (per key) and the panes merged per output scale are $\Theta(ws/wa)$. Under EO, panes follow the common refinement across all configurations in $\mathbb{WA}\times\mathbb{WS}$; in the worst case, over an interval of length $L$, the number of panes is $O\left(L\sum_{(wa',ws')\in\mathbb{WA}\times\mathbb{WS}}\frac{1}{wa'}\right)\subseteq O\left(L\frac{|\mathbb{WA}\times\mathbb{WS}|}{\min(\mathbb{WA})}\right)$. Hence, EO requires up to $O\left(\max(\mathbb{WS})\frac{|\mathbb{WA}\times\mathbb{WS}|}{\min(\mathbb{WA})}\right)$ panes in memory (per key) and up to $O\left(ws\frac{|\mathbb{WA}\times\mathbb{WS}|}{\min(\mathbb{WA})}\right)$ panes merged per output for the current epoch. These are analytical worst-case bounds; in practice, as we also show in § 5, carefully choosing the configuration sets can minimize the gap. More precisely, if the added $wa$ values are multiples of $\min(\mathbb{WA})$, the pane partition does not become finer, which helps keep the overhead of EO closer to that of AMO.

*Detailed design.* To meet our requirements (see § 3), our algorithms must ensure that, when a change is applied, it is used for all tuples from a specific event time onward, creating a clear cut between $(wa,ws)^i$ and $(wa,ws)^{i+1}$.

Immediately adopting $(wa,ws)^{i+1}$ when change is invoked is not valid. In a TB implementation, the change could occur while processing tuples with the same timestamp, breaking the "same timestamp, same epoch" requirement. In WB implementations, even if a change occurs between tuples $t_a$ and $t_b$ with $t_a.\tau < t_b.\tau$, a $t_c$ with $t_b.\tau \le t_c.\tau$ could already have been processed, again breaking the cut. When change is invoked, $A^*$ therefore tracks $\tau_{max}$, the highest tuple timestamp seen, and sets $\tau^*$, the new epoch's start time, to $\tau_{max}+1$. Thus, $\tau^*$ depends on the data and, for WB implementations, on the degree of disorder of the stream.

A second distinction concerns outputs between change's invocation and the moment $(wa,ws)^{i+1}$ becomes effective. For TB implementations, with $\tau^*$ set to the latest processed timestamp plus one, any output at $\tau^*$ already conforms to $(wa,ws)^i$ ($\gamma$s' right boundaries are exclusive, see § 2), and any output with a larger timestamp to $(wa,ws)^{i+1}$. For WB implementations, though, since $\tau^*$ may be far from the last tuple's timestamp, outputs for $\gamma$s with $\gamma.r \le \tau^*$ should be computed under $(wa,ws)^i$, and those with $\gamma.r > \tau^*$ under $(wa,ws)^{i+1}$. This separation between requesting a change and applying it at $\tau^*$ allows external controllers to issue reconfiguration commands asynchronously without violating epoch guarantees. Accordingly, in the following Algorithms, besides methods dedicated to retrieving pane boundaries and updating $(wa,ws)$, we also define a method named `reconfigAndAdvance` to apply the configuration change once $\tau^*$ is reached.

Finally, for AMO semantics, updating the prior state upon a reconfiguration entails discarding all panes that do not conform to the newly selected $(wa, ws)$ configuration, while retaining only those compatible with it. This implies that correct results matching those that would have been obtained without discarding any state are observed only once the new $\gamma$s are fully populated. If $\Omega^{i+1}$ is the number of windows a tuple falls into in epoch $i+1$, and $\gamma^{\Omega^{i+1}-1}$ denotes, at $\tau=\tau^*$, the latest window under $(wa,ws)^{i+1}$ that still contains $\tau^*$, the convergence time – i.e., the time during which result correctness is potentially affected by a reconfiguration – is:

$$\begin{cases} ws^{i+1}, & \text{if } \gamma^{\Omega^{i+1}-1}.l=\tau^*, \\ \gamma^{\Omega^{i+1}-1}.l+wa^{i+1}+ws^{i+1}-\tau^*, & \text{otherwise.} \end{cases} \quad (5)$$

**Algorithm 1:** $A^*$ - TB implementation. Methods in gray are found in Algorithms 3 and 5.

```
1  Class p { // user-defined p's class and methods
2      add (t) // add tuple t's contribution to p
3      get () } // get p's partial aggregate
   A* Parameters and Local Variables:
4  𝕎𝔸, 𝕎𝕊, f_K(t), merge(φ^a,φ^b) // A* parameters
5  Γ // list of (wa,ws) combinations in 𝕎𝔸×𝕎𝕊
6  c // Initial/current (wa,ws) index
7  j←null // next (wa,ws) index (upon call of method change)
8  τ_max // latest timestamp observed so far
9  τ* ←null // event-time of change (wa,ws)^i → (wa,ws)^{i+1}
10 TreeMap<τ, Map<k, p>> panes // p instances
11 γ*_L ←null // earliest left boundary of any γ kept by A*
12 p*_L ←null // earliest left boundary of any p kept by A*
   Auxiliary Methods:
13 getEarliestLeftBoundary (τ) // get earliest γ.l/p.l for τ
14 change ((wa,ws)) // change (wa,ws) configuration
15 reconfigAndAdvance (τ) // complete a change (if any) and
     advance A*'s state by discarding stale state up to τ
16 getOrCreate(τ,k) // get or create p and store it in panes
17 getIndex((wa,ws)) // return (wa,ws)'s index in Γ
18 Method process(t)
19     τ_max ←t.τ // update max timestamp observed so far
20     reconfigAndAdvance(τ_max) // check if change can be
       completed & discard stale ps
21     <γ_L, p_L> ← getEarliestLeftBoundary (t.τ)
       // merge panes and produce output tuples
22     while γ*_L ≠null∧γ*_L <γ_L do
23         Map<k, φ> outputs
24         for τ in panes.keys() do
25             if γ*_L ≤τ<γ*_L +Γ[c].ws then
26                 for k in panes[τ].keys() do
27                     outputs[k] ←
                         merge (outputs[k], panes[τ][k].get())
28         for k∈outputs
             do output (<γ*_L +Γ[c].ws−1, outputs[k]>)
29         γ*_L ←γ*_L +Γ[c].wa
       // add t to its pane and update earliest p.l/γ.l
30     p ← getOrCreate (p_L, f_K(t))
31     p.add (t)
32     p*_L ←p_L
33     γ*_L ←γ_L
```

In the following algorithms, we rely on the following data structures: `List`, with method `add` to append elements and notation $[c]$ to access the $c$-th element; `Map`, with method `keys` to retrieve all keys and notation $[c]$ to access the value for key $c$; and `TreeMap`, which behaves like a `Map` but traverses entries in key-sorted order. All the acceptable $(wa,ws)$ combinations are stored in $\Gamma$, where the $c$-th $wa$ and $ws$ are retrieved as $\Gamma[c].wa$ and $\Gamma[c].ws$, respectively. We also assume that `merge`$(\phi^a,\phi^b)$, which combines the partial aggregates of two panes, returns the non-`null` value if exactly one of $\phi^a$ or $\phi^b$ is `null`.

Alg. 1 presents our solution for TB implementations. Methods highlighted in gray depend on the selected semantics (AMO or EO) and are detailed in Alg. 3 and Alg. 5. Since tuples arrive in timestamp order, the algorithm (i) updates the maximum timestamp observed, (ii) invokes `reconfigAndAdvance` to complete any pending configuration change and prune stale panes, and (iii) computes the earliest left boundaries of the $\gamma$ and $p$ to which $t$ contributes (L 19-21). If an input tuple indicates the current window is shifting, outputs are produced for any $p$ that tiles a $\gamma$ for which a result can be created (L 22-29). Once any results have been produced, the tuple being processed is added to its corresponding pane (L 30-33).

**Algorithm 2:** $A^*$ - WB implementation. See Algorithm 1 for $A^*$ parameters and local variables/methods and auxiliary methods not defined here.

```
1  Method process(t)
2      τ_max ← max(τ_max, t.τ) // update max timestamp
       // add t to its pane, update earliest γ.l (if needed)
3      <γ_L, p_L> ← getEarliestLeftBoundary (t.τ)
4      p ← getOrCreate (p_L, f_K(t))
5      p.add (t)
6      if γ*_L = null ∨ γ_L < γ*_L then γ*_L ← γ_L
7  Method process(W)
8      reconfigAndAdvance (W) // check if change can be
       completed & discard stale ps
       // merge panes and produce output tuples
9      while γ*_L ≠ null ∧ γ*_L + Γ[c].ws ≤ W do
10         Map<k, φ> outputs
11         for τ in panes.keys() do
12             if γ*_L ≤ τ < γ*_L + Γ[c].ws then
13                 for * do
14                     k in panes[τ].keys()
15                 outputs[k] ← merge
                   (outputs[k], panes[τ][k].get())
16         for k ∈ outputs
             do output (<γ*_L + Γ[c].ws − 1, outputs[k]>)
17         γ*_L ← γ*_L + Γ[c].wa // update earliest γ boundary
```

Alg. 2 presents our solution for WB implementations, together with Alg. 4 and Alg. 5. Here, the processing logic is split between tuple handling (L 1-6) and watermark handling (L 7-17). When a tuple $t$ is processed, $\tau_{max}$ is updated and $t$ is added to its corresponding pane. The earliest left boundary of any $\gamma$ maintained by $A^*$ is also updated if the left boundary of the $\gamma$ into which $t$ falls precedes the current $\gamma^*_L$. Upon reception of a new watermark, `reconfigAndAdvance` is invoked to complete any pending change (L 8). Then, output tuples are created if such a watermark indicates the current window can be advanced (L 8-17).

Moving now to semantics- and implementation-specific methods, we cover TB implementations and AMO semantics in Alg. 3. The method `change` stores the index of the new configuration in $j$ and sets $\tau^*$. The method `getEarliestLeftBoundary` invokes `get_`$\gamma$`.l` and `get_AMO_`$p$`.l` using index $c$. The method `reconfigAndAdvance` checks whether $\tau^*$ matches the timestamp of the current tuple (if $\tau^*$ is set). If so, it unsets $\tau^*$, updates $c$ to $j$, and updates the current $\gamma^*_L$. Eventually, it clears the state, discarding all panes starting before the earliest window left boundary. Removing panes in this way is safe: when no change is applied, discarded panes end at or before the current earliest window boundary and can no longer contribute to any future window; when a change is applied, discarded panes cover ranges of event time that are not part of any window under the new configuration.

**Algorithm 3:** Methods for TB implementation and AMO semantics.

```
1  Method change((wa,ws))
2      j ← getIndex((wa,ws)) // store new wa/ws index in j
3      τ* ← τ_max + 1 // set τ*, implying a change is requested
4  Method getEarliestLeftBoundary(τ)
5      return (get_γ.l(τ, (Γ[c].wa, Γ[c].ws),
         get_AMO_p.l(τ, (Γ[c].wa, Γ[c].ws)))
6  Method reconfigAndAdvance(τ)
7      if τ* ≠ null ∧ τ_max ≥ τ* then // the change is applied
8          τ* ← null
9          c ← j
10         γ*_L ← get_γ.l(τ, (Γ[c].wa, Γ[c].ws))
11     for τ' in panes.keys() do // remove stale panes
12         if τ' < γ*_L then
13             panes.remove (τ')
```

For WB implementations and AMO semantics (Alg. 4), when $\tau^*$ is set, `getEarliestLeftBoundary` chooses between indexes $c$ and $j$, based on the current value of $\tau$. The other methods are the same as those presented in Alg. 3.

**Algorithm 4:** Methods for WB implementation and AMO semantics.

```
1  Method change((wa,ws))// same as in Algorithm 3
2  Method getEarliestLeftBoundary(τ)
3      if τ* ≠ null ∧ τ ≥ τ* then // boundaries chosen
       depending on the epoch τ falls in
4          return ( get_γ.l(τ, (Γ[j].wa, Γ[j].ws)),
             get_AMO_p.l(τ, (Γ[j].wa, Γ[j].ws)) )
5      else return ( get_γ.l(τ, (Γ[c].wa, Γ[c].ws)),
         get_AMO_p.l(τ, (Γ[c].wa, Γ[c].ws)) )
6  Method reconfigAndAdvance(τ)// same as in Algorithm 3
```

For EO semantics (Alg. 5), the same methods apply to both TB and WB implementations. Method `change` updates immediately $c$ to the new configuration index. Method `getEarliestLeftBoundary` also relies on index $c$, but invokes `get_EO_`$p$`.l`. Finally, method `reconfigAndAdvance` removes stale panes only if they fall strictly before the largest $ws$ in $\mathbb{WS}$ – the maximum time span that the window can cover – and the smallest $wa$ in $\mathbb{WA}$ – the precise alignment of the acceptable configuration start a window, ensuring that no state required by any future configuration is lost. In practice, this means buffering panes longer to avoid inconsistencies during future reconfigurations.

## 5 Evaluation

In our evaluation, we assess Chill's performance by running $A^*$ under both TB and WB implementations, as well as their AMO and EO

**Algorithm 5:** Methods for TB and WB implementations, and EO semantics.

```
1 Method change((wa,ws))
2     c ← getIndex((wa,ws))
3     γ_L ← get_γ.l(τ,(Γ[c].wa,Γ[c].ws)
4 Method
    getEarliestLeftBoundary(τ)// as in Algorithm 3 but
    calling get_EO_p.l
5 Method reconfigAndAdvance(τ)
6     for τ' in panes.keys() do
7         if τ' < τ − max(𝕎𝕊) + min(𝕎𝔸) then // keep panes
          based on the largest ws and the smallest wa
8             panes.remove (τ')
```

variants (see § 4). We first study the reconfiguration overhead of $A^*$. To further contextualize the performance characteristics observed for the different implementations and semantics, we analyze their trade-offs as the window configuration space expands, which is a key aspect of $A^*$'s design. Finally, we compare $A^*$ against state-of-the-art baselines. The first baseline is Liebre [11], since Liebre is the SPE upon which we build $A^*$ and it adopts the Single-Window (SW) execution strategy with a TB implementation. To also consider an SPE widely adopted by the community and the industry, we use Flink [9] as a baseline, a widely used SPE that supports out-of-order processing and employs the Multi-Window (MW) execution strategy, making it a natural point of comparison for our WB implementation. We note that these baseline comparisons involve two orthogonal factors: (1) the execution strategy – SW, Pane-based, or MW (see § 2); and (2) the underlying SPE – Liebre or Flink. On the one hand, since $A^*$ runs on the Liebre SPE, we compare it with the SW execution strategy in Liebre to isolate the algorithmic contribution of $A^*$. On the other hand, we compare $A^*$ with Flink and its MW execution strategy to position $A^*$ relative to a widely adopted SPE.

To ensure that our evaluation captures both controlled stress testing and application realism, we employ two complementary datasets: a *Synthetic* dataset that allows fine-grained control over input characteristics (e.g., key distributions) and the widely used *Linear Road* benchmark [3], representing a realistic use case with data generated by vehicles moving on highways. As stated in § 3, beyond assessing how implementations and semantics affect the *Throughput*, *Latency*, *CPU*, and *Panes* metrics, we analyze reconfiguration behavior by measuring both *Reconfiguration time (RC)* and *Convergence time (CV)* across transitions between epochs. On the one hand, our experiments aim to show that, despite performance differences across implementations and semantics, the proposed algorithms enable fast reconfigurations and trade-offs between AMO and EO semantics, with AMO favoring performance at the cost of temporary result inaccuracy during reconfiguration, and EO guaranteeing correctness throughout, incurring a modest and predictable performance overhead. On the other hand, we also show that $A^*$ is competitive with, or can outperform, state-of-the-art baselines across all scenarios. All our code is publicly available at https://doi.org/10.5281/zenodo.20280981.

**Hardware/Software.** All experiments are conducted on a single server with an Intel Xeon E5-2637 v4@3.50GHz, 4 cores, 8 threads, and 64 GB RAM. As discussed in [12], this server is a device representative of those found in edge-to-cloud applications. Our framework uses Java (OpenJDK version 17.0.14). $A^*$ is implemented as a library on top of Liebre version 0.1.1, while the baselines are implemented using Liebre version 0.1.1 and Flink version 1.19.0, respectively.

**Data and use cases.** The first dataset, Synthetic, generates tuples live during execution rather than loading them from recorded traces. Hence, each tuple is timestamped at the moment it is generated and forwarded. Each tuple consists of a key in [1,100000] (used by $f_K(t)$) and a randomly generated integer value. Synthetic offers full controllability: injection rate, key distribution, and value range can be precisely configured to support reproducible experiments and targeted stress testing.

The second dataset, Linear Road [3], is a widely used benchmark for evaluating stream processing systems. It emulates a real-time traffic monitoring application in which vehicles emit position reports every 30 seconds. Each tuple in this recorded trace includes an event timestamp, a vehicle ID, its position, and its speed; the vehicle ID is used by $f_K(t)$ to maintain independent window states. Individual vehicles contribute data ranging from a few minutes to tens of minutes, and the full trace spans about 3 hours of event time, containing roughly 44 million tuples.

Since our goal is to measure the maximum sustainable performance of $A/A^*$ – for example, maximum Throughput – both datasets are injected as fast as the system can process them under *flow control*. This design choice, together with how tuples are timestamped (i.e., at the moment of generation for Synthetic and loading the timestamps from the recorded trace for Linear Road), has an important consequence. For Synthetic, each tuple is timestamped at the time it is generated and forwarded. As a result, event time and wall-clock time evolve synchronously, making window behavior – e.g., output emission precisely every $wa$ – faithfully observable and preserving the semantics expected from a live data stream processed by $A/A^*$. For Linear Road, though, given that its data is a recorded trace, tuples are replayed faster than their event timestamps advance. Event time progresses over 3 hours, whereas wall-clock time progresses over only a few minutes, meaning behaviors tied to event-time evolution – such as producing outputs exactly every $wa$ time units – are not observable in wall-clock execution (see § 2). Together, these datasets allow us to evaluate $A^*$ across use cases in which event time and wall-clock time are aligned and others in which they are not, thereby covering both live data streams and recorded traces, which are common in real-world applications.

In both datasets, we evaluate an aggregate function with a non-negligible computational cost. In Synthetic, this function performs square-sum computations over tuple values, where `add(t)` applies arithmetic operations on incoming tuples and `get()` outputs the accumulated result. In Linear Road, it computes the number of non-consecutive stops per vehicle, where a stop is a sequence of zero-speed reports separated by non-zero-speed reports, including those spanning window boundaries. For a given $\gamma$, `add(t)` incorporates incoming tuples and `get()` returns the number of non-consecutive stops within $\gamma$. Note that, since Synthetic uses a fixed set of keys, `add(t)` and `get()` exhibit stable per-tuple and per-key costs. In Linear Road, instead, such costs are key-dependent, since some vehicles contribute many more position reports than others in the recorded

trace. Synthetic's aggregate function is computationally less expensive than Linear Road's. Unless otherwise stated, each experiment is repeated 5 times and we report the average values per epoch across repetitions. In WB implementations, watermarks are emitted 5 times per second, i.e., every 200 ms, and tuples are delivered out of order for both Synthetic and Linear Road.

### Window Reconfiguration Overhead

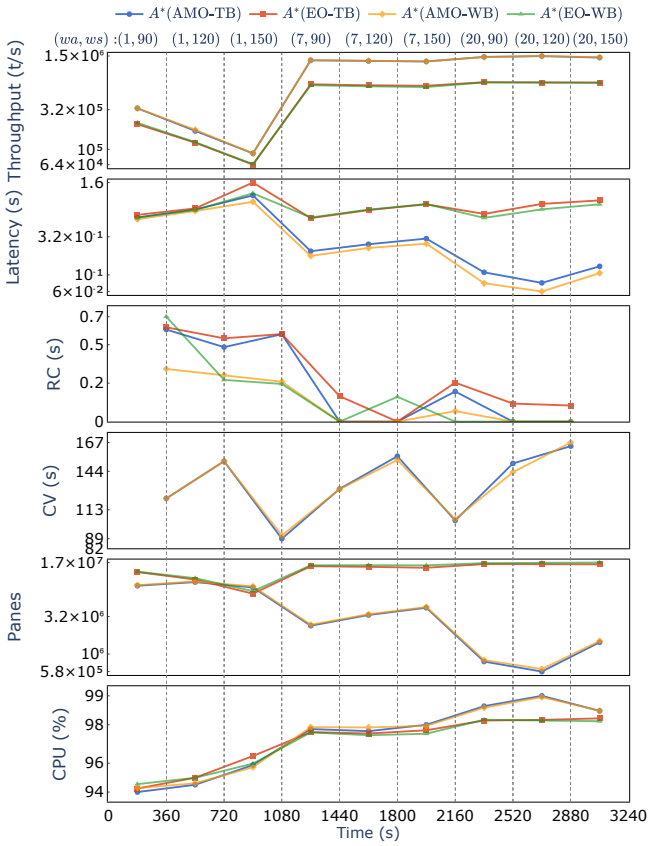



**Figure 3: Window reconfiguration overhead for Synthetic.**

To quantify window reconfiguration overhead, we consider the full Cartesian product of $\mathbb{WA} = \{1, 7, 20\}$ and $\mathbb{WS} = \{90, 120, 150\}$ seconds, yielding 9 (*wa*,*ws*) configurations. For Linear Road, this space includes the two setups from the running example in § 1, namely (1,90) and (20,90), although here they are evaluated with the heavier aggregation function described above. This configuration space captures scenarios in which analysts may wish to vary both the output frequency and the temporal span of the aggregation. Each epoch lasts 360 seconds, including a 160-second warm-up and a 20-second cool-down. The warm-up period is based on the largest *ws* (150 seconds) to ensure that all windows are fully populated before collecting any performance measurements. For each metric, we first compute the average within each repetition and then average those values across repetitions. In the plots, time is shown on the x-axis and each row reports one statistic. The (*wa*,*ws*) pair used in each epoch is indicated at the top, while dotted vertical lines mark the transitions between epochs. Points located in the middle of an epoch report statistics collected during that epoch, whereas points on the vertical lines correspond to transition-related measurements (i.e., RC and CV). Some y-axes are shown on a logarithmic scale to better appreciate the differences across configurations.

Figure 3 shows the window reconfiguration overhead results on Synthetic. For a fixed *wa*, Throughput decreases as *ws* increases, since each output requires merging more panes; conversely, for a fixed *ws*, Throughput is lower for smaller *wa*, as outputs are produced more frequently. These differences are most visible when *wa* is small, especially for $wa = 1$, while they are much less pronounced between $wa = 7$ and $wa = 20$. Quantitatively, for $wa = 1$, Throughput decreases from about $3 \times 10^5$ t/s at $ws = 90$ to roughly $7 \times 10^4$ t/s at $ws = 150$, whereas for $wa = 20$ it consistently exceeds $6.3 \times 10^5$ t/s for all *ws* values.

Latency mirrors this behavior, increasing with larger *ws* and smaller *wa*. Across both TB and WB implementations, EO consistently yields lower Throughput and higher Latency than AMO, which is expected because EO maintains finer-grained panes over the full acceptable configuration space induced by $\mathbb{WA}$ and $\mathbb{WS}$ (see Eq. 4). In quantitative terms, all latency values remain below 1 second except for the most demanding case, $(wa,ws) = (1,150)$, which rises slightly above that threshold.

RC remains small overall, albeit with some oscillation, and is generally higher for $wa = 1$ than for larger *was*; this is because RC is tied to output production, as reconfiguration may need to wait for ongoing emissions on $A^*$'s main thread, making it closely related to the observed Latency. In all cases, however, RC remains below 1 second.

CV under AMO falls within the range predicted by Eq. 5. Finally, Panes shows only minor differences between TB and WB implementations; under EO, it remains nearly constant because it depends on the full acceptable configuration space, whereas under AMO it decreases with larger *wa* and increases with larger *ws*. CPU remains high and stable throughout, indicating that the system operates close to saturation.

Figure 4 confirms the same overall behavior observed for Synthetic. Throughput decreases as *ws* increases and increases as *wa* increases; however, because the Linear Road aggregate function is computationally heavier, overall Throughput is lower, ranging from approximately $4 \times 10^4$ t/s for $wa = 1$ to approximately $2.3 \times 10^5$ t/s for $wa = 20$, and the effect of larger *ws* values is less pronounced than in Synthetic, whereas the gain from increasing *wa* remains clearly visible even when moving from $wa = 7$ to $wa = 20$. Moreover, for a fixed (*wa*,*ws*) configuration, the higher computational cost tends to dominate the execution, making the Throughput of the different implementations and semantics much closer to each other.

Latency follows the inverse trend of Throughput, as before, but with one important difference: WB implementations exhibit higher values than TB ones, with values around 0.45 seconds for WB and 0.25 seconds for TB when $wa = 1$, and around 0.2 seconds for WB and 0.1 seconds for TB when $wa = 7$, across the corresponding *ws* values. This is expected because, in Linear Road, event time advances faster than wall-clock time, allowing TB implementations to produce results earlier, whereas WB ones still wait for periodically emitted watermarks, which in our setup arrive every 200 ms.

RC follows the same qualitative pattern and, again, TB implementations tend to show smaller values. CV exhibits the same trend as in Synthetic, but the observed values are significantly smaller than the corresponding bounds, which is also expected because those

bounds are expressed in event time, while in this experiment event time advances much faster than wall-clock time.

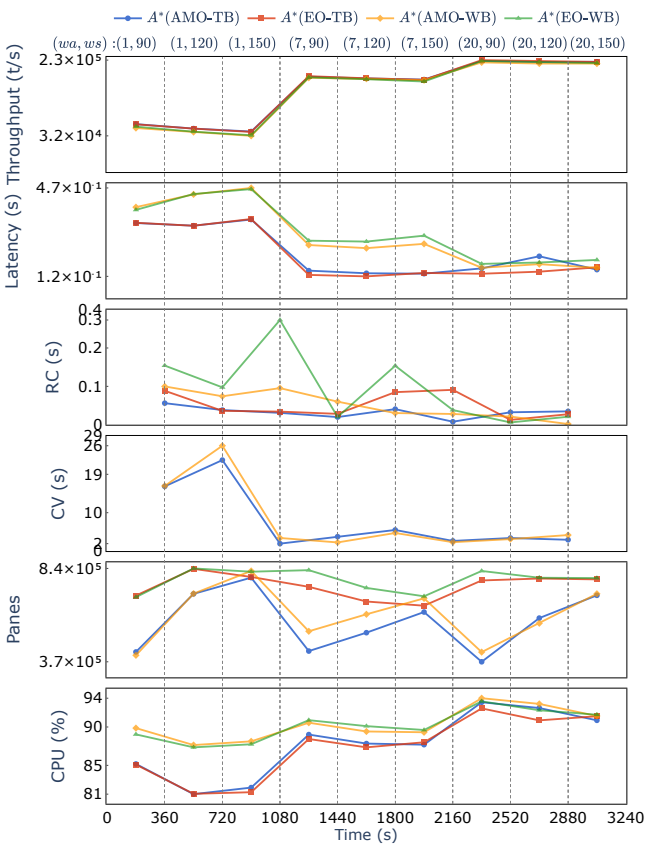



**Figure 4: Window reconfiguration overhead for Linear Road.**

The Panes metric follows the same overall trend as well, but the difference between TB and WB implementations is more visible here for two reasons: unlike Synthetic, the number of active keys changes over time, and WB implementations purge stale panes less frequently than TB ones, resulting in higher values. Finally, the CPU plot shows that the system remains close to saturation and follows the same expected trend as in Synthetic, while also displaying a clearer separation between TB and WB implementations.

### Impact of Growing Window Configuration Spaces

To further validate the expected trade-offs, we examine how enlarging the window configuration space affects EO, whose overheads grow because it must maintain $ps$ that tile $ys$ for every $(wa,ws) \in \mathbb{WA} \times \mathbb{WS}$ (see § 4). As $|\mathbb{WA} \times \mathbb{WS}|$ grows, the pane partition becomes increasingly granular, which increases the number of panes and, in turn, reduces Throughput while increasing Latency and Panes.

Figure 5 shows this effect on Synthetic. In this experiment, the warm-up and cool-down are 95 and 10 seconds, respectively. Each column reports Throughput, Latency, CPU, and Panes for one choice of $\mathbb{WA}$ and $\mathbb{WS}$, shown at the top of the figure. The bold values identify the fixed execution configuration, which remains $(wa,ws) = (20,90)$ throughout, while the surrounding sets grow from left to right. For each semantics/implementation pair, we use a violin plot to summarize the full distribution of the observed values; inside each violin, three black dashed lines indicate the first, second, and third quartiles from top to bottom.

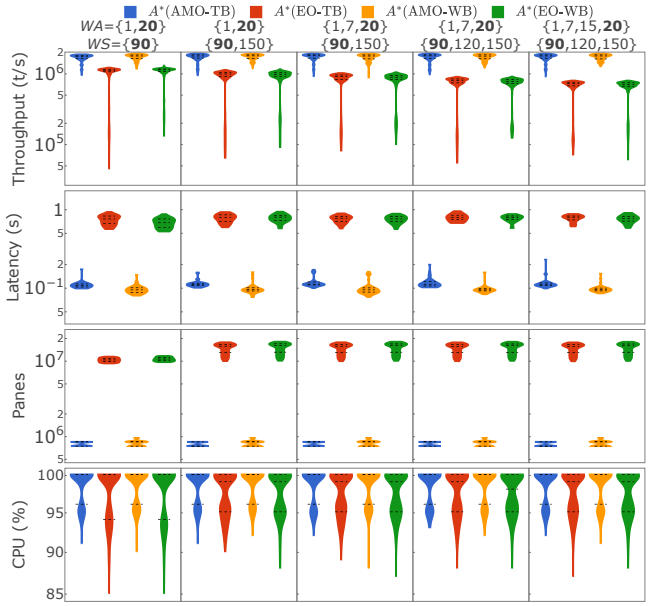



**Figure 5: Performance impact of different $|\mathbb{WA} \times \mathbb{WS}|$ for Synthetic.**

AMO exhibits limited sensitivity to the configuration space size, with metrics remaining relatively constant: Throughput hovering above $1.5 \times 10^6$ t/s, Latency remaining around 0.1 seconds, CPU close to saturation, and Panes stabilizing at approximately $7.5 \times 10^5$. In contrast, EO follows the expected trend: as $|\mathbb{WA} \times \mathbb{WS}|$ increases, Throughput decreases, while Latency and Panes increase.

While the cost of EO semantics can increase as the acceptable configuration space grows, we note that it can be kept under control through a careful choice of that space. In particular, although larger spaces generally increase the overhead of EO, introducing extra $wa$ values does not substantially raise the cost when they are all multiples of the smallest advance, because the partitioning of event time into panes does not change.

Figure 6 illustrates this point on Linear Road. Here we enlarge the configuration space while choosing $wa$ values that are all multiples of the smallest one. As shown, while a modest overhead is still visible for larger spaces and larger $ws$ values, the gap between AMO and EO is much smaller than in the previous experiment, indicating that a careful selection of the configuration space can help mitigate the cost of EO semantics.

### Comparison with Baseline Solutions

We now turn to the baseline comparison, building directly on the reconfiguration experiments. For space reasons, we do not report all epochs, but only the first and the last, which correspond to the two extreme points of the explored configuration spectrum: the pair with the smallest $wa$ and $ws$, and the pair with the largest ones. This lets us compare $A^*$ against the baselines within each selected epoch while also showing how the performance metrics shift from the first to the last $(wa,ws)$ pair.

To focus the comparison on $A^*$ versus the baselines – rather than on the trade-offs between TB and WB implementations or between

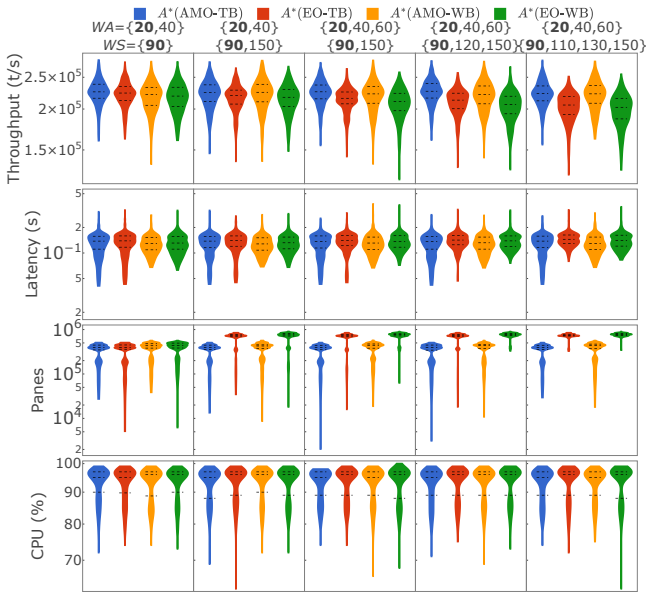



**Figure 6: Performance impact of different $|\mathbb{WA} \times \mathbb{WS}|$ for Linear Road.**

AMO and EO semantics – we assume that the chosen $(wa,ws)$ pair is the only acceptable configuration when running $A^*$. That is, for $A^*$, the acceptable configuration space contains only one element, namely the selected $(wa, ws)$ pair. As in the reconfiguration experiments, epoch duration, warm-up, and cool-down are 360, 160, and 20 seconds, respectively. Figures 7 and 8 report violin plots for Throughput, Latency, and CPU only. Each violin summarizes the full distribution of the observed values, with an overlaid three black dashed lines marking the first, second, and third quartiles, carrying the same information as in the increasing-set experiment above. We omit Panes because Liebre follows the SW execution strategy and Flink the MW one (see § 2), and neither baseline relies on panes.

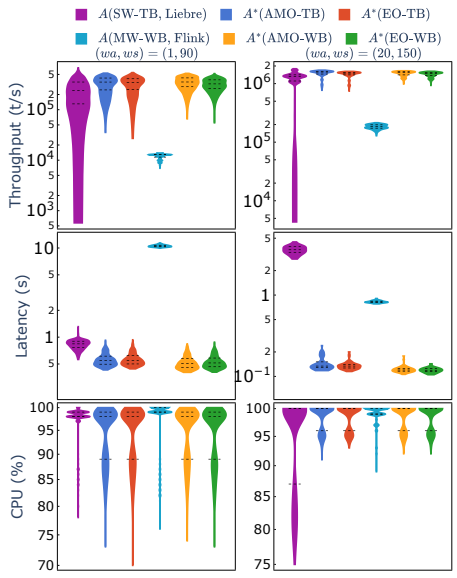



**Figure 7: Violin plots of Throughput, Latency, and CPU across baselines and $A^*$ variants for Synthetic.**

Figure 7 shows that, for the smallest configuration $(wa, ws) = (1,90)$, Liebre and the four $A^*$ variants achieve comparable Throughput in the few-$10^5$ t/s range on average, whereas Flink remains far lower. The relatively wide violins in this case indicate substantial variability, which is expected: with $wa$ set to 1 second and 100,000 keys, outputs are produced at a very high rate, making output generation itself a significant cost and causing temporary throughput drops when results are emitted. For the largest configuration, $(wa,ws) = (20,150)$, the same qualitative ranking remains, but Throughput increases overall and the gap between Flink and the other approaches narrows. Quantitatively, for (1,90), Liebre and the $A^*$ variants average between $3\times10^5$ and $4\times10^5$ t/s, whereas Flink remains around $10^4$ t/s. For (20,150), Liebre and $A^*$ move into the $10^6$–$2\times10^6$ t/s range, while Flink reaches about $2\times10^5$ t/s. Liebre remains close to $A^*$, although with somewhat larger dispersion. Latency follows the same pattern: $A^*$ and Liebre remain comparable, with Liebre slightly higher, while Flink is noticeably higher, especially for (1,90). CPU utilization stays close to saturation in all cases, and the spread of CPU values largely follows the same trend observed for Throughput.

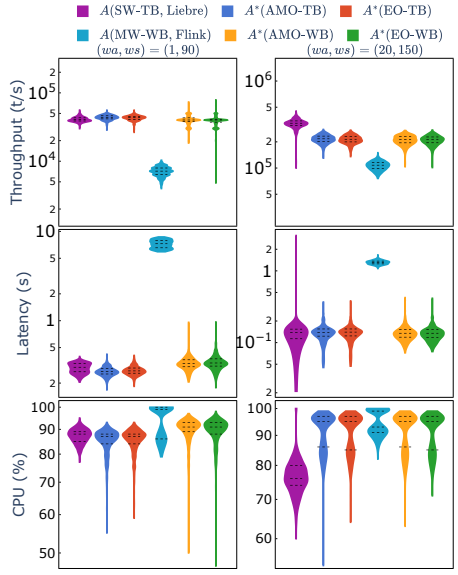



**Figure 8: Violin plots of Throughput, Latency, and CPU across baselines and $A^*$ variants for Linear Road.**

Figure 8 highlights a similar overall picture on Linear Road. For the smallest configuration, $A^*$ achieves Throughput comparable to Liebre and clearly higher than Flink, with TB variants slightly above and less variable than WB ones. Quantitatively, for (1,90), Liebre and $A^*$ remain between $4\times10^4$ and $5\times10^4$ t/s, whereas Flink stays around $7\times10^3$ t/s. When moving to the largest configuration, Throughput increases for all systems; Liebre reaches the highest values, $A^*$ follows closely, and Flink narrows the gap while remaining lower. For (20,150), Liebre reaches around $3\times10^5$ t/s, $A^*$ around $2\times10^5$ t/s, and Flink about $10^5$ t/s. Notably, Flink's Throughput is of the same order as in Synthetic, which likely reflects serialization overheads rather than only workload characteristics.

Latency follows the same qualitative trend as in Synthetic. The latency of $A^*$ and Liebre remains around 0.2 seconds, whereas Flink still exhibits a much wider range, from about 7.3 seconds for the

smallest pair to about 1.3 seconds for the largest. CPU utilization is high throughout: for the smallest pair, all approaches operate around 90% or more, with Flink reaching almost full utilization, while for the largest pair CPU tends to increase further and shows greater variability as the systems process larger amounts of data between outputs.

**Evaluation Summary.** Overall, Chill delivers robust performance across diverse window configurations, datasets, and workloads, maintaining low latency and high efficiency while exhibiting predictable scaling of resource usage; RC is negligible in all cases, CV fits our expectations (see Eq. 5 in § 4), and both semantics enable fast reconfigurations, with AMO favoring performance and EO preserving correctness at a modest, predictable cost. Results also show consistent effects when enlarging the window configuration space, indicating an additional trade-off between AMO and EO: AMO continues to favor performance even when the set of possible $(wa,ws)$ configurations becomes larger. Finally, although $A^*$ does not always outperform every baseline – for example, Liebre on Linear Road for $(wa,ws) = (20,150)$ – it remains comparable across all scenarios and can outperform established systems such as Flink.

## 6 Related Work

To the best of our knowledge, Chill is the first stream Aggregate that allows analysts to define a set of acceptable window configurations and actively switch among them at runtime, while accounting for both the processing of timestamp-sorted (TB implementation) and out-of-order (WB implementation) streams and offering both at-most-once (AMO) and exactly-once (EO) semantics during reconfigurations. Prior work has explored how to trade relaxed semantics for cost in streaming, but typically without changing $wa$ and $ws$ via explicit, user-directed reconfiguration across discrete configurations with epoch guarantees.

A large body of work allows for adjusting per-node computation costs by shaping where and how computation happens, or by approximating results. Many of the following techniques can also be combined (with ours, too):

- **Distribution** [1]: push operators to multiple nodes to reduce per-node load; semantics are preserved, and per-node load decreased, but at the price of added coordination and network costs.
- **Parallelization** [13]: replicate operators (e.g., key-partitioned) across cores and nodes to scale up and out, respectively, while preserving semantics and reducing per-node load, but at the price of added coordination and network costs.
- **Wavelets** [8]: compress streams into hierarchical summaries for sliding windows, enabling fast queries whose approximation degree depends on the transform resolution rather than relaxing on changing $(wa,ws)$.
- **Sketches** [18]: maintain compact probabilistic summaries (e.g., heavy hitters) with provable error bounds; also orthogonal to a specific $(wa,ws)$ and usually covering streams in their entirety.
- **Sampling** [6]: down-sample inputs (random or structure-aware) to cut processing cost, which lowers load but introduces sampling error decoupled from explicit $(wa,ws)$ choices.
- **Compression** [17]: reduce the footprint of operator state via on-demand compression; preserves semantics but targets memory pressure rather than runtime changes to $wa/ws$.
- **Load shedding** [21]: drop tuples or entire $\gamma$s under overload to preserve latency/QoS; trades completeness for timeliness without reconfiguring window parameters.

Closer in spirit are models where the *effective* window span evolves over time:

- **Session windows** (supported in SPEs such as Flink [9]): window boundaries are driven by inactivity gaps (merge contiguous events if inter-arrival time is below a gap). Window length thus depends on characteristics of the input data, not on explicit $(wa,ws)$ pairs selected at runtime. Analysts set a gap parameter, but do not *switch among multiple* $(wa,ws)$ configurations with epoch-level guarantees.
- **Time-decayed/weighted windows**: exponential or polynomial decay assigns a lower weight to older tuples while logically spanning the entire stream [7]; the decay factor controls recency emphasis, not discrete window advance/size. This *approximates* earlier contributions rather than selecting and switching among exact $(wa,ws)$ settings.

Both lines yield outputs that adapt to workload characteristics, but they do not provide an interface to choose from a predeclared family of $(wa,ws)$ configurations and to *switch* among them with explicit at-most-once/exactly-once behavior across epochs as in Chill. In particular, session gaps and decay factors shape window *behavior* continuously, whereas Chill reconfigures window *parameters* discretely and deterministically, preserving well-defined boundaries between epochs and accommodating both TB and WB assumptions.

## 7 Conclusions and Future Work

We presented Chill, the first stream Aggregate that, to the best of our knowledge, supports runtime reconfiguration of window advance ($wa$) and size ($ws$). Chill adopts a pane-based design for efficient state reuse across configurations and supports two semantics – AMO and EO, allowing analysts to trade accuracy for performance as needs evolve.

We introduced algorithms suitable for SPEs that process either timestamp-ordered or out-of-order streams, specializing only a small set of semantics- and implementation-dependent methods while ensuring clean epoch boundaries across reconfigurations (see § 4). We provided full implementations and an exhaustive evaluation showing that Chill matches fixed-configuration baselines, with negligible reconfiguration overhead under both AMO and EO, and predictable convergence under AMO semantics.

Future work includes addressing Chill's scalability in distributed, multi-node settings; AI-driven controllers to trigger reconfiguration and choose $(wa,ws)$ and semantics online based on workload and resource signals; broader families of relaxed policies across transitions; and extensions to multi-instance deployments (e.g., key-partitioned, elastic scaling) with end-to-end correctness guarantees.

## Acknowledgments

Work supported by the Marie Skłodowska-Curie Doctoral Network project RELAX-DN, funded by the European Union under Horizon Europe 2021-2027 Framework Programme Grant Agreement number 101072456, the Chalmers AoA Energy projects DEEP and INDEED, the Swedish Energy Agency (SESBC) project TANDEM, the Wallenberg AI, Autonomous Systems and Software Program and

Wallenberg Initiative Materials for Sustainability project STRATIFIER.

# A Symbols and Abbreviations

Table 1 collects all the symbols and abbreviations used in the paper.

**Table 1: Symbols and abbreviations used in the paper.**

| | |
|---|---|
| $A$ | Generic stream Aggregate |
| $A^*$ | Our solution, which adopts a pane-based execution strategy to change its configuration from $(wa, ws)^i$ to $(wa, ws)^{i+1} \in \mathbb{WA} \times \mathbb{WS}$ at runtime |
| AMO semantics | At-most-once semantics |
| CV | Convergence time |
| $\delta$ | SPE-specific event time increments |
| EO semantics | Exactly-once semantics |
| epoch | A contiguous interval of event time during which a specific $(wa, ws)$ pair is used by $A^*$ |
| $f_K(t)$ | Key extraction function for tuple $t$ |
| $\gamma$ | Window instance |
| $\gamma.l$ | Left boundary of the $\gamma$ |
| $\gamma.r$ | Right boundary of the $\gamma$ |
| $\Gamma$ | List of $(wa, ws)$ combinations in $\mathbb{WA} \times \mathbb{WS}$ |
| MW | Multiple-Window execution strategy |
| $\Omega$ | The set of $\gamma$s of $wa$and $ws$to which a tuple $t.\tau = \tau$ contributes |
| $p$ | Pane instance |
| $p.l$ | Left boundary of the $p$ |
| $p.r$ | Right boundary of the $p$ |
| RC | Reconfiguration time |
| $S$ | Stream |
| SW | Single-Window execution strategy |
| SPE | Stream Processing Engine |
| $t$ | Generic tuple |
| $t_i$ | Generic input tuple |
| $t.\tau$ | Timestamp carried by tuple $t$ |
| $t.\phi$ | Payload carried by tuple $t$ |
| $\tau_{max}$ | The highest tuple timestamp observed by $A^*$ during its execution |
| $\tau^*$ | The event-time from which the new epoch starts when a reconfiguration is triggered |
| $t_o$ | Generic output tuple |
| TB implementation | Timestamp-based implementation |
| $W$ | Generic watermark |
| $W_A^\omega$ | Aggregate $A$'s watermark $W$ at wall-clock time $\omega$ |
| $wa$ | Window advance |
| $\mathbb{WA}$ | Set of acceptable $wa$ values |
| WB implementation | Watermark-based implementation |
| $ws$ | Window size |
| $\mathbb{WS}$ | Set of acceptable $ws$ values |

# References


[1] Yanif Ahmad, Bradley Berg, Uğur Cetintemel, Mark Humphrey, Jeong-Hyon Hwang, Anjali Jhingran, Anurag Maskey, Olga Papaemmanouil, Alexander Rasin, Nesime Tatbul, Wenjuan Xing, Ying Xing, and Stan Zdonik. 2005. Distributed Operation in the Borealis Stream Processing Engine. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data (SIGMOD '05)*. Association for Computing Machinery, New York, NY, USA, 882–884. https://doi.org/10.1145/1066157.1066274

[2] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. 2015. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proc. VLDB Endow.* 8, 12 (Aug. 2015), 1792–1803. https://doi.org/10.14778/2824032.2824076

[3] Arvind Arasu, Mitch Cherniack, Eduardo Galvez, David Maier, Anurag S Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. 2004. Linear road: a stream data management benchmark. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*. 480–491.

[4] Cagri Balkesen, Nesime Tatbul, and M. Tamer Özsu. 2013. Adaptive Input Admission and Management for Parallel Stream Processing. In *Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems (DEBS '13)*. Association for Computing Machinery, New York, NY, USA, 15–26. https://doi.org/10.1145/2488222.2488258

[5] beam [n.d.]. Apache Beam®. https://beam.apache.org/.

[6] Edith Cohen, Graham Cormode, and Nick Duffield. 2011. Structure-Aware Sampling on Data Streams. In *Proceedings of the ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '11)*. Association for Computing Machinery, New York, NY, USA, 197–208. https://doi.org/10.1145/1993744.1993763

[7] Graham Cormode and S. Muthukrishnan. 2005. An Improved Data Stream Summary: The Count-Min Sketch and Its Applications. *Journal of Algorithms* 55, 1 (April 2005), 58–75. https://doi.org/10.1016/j.jalgor.2003.12.001

[8] Mayur Datar and Rajeev Motwani. 2016. The Sliding-Window Computation Model and Results. In *Data Stream Management: Processing High-Speed Data Streams*, Minos Garofalakis, Johannes Gehrke, and Rajeev Rastogi (Eds.). Springer, Berlin, Heidelberg, 149–165. https://doi.org/10.1007/978-3-540-28608-0_7

[9] flink [n.d.]. Apache Flink® — Stateful Computations over Data Streams. https://flink.apache.org/.

[10] Prajith Ramakrishnan Geethakumari, Vincenzo Gulisano, Bo Joel Svensson, Pedro Trancoso, and Ioannis Sourdis. 2017. Single Window Stream Aggregation Using Reconfigurable Hardware. In *2017 International Conference on Field Programmable Technology (ICFPT)*. 112–119. https://doi.org/10.1109/FPT.2017.8280128

[11] Vincenzo Gulisano. 2022. Vincenzo-Gulisano/Liebre.

[12] Vincenzo Gulisano. 2023. An Algorithm for Tunable Memory Compression of Time-Based Windows for Stream Aggregates. In *European Conference on Parallel Processing*. Springer, 18–29.

[13] Vincenzo Gulisano, Ricardo Jiménez-Peris, Marta Patiño-Martínez, Claudio Soriente, and Patrick Valduriez. 2012. StreamCloud: An Elastic and Scalable Data Streaming System. *IEEE Transactions on Parallel and Distributed Systems* 23, 12 (Dec. 2012), 2351–2365. https://doi.org/10.1109/TPDS.2012.24

[14] Vincenzo Gulisano, Yiannis Nikolakopoulos, Daniel Cederman, Marina Papatriantafilou, and Philippas Tsigas. 2017. Efficient Data Streaming Multiway Aggregation through Concurrent Algorithmic Designs and New Abstract Data Types. *ACM Trans. Parallel Comput.* 4, 2 (Oct. 2017), 11:1–11:28. https://doi.org/10.1145/3131272

[15] Vincenzo Gulisano, Dimitris Palyvos-Giannas, Bastian Havers, and Marina Papatriantafilou. 2020. The role of event-time order in data streaming analysis. In *Proceedings of the 14th ACM International Conference on Distributed and Event-based Systems (DEBS '20)*. Association for Computing Machinery, New York, NY, USA, 214–217. https://doi.org/10.1145/3401025.3404088

[16] Bastian Havers, Romaric Duvignau, Hannaneh Najdataei, Vincenzo Gulisano, Marina Papatriantafilou, and Ashok Chaitanya Koppisetty. 2020. DRIVEN: A Framework for Efficient Data Retrieval and Clustering in Vehicular Networks. *Future Generation Computer Systems* 107 (June 2020), 1–17. https://doi.org/10.1016/j.future.2020.01.050

[17] Jingyu Liu and Vincenzo Gulisano. 2025. On-Demand Memory Compression of Stream Aggregates through Reinforcement Learning. In *Proceedings of the 16th ACM/SPEC International Conference on Performance Engineering (ICPE '25)*. Association for Computing Machinery, New York, NY, USA, 240–252. https://doi.org/10.1145/3676151.3719369

[18] Vinh Quang Ngo and Marina Papatriantafilou. 2025. Cuckoo Heavy Keeper and the Balancing Act of Maintaining Heavy Hitters in Stream Processing. *Proc. VLDB Endow.* 18, 9 (Sept. 2025), 3149–3161. https://doi.org/10.14778/3746405.3746434

[19] Dimitris Palyvos-Giannas, Bastian Havers, Marina Papatriantafilou, and Vincenzo Gulisano. 2020. Ananke: A Streaming Framework for Live Forward Provenance. *Proc. VLDB Endow.* 14, 3 (Nov. 2020), 391–403. https://doi.org/10.14778/3430915.3430928

[20] Dimitris Palyvos-Giannas, Katerina Tzompanaki, Marina Papatriantafilou, and Vincenzo Gulisano. 2022. Erebus: Explaining the Outputs of Data Streaming Queries. *Proc. VLDB Endow.* 16, 2 (Oct. 2022), 230–242. https://doi.org/10.14778/3565816.3565825

[21] Nesime Tatbul. 2002. QoS-Driven Load Shedding on Data Streams. In *XML-Based Data Management and Multimedia Engineering — EDBT 2002 Workshops*, Akmal B. Chaudhri, Rainer Unland, Chabane Djeraba, and Wolfgang Lindner (Eds.). Springer, Berlin, Heidelberg, 566–576. https://doi.org/10.1007/3-540-36128-6_36

[22] Nesime Tatbul, Uğur Çetintemel, Stan Zdonik, Mitch Cherniack, and Michael Stonebraker. 2003. Load shedding in a data stream manager. In *Proceedings 2003 vldb conference*. Elsevier, 309–320.

[23] Jonas Traub, Philipp Marian Grulich, Alejandro Rodríguez Cuéllar, Sebastian Breß, Asterios Katsifodimos, Tilmann Rabl, and Volker Markl. 2021. Scotty: General and efficient open-source window aggregation for stream processing systems. *ACM Transactions on Database Systems (TODS)* 46, 1 (2021), 1–46.