Thesis for The Degree of Licentiate of Engineering

Adaptive and Efficient Event Stream Processing through Relaxed Semantics

Jingyu Liu

Department of Computer Science and Engineering
Chalmers University of Technology | University of Gothenburg
Gothenburg, Sweden, 2026

Adaptive and Efficient Event Stream Processing through Relaxed Semantics

Jingyu Liu

ISSN 1652-876X

Department of Computer Science and Engineering
Division of Computer and Network Systems
Distributed Computing and Systems
Chalmers University of Technology | University of Gothenburg
SE-412 96 Göteborg,
Sweden
Phone: +46(0)31 772 1000

Printed by Chalmers Digitaltryck,
Gothenburg, Sweden 2026.

路漫漫其修远兮，吾将上下而求索。
- 屈原《离骚》

Long, long had been my road and far, far was the journey; I would go up and down to seek my heart's desire. (David Hawkes trans.)
- Qu Yuan, Li Sao

Adaptive and Efficient Event Stream Processing through Relaxed Semantics

Jingyu Liu
Department of Computer Science and Engineering
Chalmers University of Technology | University of Gothenburg

Abstract

Stream processing has become a fundamental data processing mechanism in modern edge-to-cloud systems, enabling the transformation of continuous data streams into timely aggregated results and actionable insights, e.g., in traffic or energy consumption monitoring. In practice, deploying and executing streaming applications, referred to as continuous queries, is often challenged by fluctuating workloads, changing resource availability, and varying application requirements. For example, a traffic monitoring system running on a roadside device might observe a surge in events reported by cars and, as a result, might need to produce outputs more frequently to support timely control decisions, despite limited computational power. This necessitates that Stream Processing Engines (SPEs) – software systems for executing continuous queries on unbounded data streams – dynamically adjust their behavior at runtime rather than relying on fixed configurations. Yet existing SPEs offer limited support for such adaptive capabilities, and even core operations such as data aggregation are no exception.

This thesis addresses this limitation by exploring how stream Aggregates – stateful operators that summarize streaming data – can be relaxed at runtime to adaptively trade performance against resource consumption and output guarantees. On the one hand, stream Aggregate states can vary significantly and shift over time in response to variations in the incoming data, and not all states are accessed with equal frequency; therefore, those states updated infrequently could be compressed. However, deciding when and what to compress is not trivial, as the optimal trade-off between memory savings and processing overhead varies with workload. We thus design an on-demand compression mechanism that uses Reinforcement Learning (RL) to dynamically tune Aggregate state compression levels at runtime, balancing memory consumption and processing efficiency under a target latency threshold, and identify policies that balance feedback timeliness and learned quality. On the other hand, Aggregates are defined by two key parameters – window advance (the output frequency) and window size (the interval length) – which typically remain fixed throughout execution, limiting their ability to adapt to changing workloads and resource conditions. Therefore, we support dynamic reconfiguration of both parameters at runtime, allowing analysts to specify a set of acceptable advances and sizes to trade performance and guarantees via weaker (at-most-once) or stronger (exactly-once) reconfiguration semantics. We provide alternative reconfiguration strategies for both in-order and out-of-order tuple arrival models, and empirically show that the approach matches fixed-configuration baselines' performance while supporting reconfiguration in negligible time.

Keywords
Stream Processing, Stream Aggregates, Relaxation

List of Publications

Appended publications

This thesis is based on the following publications:

[**Paper I**] **J. Liu**, V. Gulisano, *On-demand Memory Compression of Stream Aggregates through Reinforcement Learning.*
Proceedings of the 16th ACM/SPEC International Conference on Performance Engineering, 2025, pp. 240-252.

[**Paper II**] **J. Liu**, V. Gulisano, *Chill: Runtime Reconfiguration of Windowed Stream Aggregates with Semantic Guarantees.*
Proceedings of the 20th ACM International Conference on Distributed and Event-based Systems. 2026, pp. 157-168.

Acknowledgments

First of all, thanks to my supervisor, Vincenzo Gulisano, who gave me the opportunity to pursue a PhD at a time when I knew nothing about stream processing. Thanks for your constant encouragement over the years. Especially during those first few months, when I felt overwhelmed by the unknowns of this new field, you encouraged me by sharing your own experiences, and your patience and guidance since then have left a deep impression on me. I also thank my co-supervisors, Marina and Philippas, for the discussions and feedback.

I am grateful to Eleni for serving as my discussion leader, and to Ioannis for his support as my examiner.

Thanks (alphabetically) Kåre, Martin, Vinh, and Yixing for interesting chats and activities in the PhD, and everyone I met at Chalmers who is sincere and kind.

Thanks everyone in NCCC Gothenburg, especially (alphabetically) Eddie, Emma, Hannah, Iris, Joshua, Madeleline, Sam, Sarah, Winnie in ALIYA for every time we have gathered, and a special thanks to Yixin, who always listens attentively to my concerns about the work especially when I just started my PhD, and offers a lot of advice and guidance on both living and working.

Finally, to the people that I love and care about. Thank you to my parents for giving me a carefree life and warm home, and for giving me the freedom to do whatever I want. To Jacob, people are meant to meet. Thanks for every day we have spent together, it has been very fun and happy.

This work and the enclosed publications have been 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.

Contents

Abstract iii

List of Publications v

Acknowledgment vii

1 Overview 1
- 1.1 Relaxation Yields Richer Insights . . . 1
- 1.2 Research Problems . . . 6
- 1.3 Preliminaries . . . 7
  - 1.3.1 Stream Processing . . . 7
  - 1.3.2 Stream Aggregates . . . 8
  - 1.3.3 Window Execution Strategies . . . 9
  - 1.3.4 Reinforcement Learning . . . 10
- 1.4 Thesis Contributions . . . 12
  - 1.4.1 Adaptive Aggregate Compression . . . 12
  - 1.4.2 Runtime Guarantees for Aggregate Semantics . . . 13
- 1.5 Conclusion and Outlook . . . 14

2 On-demand Memory Compression of Stream Aggregates through Reinforcement Learning 17
- 2.1 Introduction . . . 20
  - 2.1.1 Challenges . . . 20
  - 2.1.2 Contributions . . . 21
- 2.2 Preliminaries . . . 21
  - 2.2.1 Stream Processing/Stream Aggregates . . . 21
  - 2.2.2 Reinforcement learning . . . 23
  - 2.2.3 Aggregation/Compression Algorithms . . . 24
- 2.3 Problem Formalization . . . 24
- 2.4 Wallclock/event time and metrics alignment . . . 26
- 2.5 Reinforcement Learning Model . . . 29
- 2.6 Learning Framework . . . 32
- 2.7 Evaluation . . . 35
- 2.8 Related Work . . . 42
- 2.9 Conclusions and Future Work . . . 43

3 Chill: Runtime Reconfiguration of Windowed Stream Aggregates with Semantic Guarantees 45
3.1 Introduction 48
3.2 Preliminaries 50
3.3 Problem Formalization 52
3.4 Algorithms 54
3.5 Evaluation 59
3.6 Related Work 70
3.7 Conclusions and Future Work 72

A Symbols and Abbreviations 73

Bibliography 77

List of Figures

1.1 Memory footprint of a stream Aggregate, measured in terms of window instances (Active Windows) maintained by the Aggregate under varying workload dynamics. 4
1.2 Computation cost under different window advances and window sizes. 5

2.1 Sample Aggregate A computing the average per-vehicle speed from their position reports over a window of W_{Advn} and W_{Size} of 15 and 60 minutes, respectively. The execution excerpt shows how the Aggregate functions contribute to processing input tuples, maintaining A's window instances, and producing output tuples. 21
2.2 Throughput, latency, and CPU consumption metrics reported by a sample Aggregate A during a period in which a change in D triggers a transition (from seconds 10.5 to 11.5). 27
2.3 Throughput, latency, and CPU consumption values reported in the state measured after a transition triggered by a change in D depending on the chosen policy. 28
2.4 Architecture of the proposed framework. 33
2.5 Main steps performed by the framework during the training/execution of an Agent. 34
2.6 Agents performance vs. input rate, and Agents vs. baselines performance for n/c ratio, latency, and CPU consumption and different state reporting policies – Linear Road. 38
2.7 X values resulting from the actions taken by the Agent (Linear Road, L-OB). The moving average – dashed line – is added to better appreciate the actions' fluctuations. 39
2.8 Agents performance vs. input rate, and Agents vs. baselines performance for n/c ratio, latency, and CPU consumption and different state reporting policies – Synthetic. 40
2.9 State initialization times (Linear Road/Synthetic). 41
2.10 CPU/latency (top) and overheads of the RL framework measured as differences for Linear Road (LR)/Synthetic (S) Aggregates alone vs. connected to the Agent (bottom). 42

3.1 Computation cost under different advances and sizes. 48
3.2 Sample A building on Example-Part 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 ps, and produce output tuples by merging the partial aggregates of all ps within the same γ. . . . 52
3.3 Window reconfiguration overhead for Synthetic. 64
3.4 Window reconfiguration overhead for Linear Road. 65
3.5 Performance impact of different $|\mathbb{WA} \times \mathbb{WS}|$ for Synthetic. . . . 66
3.6 Performance impact of different $|\mathbb{WA} \times \mathbb{WS}|$ for Linear Road. . 67
3.7 Violin plots of Throughput, Latency, and CPU across baselines and A^* variants for Synthetic. 69
3.8 Violin plots of Throughput, Latency, and CPU across baselines and A^* variants for Linear Road. 70

List of Tables

2.1 Values of the metrics contained in the next reported state based on the chosen state reporting policy. 29

A.1 Symbols and abbreviations used in the thesis. 73

Chapter 1

Overview

1.1 Relaxation Yields Richer Insights

Modern digital systems continuously generate large volumes of data from both software and hardware sources, across a wide range of devices – from resource-constrained edge devices to more powerful cloud nodes. These continuous flows of data, commonly referred to as *data streams*, are prevalent across multiple domains, including network traffic monitoring [1], [2], [3], financial time series analysis [4], sensor-based systems [5], [6], [7], telecommunications [8], smart grids [9], [10], and vehicular networks [11], [12], [13], [14]. Telecommunications systems, for example, generate vast amounts of operational events every day, all of which must be processed in a timely manner to support network management, fault detection, and quality of service monitoring [8]. In smart-grid environments, geographically distributed sensing devices continuously report measurements that have to be processed on time to support monitoring and control [9]. Similarly, modern vehicle systems rely on a large number of on-board sensors, which continuously monitor changing physical conditions during operation, thereby producing huge amounts of data relevant to traffic monitoring, safety, and coordination [11].

These continuous data streams share the following key characteristics:

- *High arrival rate.* They arrive rapidly and continuously, often at very high rates, and thus demand processing mechanisms that are lightweight and fast enough to support timely analysis and decision-making. For instance, in vehicular networks, continuously produced driving data might need to be processed quickly so that the system can maintain an up-to-date view of road conditions and respond in time [13], [15].

- *Large volume.* Data streams, by definition, are unbounded; therefore, it is impossible to store them in their entirety, regardless of the available resources. Furthermore, the incoming data volume is often so large that

it is difficult to retain a large portion of it, particularly in resource-constrained environments. In network measurement settings, for example, switches and other devices continuously monitor a continuous stream of data packets; the massive volume quickly renders complete storage impractical, and the system can not keep pace by storing every record in full or retrieving it retrospectively. Hence, data often needs to be processed sequentially, in one pass, as it arrives [1], [2], [3].

- *Temporal variability.* The input rate and data distribution are not static – they could vary noticeably over time and are often difficult to predict. Consider, for instance, a vehicle traffic network where traffic flows differ vastly between peak and off-peak hours. This temporal variability implies that a processing configuration that performs well under certain conditions may become inefficient in others. Accordingly, there is a need for stream processing systems capable of adapting their behaviors at runtime, rather than relying on fixed configurations set at deployment.

Together, these characteristics illustrate a common reality across edge-to-cloud systems: many applications should process high-rate, continuous, and time-varying data streams under both latency and resource constraints, whilst adapting flexibly as conditions change.

A key mechanism for processing such data streams is the use of *Stream Aggregates*, or simply *Aggregates*. They are a fundamental building block of modern edge-to-cloud data pipelines capable of transforming continuous raw data streams into timely summaries and actionable insights. In domains such as smart grids [10] and vehicular networks [12], [14], they are typically executed by *Stream Processing Engines (SPEs)* to summarize data within windows and periodically produce outputs for monitoring, decision-making, and control. Such windows are typically governed by two parameters: *window advance*, which determines the output frequency, and *window size*, which determines the interval length.

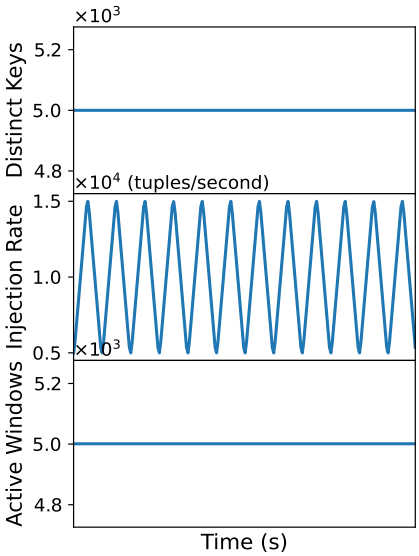
Depending on Aggregates' computational footprint and the available resources at each node, Aggregates may be deployed at different points in the pipeline. In data-center environments, SPEs typically handle time-varying workloads by executing tasks in a distributed mode across multiple nodes or by leveraging parallelism within a single machine. However, in edge computing environments, neither of these approaches can be readily adopted: operators cannot easily migrate between nodes, and the limited number of physical cores makes it difficult for parallel processing to absorb workload fluctuations effectively. In addition to these limitations, existing SPEs also tend to treat Aggregates as essentially static components, with key aspects of their behavior fixed before execution begins and remaining unchanged throughout the entire runtime. This design constrains systems to commit to a single operating state – for instance, a fixed window advance and a fixed window size – in advance, even though the optimal trade-off between resource consumption, output freshness, and processing guarantees may vary during execution. If non-static Aggregates are to be applied to SPEs, it would be intuitive to achieve reconfiguration simply and directly: by terminating the running Aggregate and restarting it

with the new configuration. This blunt approach, however, is not only extremely inefficient but also semantically unsafe. It incurs non-negligible downtime, and more critically, might break correctness – tuple loss may occur without notice, and the scope of such losses is difficult to predict. For applications that rely on reliable and timely results, this might be unacceptable.

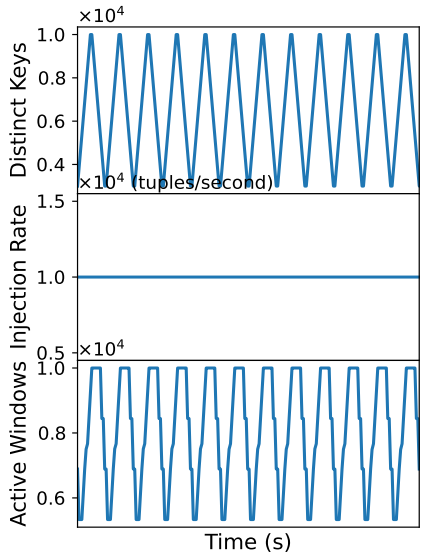
This motivates establishing mechanisms that can efficiently manage Aggregate resource usage while supporting safe and correct principled reconfiguration. Achieving this requires particular attention to two aspects of Aggregate behavior, which are especially critical in edge deployments:

Memory footprint. Aggregates may consume substantial memory to store per-key window aggregation states, especially when processing high-rate streams, large windows, or many keys. Here, a window refers to a finite interval into which an input stream is divided for computation [16], and a key refers to a distinct value extracted from data to logically partition the data stream [17]. Since not all per-key window states are accessed or updated with the same frequency – some keys remain active and are updated regularly while others remain inactive for long periods – an intuitive approach is to compress the state of infrequently accessed keys, thereby reclaiming memory without discarding data. However, deciding when and what to compress is far from trivial, as it requires predicting how the memory footprint of Aggregates will change based on observable workload characteristics, such as injection rate or key distribution. One might intuitively assume that the active window count – a direct indicator of the memory required to maintain the aggregated state – is directly proportional to the injection rate; that is, the higher the rate, the more windows. Figure 1.1, however, shows that the actual situation is far more complex: the number of active windows is influenced by the interplay of several workload characteristics, rather than being dominated by any single factor. As shown in Figure 1.1(a) and Figure 1.1(b), fluctuations in the active window count do not necessarily correspond to variations in the injection rate: varying the injection rate alone does not cause the active window count to change, since the rate might only affect how frequently each key receives new data, not how many distinct keys are being tracked. In contrast, varying the number of distinct keys causes it to correlate closely with the key count, as each key typically maintains its own independent window. More notably, Figure 1.1(c) shows that even when both the injection rate and total number of keys remain constant, the active window count still fluctuates, as the set of active keys shifts over time: some keys expire, whilst others emerge. This demonstrates that Aggregate memory pressure arises from the interaction of several distinct factors, and it is challenging to predict or control it through static configuration alone. Furthermore, existing SPEs offer limited support for this selective runtime memory management, making the system prone to becoming unresponsive when memory pressure increases, or available resources are reduced.

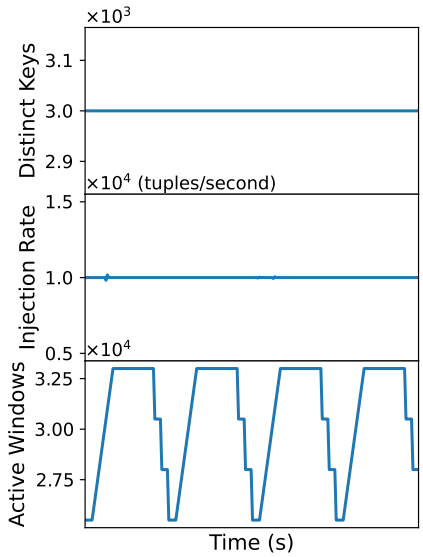
Window configuration. An alternative approach to relaxing Aggregates'



(a) Varying injection rate, fixed key set.



(b) Fixed injection rate, varying key set.



(c) Fixed injection rate, fixed number of distinct keys with key shifting.

Figure 1.1: Memory footprint of a stream Aggregate, measured in terms of window instances (Active Windows) maintained by the Aggregate under varying workload dynamics.

states handling – particularly suitable when selective compression is not desirable, for instance when all states are equally important or require

uniform processing – is to adjust the window configuration, namely window advance and window size, adaptively at runtime. These parameters, however, are typically fixed in Aggregates, even though changing workload and resource conditions may call for different window configurations over time [18], [19], [20], [21], [22]. For example, as shown in Figure 1.2, a larger window advance paired with a smaller window size results in infrequent data updates and a limited data range, leading to stale results. Smaller window advances and window sizes frequently capture short-term patterns, but the insights might be limited or noisy due to less data; conversely, larger window advances and window sizes capture a wider range, but infrequent updates lead to late results. A small window advance and a larger window size provide the most up-to-date results, but at a high computational cost. Since these parameters directly affect output frequency, retained state, computational cost, and temporal semantics, keeping them fixed throughout execution prevents the system from adapting effectively to changing runtime needs. Beyond deciding on the right configuration, changing window advance and window size at runtime introduces an additional task: unlike compression, which does not change what is being computed, reconfiguring window parameters does change the computational content of the aggregation and how results are produced. This raises the question of how such reconfiguration can be performed with correct and principled guarantees regarding the semantics of the results.

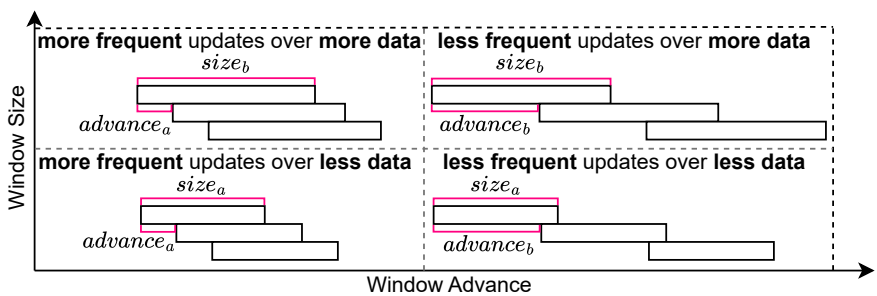


Figure 1.2: Computation cost under different window advances and window sizes.

These two techniques – compression and window reconfiguration – differ fundamentally in principle: compressing the Aggregate state reduces memory consumption without changing the correctness semantics, while reconfiguring window parameters affects the correctness of the aggregation itself.

Building on this distinction, this thesis investigates how stream Aggregates can adapt to changing operating conditions by dynamically managing their state to control memory consumption, and by reconfiguring window parameters at runtime to trade performance against guarantees through weaker (at-most-once) and stronger (exactly-once) reconfiguration semantics.

1.2 Research Problems

As explained in Section 1.1, the challenge is how to enable stream Aggregates to adapt at runtime while relaxing memory usage and window configurations. In that context, this thesis raises two main questions that guide the work in the appended papers. The first concerns the computational footprint of an Aggregate: how can its state be managed more efficiently at runtime, so as to reduce memory consumption while keeping processing overhead under control? The second pertains to the semantics of the Aggregate: how can window parameters be reconfigured at runtime with explicit correctness guarantees?

Research Question 1:

"How to live-tune the Aggregates' computational footprints by trading off memory consumption for computational efficiency under a latency threshold?"

The first question is the problem of runtime memory adaptation. Aggregates often maintain a large state, especially when summarizing high-volume streams or long time spans, e.g., in traffic monitoring applications. This can result in state entries spanning tens or even hundreds of thousands of different keys (see Section 1.1). However, not all keys are active; some of them are updated frequently, while others are not. This asymmetry gives a chance for compression – compressing the state of less frequently updated keys to reclaim memory without discarding the data entirely. In constrained environments, this memory footprint reduction can be essential to free resources for high-priority tasks, fit execution within device limits, or support state migration across nodes. However, deciding when and what to compress is not straightforward, as compression introduces overhead, such as latency, and the optimal trade-off between memory savings and processing efficiency varies with workload.

To address this challenge, this thesis introduces an on-demand adaptive memory compression scheme for stream Aggregates. The proposed approach uses Reinforcement Learning (RL) to dynamically tune the compression level of the Aggregate state during execution, balancing memory consumption and processing efficiency under a desired latency threshold. A key challenge is that the effects of compression decisions may only become visible over time, creating a trade-off between fast but incomplete feedback and slower but more reliable feedback during training. This thesis studies the trade-off and defines policies for training RL agents effectively in live SPE environments.

Research Question 2:

"Can computational costs be adapted at runtime by allowing analyst-defined families of acceptable window configurations with varying computational demands, and by enabling efficient switching between them with clear guarantees?"

The second question is about runtime window adaptation. In existing systems, the window parameters – window advance and window size – are

usually fixed before execution starts [18], [19], [20], [21], [22]. Yet, in practical deployments, analysts may wish to trade output freshness, computation cost, and semantics guarantees differently over time as workload and resource conditions change.

To overcome this limitation, this thesis explores how Aggregates can support the dynamic reconfiguration of window parameters at runtime. Rather than committing to a single window configuration, analysts can define a family of acceptable window advances and window sizes, and the system can switch among them during execution. To achieve this efficiently, it is necessary to redesign how the Aggregate state is maintained internally. Building on pane-based aggregation (see Section 1.3.3), which decouples state maintenance from the window itself, this thesis presents algorithms that enable such switching efficiently and with well-defined guarantees. Also, it studies reconfiguration strategies with varying correctness guarantees, tailored to support the diverse aggregation types and execution models offered by state-of-the-art SPEs.

1.3 Preliminaries

This section covers topics and techniques that are important for understanding the approaches toward Research Questions 1 and 2 outlined in Section 1.2 and that ease the understanding of the appended publications (see Chapter 2 and Chapter 3).

1.3.1 Stream Processing

A *stream* S [1] is an unbounded sequence of *tuples* [16]. Each tuple t carries a timestamp $t.\tau$ – the *event time* set by the source when the event occurred, expressed in time units from a given epoch that progress in SPE-specific δ increments (e.g., milliseconds) [23] – and a payload $t.\phi$ – the application-specific data associated with the event (e.g., a sensor reading or a network packet). Stream processing *queries* are composed of *sources*, *operators*, and *sinks*. Sources forward tuples (e.g., sensor-reported events) to operators. Operators, connected as a *Directed Acyclic Graph (DAG)*, process incoming tuples, forward and produce new ones. Eventually, tuples are delivered to sinks, which expose results to end-users or downstream applications. Operators can be either *stateless* or *stateful*. A stateless operator, as the name indicates, does not maintain any state across tuples. In other words, its output depends only on the tuple currently being processed and not on previously observed data. In contrast to stateless operators, a stateful operator creates and maintains state as it processes the incoming tuples. Such a state, along with the operator's logic for updating it as tuples arrive, affects the results the operator produces. A typical stateful operator is the Aggregate operator, for example, a windowed count or

[1] Chapter 1 uses conventional symbols to introduce key concepts. Chapter 2 and Chapter 3 retain the symbols used in their proceedings, which are largely consistent with those used in Chapter 1. Table A.1 in Appendix A summarizes all symbols in the thesis, highlighting cases where the same concept is denoted differently across Chapter 2 and Chapter 3.

sum; such an operator must maintain intermediate results (e.g., partial counts or sums) across multiple tuples. Computing such results, whether counting events, summing values, or performing any other stateful computations within a sliding time window (see Section 1.3.2) requires storing and updating state as new tuples arrive and old ones expire, making the output depend not only on the current tuple but also on previously processed data.

When a query is deployed within an SPE, to support parallel processing, 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 where each instance has a dedicated state exclusively managed by one, without sharing memory with other instances [23], [24]. 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.

1.3.2 Stream Aggregates

In this thesis, we focus on Aggregates, stateful operators defined over *time-based windows* (or simply *windows*) which are commonly provided by SPEs [23], [24], [25]. An Aggregate A is defined by the following components:

Window Advance (wa) and Window Size (ws): which determine the event time intervals $[m\,wa, m\,wa + ws)$, with $m \in \mathbb{N}$, $wa > 0$, and $ws > 0$, over which A summarizes data. Each such interval is referred to as a *window instance* γ, in which $\gamma.l$ and $\gamma.r$ denote its left and right boundaries, respectively. If $wa = ws$, γs are *tumbling* windows, and each tuple falls into exactly one instance. If $wa < ws$, consecutive γs overlap, yielding *sliding* windows; in this case, a tuple can contribute to several γs.

Key-by Function $f_K(t)$: which can be used to extract a key from a tuple t, allowing A to maintain separate window instances for tuples sharing the same key. Although key-by is optional in [23], [24], from a modeling perspective, $f_K(t)$ can always be defined, returning the same value for all tuples when no explicit key partitioning is required.

From an operational perspective, wa controls how frequently outputs are produced, while ws determines how much event-time history is summarized; together, they affect the Aggregate's computational cost and state footprint.

As discussed above, when deployed in an SPE, a user-defined Aggregate is typically instantiated as multiple A instances that can execute in parallel, potentially distributed across nodes. In shared-nothing environments, each instance maintains its own local state in dedicated memory, and processes a portion of the key space of the input stream [23], [24].

In state-of-the-art SPEs [23], [26], when an output tuple t_o is produced for a γ, its timestamp is typically set to the maximum event time covered by that window, namely $t_o.\tau = \gamma.r - \delta$. Hence, the event time of each output tuple takes a value of the form $n\,wa + ws - \delta$ (for integer n). If event time

progresses at approximately the same pace as wall-clock time, then one output is produced roughly every wa time units in both event time and wall-clock time.

To decide when the correct result for a given γ can be produced – that is, a result accounting for all of γ's tuples – an SPE must know when no more tuples falling into γ are still to be processed. This can be ensured in two ways:

(1) If the SPE assumes timestamp-sorted streams [24], [27], correctness of the output of a γ is guaranteed as soon as a tuple t with $t.\tau \geq \gamma.r$ is observed.

(2) Alternatively, if the SPE supports out-of-order streams, it can rely on *watermarks* [23], [28], special tuples that signal event-time progress from each source's perspective. More precisely, W_A^ω, the watermark of A at wall-clock time ω, represents the earliest event time any future tuple t processed by A can have from ω onward; that is, for every such tuple t, it holds that $t.\tau \geq W_A^\omega$. Sources periodically emit watermarks as special tuples with monotonically increasing timestamps [23], [28], indicating event-time progress from each source's perspective. Upon receiving a watermark W, A records it and updates W_A^ω to the minimum of the latest watermarks observed on all its input streams. When W_A^ω increases, A can safely output all windows whose right boundary is not greater than W_A^ω, i.e., all γ such that $\gamma.r \leq W_A^\omega$, and then forward the updated W_A^ω downstream.

In the remainder of this thesis, we refer to case (1) as the timestamp-based (TB) implementation, and to case (2) as the watermark-based (WB) implementation.

1.3.3 Window Execution Strategies

In addition to the logical definition of A via its window parameters, window advance (wa) and window size (ws), and user-defined functions, as described in Section 1.3.2, another important aspect is the internal execution of A. In practice, as tuples continue to arrive, windows advance, and output results are generated periodically, A must maintain the state of each γ. The organization, updating, and eventual use of this state to produce outputs depend on A's execution strategy [29]. This strategy directly impacts memory consumption, computational overhead, result production, and implementation complexity and is therefore a key design choice in SPEs. The main execution strategies discussed in the literature include single-window, multi-window, and pane-based (also referred to as slicing [30]) execution.

A provides a generic interface centered on three methods: `add(`t`)` incorporates the contribution of t into the state being maintained; `get()` is used to return the current (partial or complete) aggregation result; and, where applicable, `evict(`t`)` removes the contribution of t once it no longer belongs to the relevant γ.

Single-window. It maintains one accumulator per key per active window, where `add` updates the accumulator for γ, `get` returns its current result, and

`evict` removes the contribution of tuples that no longer belong to the γ after it has advanced. This strategy is straightforward and strictly adheres to the logical definition of windowed aggregation, as each maintained state object corresponds directly to a single γ.

Multi-window. It maintains one accumulator per key per overlapping γ to which each incoming t belongs, invoking `add(`t`)` on each corresponding accumulator. Each γ evolves independently and is discarded after producing its output through `get()`. This strategy naturally supports sliding windows and avoids the need for tuple eviction, as old data is removed by deleting completed γs rather than by updating long-lived state.

Pane-based. It does not maintain state directly per γ, but instead divides the event time into a series of consecutive, non-overlapping intervals known as *panes* (denoted as *p*s), whose length is derived from the window configuration – that is, from the wa and ws values in use – and can therefore vary across different configurations. Each tuple belongs to exactly one pane, and A maintains a partial aggregate for each p and key. Consequently, a γ is represented as a sequence of consecutive *p*s. In this strategy, A does not repeatedly update the entire γs; instead, it updates partition-level state via `add(`t`)`; retrieves partition-level partial results through `get()`; and merges them through a `merge()` method when the window output needs to be produced. The primary advantage of pane-based execution lies in its ability to reduce redundant computations when overlapping γs share a common time interval. Since each t contributes only once in the p, repeated updates across multiple overlapping γs are avoided. This makes the strategy particularly suitable when there is a high degree of window overlap or when partial results can be reused across different configurations.

Each execution strategy involves an inherent trade-off between memory consumption and computational overhead, and which strategy performs better for a given Aggregate depends on factors such as the window configuration, the aggregation function, and the tuple arrival rate [29]. Single-window requires evicting tuples that no longer belong to the current γ. Multi-window avoids eviction by discarding completed γs entirely, but may require maintaining the state of a large number of concurrently active γs. The pane-based approach reduces redundant computations by sharing partial results among γs, but increases the complexity of managing p boundaries and merging partial results.

1.3.4 Reinforcement Learning

Machine Learning broadly comprises three paradigms: supervised learning – learning from labelled data, unsupervised learning – discovering patterns in unlabelled data, and RL – learning through interaction and feedback. Unlike the first two, which learn from pre-collected data, RL involves learning through direct interaction with the environment: an *agent* – a learner – observes a *state*, which is a representation of the environment's condition at a specific time, and selects an *action* – a decision that the agent can take to change the state of the environment. In response, the agent receives a *reward* – a value indicating the quality of the outcome. Through repeated interactions, the agent gradually optimizes its behavior to maximize the long-term *cumulative reward*, that is,

the sum of rewards accumulated over time [31].

In the CartPole problem [32], for example, an agent learns how to keep a pole balanced on a moving cart by applying forces to the left or right, receiving a reward for maintaining the pole upright for a certain period of time. Without any prior knowledge of the underlying physical principles, it learns entirely through trial and error – trying different actions, observing the outcomes, and adjusting its behavior accordingly.

A key challenge lies in the fact that the environment faced by the agent may change over time, and its state transitions are typically unknown in advance; therefore, the agent cannot rely on explicit models of these transitions but has to discover effective strategies solely through experience.

The overall process of RL is commonly formulated as a *Markov Decision Process (MDP)* [33], [34], [35], usually described by a quadruple $\langle \mathbb{S}; \mathbb{A}; T; R \rangle$. Here, $\mathbb{S} = \{s\}$ denotes the set of states, $\mathbb{A} = \{a\}$ denotes the set of all available actions the agent can perform, $T(s_k, a_k, s_{k+1}) \sim p(s_{k+1}|s_k, a_k)$ denotes the state transition function, which describes the probability distribution of the agent transitioning to the next state s_{k+1} after executing action a_k from the current state s_k in an interaction step k; while $R : s_k \times a_k \to r_k$ is the reward function provided by the environment, i.e., the immediate return or reward obtained after taking action a_k in state s_k. Furthermore, an MDP is usually associated with a discount factor γ_{RL}, where $0 < \gamma_{RL} < 1$, which determines the extent to which future rewards are discounted relative to immediate ones.

In each interaction step k, the agent first observes the current state $s_k \in \mathbb{S}$. It then selects an action $a_k \in \mathbb{A}$ according to its policy and applies it to the environment. In response, the environment moves to a new state s_{k+1} via the state transition function $T(\cdot)$ and returns a reward r_k to the agent. Repeated interactions, therefore, generate a sequence of states, actions, and rewards, often referred to as an *episode*. From these episodes, the agent learns policies that increase its long-term expected return.

Formally, a *policy* $\pi \sim p(a_k|s_k)$ takes the current state s_k as input and outputs a probability distribution $\pi : \mathbb{S} \times \mathbb{A} \to [0, 1]$ over the agent's actions, such that the cumulative reward G_k over K steps is maximized. Therefore, the agent's optimal policy π^* can be expressed as: $\pi^* = \text{argmax}_\pi \mathbb{E}[G_k] = \text{argmax}_\pi \mathbb{E}\left[\sum_{i=0}^{K} \gamma_{RL}^i r_{k+i}\right]$. In order to find this optimal policy π^*, the agent needs a way to evaluate the quality of taking a specific action in a given state. This is represented by the state-action function $Q_\pi = \mathbb{E}[G_k|s = s_k, a = a_k]$, which denotes the expected cumulative reward the agent receives when taking action a in state s and subsequently following policy π. It can be expressed recursively via the Bellman equation:

$$Q_\pi(s, a) = \sum_{s', r} p(s', r|s, a) \left[r + \gamma_{RL} \sum_{a'} \pi(a'|s') Q_\pi(s', a') \right]. \tag{1.1}$$

And the optimal state-action function $Q^*(s, a)$ is defined as:

$$Q^*(s, a) = \sum_{s', r} p(s', r|s, a) \left[r + \gamma_{RL} \max_{a'} Q^*(s', a') \right]. \tag{1.2}$$

Once $Q^*(s, a)$ is known, the optimal strategy becomes apparent – in each state, the agent simply needs to choose the action with the highest Q value. However, in practical applications, $Q^*(s, a)$ cannot be computed analytically, as this would require knowledge of the state transition probabilities $p(s', r|s, a)$, which are often difficult to obtain; consequently, it can only be estimated through experience.

One of the most common methods for estimating the state-action function is *Temporal Difference (TD)* learning [36]. It does not wait until the end of the episode to update the estimate. Instead, it updates the state-action function step by step after each interaction, using the difference between the current estimate and an updated estimate based on the observed reward and the next state. *Q-learning* [37] is a widely used RL algorithm that is based on TD learning and approximates the optimal state-action function $Q^*(s, a)$ by maintaining a table of estimates for every state-action pair, iteratively updated via the Bellman optimality equation:

$$Q(s_k, a_k) \leftarrow Q(s_k, a_k) + \alpha \left[r_{k+1} + \gamma_{RL} \max_a Q(s_{k+1}, a) - Q(s_k, a_k) \right], \quad (1.3)$$

where α is the learning rate, which controls the step length of each update. The term in brackets – the difference between the target value $r_{k+1} + \gamma_{RL} \max_a Q(s_{k+1}, a)$ and the current estimate $Q(s_k, a_k)$ – is the TD error, which determines the direction and magnitude of each update. However, when the state space or action space is large or continuous, this tabular approach becomes inefficient, as it requires maintaining separate estimates for each state-action pair. To overcome this limitation, in Chapter 2 we employ the *Deep Q-network (DQN)* [38], a widely used extension of Q-learning that utilizes neural networks to approximate $Q^*(s, a)$.

1.4 Thesis Contributions

Having introduced key preliminary topics and techniques (see Section 1.3), I highlight in the following the main contributions of this thesis.

1.4.1 Adaptive Aggregate Compression

In Chapter 2, Research Question 1, as posed in Section 1.2, is approached: *"How to live-tune the Aggregates' computational footprints by memory consumption for computational efficiency under a latency threshold?"*. We address this question by proposing an RL-based approach designed to dynamically tune aggregate state compression in response to changing execution conditions, intending to reduce memory consumption whilst controlling processing latency. The core idea behind this approach is to view compression not as a one-off, workload-independent decision, but as an ongoing control problem: since the impact of compressions on latency and memory may take some time to become apparent, the agent has to learn through trial and error to determine the appropriate level of compression to apply under different and constantly changing workload conditions, rather than relying on a fixed compression strategy. To support this,

the thesis constructs a model, a training policy, and an experimental framework to investigate the trade-offs involved in real-time adaptive memory compression. These elements are then evaluated through both a benchmark-based and a synthetic study:

(1) using the Linear Road benchmark [39], the thesis evaluates the different state reporting policies – that is, strategies determining how and when the observed states of a RL agent should be reported during training, to balance the trade-off between the timeliness of training feedback and the quality of the learned behavior – as well as the RL model, and the RL environment in a real-world streaming application;

(2) using a complementary synthetic usecase with more extreme input rates and data distribution variability, the thesis investigates the approach's robustness under more challenging and dynamic execution conditions;

(3) finally, the thesis also analyzes the time required to initialize the Aggregate state at the beginning of training episodes, as well as the CPU and memory overhead introduced by the Agent and by the framework used to exchange states, actions, and rewards during execution, to verify the approach's scalability and practical viability.

Together, these evaluations provide a quantifiable answer to the Research Question 1 by demonstrating that the computational footprint of a stream Aggregate can be live-tuned through adaptive memory compression, thereby reducing memory usage whilst keeping processing latency within acceptable bounds.

1.4.2 Runtime Guarantees for Aggregate Semantics

In Chapter 3, an approach to Research Question 2 is presented: *"Can computational costs be adapted at runtime by allowing analyst-defined families of acceptable window configurations with varying computational demands, and by enabling efficient switching between them with clear guarantees?"*. We investigate how stream Aggregates adapt their external aggregation semantics at runtime through dynamic window reconfiguration. The key idea is to decouple state maintenance from the window itself rather than storing accumulators separately for each window. Pane-based execution maintains state at the level of non-overlapping panes, which are shared across overlapping windows without incurring redundant computations. Consequently, changing window parameters does not require restructuring the maintained state; it only changes the combination of existing panes to output the result. However, whether this restructuring can reuse existing partition-level state depends on whether pane boundaries are guaranteed to align across different configurations. This gives rise to two distinct reconfiguration semantics with different trade-offs between correctness and efficiency. Specifically, we present algorithms that support changing window advance and window size during execution, allowing analysts to rebalance output freshness, computational cost, and semantic guarantees as conditions change. The thesis further studies alternative reconfiguration

strategies under different execution models and evaluates the trade-offs between weaker (at-most-once) and stronger (exactly-once) guarantees. The proposed approach is evaluated across three main dimensions:

(1) it analyzes the overhead of runtime window reconfiguration, showing that reconfiguration time is negligible and that convergence behavior remains predictable across transitions;

(2) it studies the impact of expanding the window configuration space, highlighting how different semantics expose distinct trade-offs between performance and correctness, with at-most-once favoring higher efficiency and exactly-once preserving correctness at a modest and predictable cost;

(3) it compares the proposed solution against baselines across both the Linear Road benchmark [39] and synthetic workloads designed to stress-test the solution under extreme conditions. The results demonstrate that the approach remains competitive across scenarios and outperforms baselines in several cases.

Together, these evaluations demonstrate that stream Aggregates can indeed adapt their window parameters at runtime, thereby achieving a dynamic balance among freshness, computational cost, and correctness guarantees without sacrificing practical efficiency, thus answering the Research Question 2.

1.5 Conclusion and Outlook

This thesis explores two complementary directions: adjusting the computational footprint of Aggregate through memory compression, and adapting its aggregation semantics through runtime window reconfiguration, to support *Adaptive and Efficient Event Stream Processing through Relaxed Semantics*. In Chapter 2, the thesis shows an RL-based scheme in which an Agent can be activated on a live Aggregate to adjust its memory compression while dynamically adhering to a latency threshold. Chapter 3 introduces Chill, an approach that supports runtime reconfiguration of both window advance and window size by a pane-based design that enables efficient state reuse across reconfigurations and supports both weaker (at-most-once) and stronger (exactly-once) semantics, allowing analysts to trade off between performance and stronger correctness guarantees as requirements evolve.

In ongoing and future work, we could further explore the following ways to enhance efficient stream processing.

Multi-Objective Compression Tuning. The adaptive memory compression (see Chapter 2) approach could be extended beyond latency-aware optimization to take into account other Quality-of-Service metrics, such as throughput, memory pressure, energy consumption, or a combination of multiple objectives. This would provide a more comprehensive basis for runtime adaptation in heterogeneous deployment environments.

Broader Adaptation Mechanisms for Aggregate State. The compression framework (see Chapter 2) could be generalized to support a wider range of

compression schemes and adaptive mechanisms. Further solutions will not be limited to selecting the degree of compression, but will also be able to learn when to combine different compression techniques, or when to adapt other aggregate properties, such as parallelism, placement, or load shedding.

Optimizing Learning Process. Transfer learning, pre-training, or hybrid live-and-simulated training could help reduce the time required for agents to adapt to new workloads, aggregate instances, or deployment conditions. More generally, future work could study how to improve sample efficiency and policy robustness in lifelong learning scenarios involving stream processing.

Wider Deployment Settings. Both the compression mechanism (see Chapter 2) and the window reconfiguration approach (see Chapter 3) could be extended to a wider range of deployment scenarios. This includes implementing the proposed techniques in widely used SPEs such as Apache Flink [23] and investigating multi-instance or distributed deployments under key partitioning or elastic scaling.

Learning-based Runtime Control of Window Semantics. Although Chill (see Chapter 3) opens the door to more advanced runtime control of window semantics, a promising direction is the design of learning-based or AI-driven controllers capable of choosing window advance, window size, and semantics online based on observed workload and resource signals.

Richer Reconfiguration Policies and Cost Models. It could be valuable to explore broader families of relaxed or hybrid correctness policies during reconfiguration, as well as cost models that guide when and how to carry out reconfiguration. Such models could help analysts to more clearly identify the trade-offs between performance, accuracy, and correctness guarantees.

The above directions point toward a broader research direction in which stream Aggregates are no longer viewed as fixed operators, but rather as adaptive components whose state management, semantics, and execution strategies can evolve dynamically at runtime. By extending them to encompass richer targets, broader adaptive mechanisms, more robust learning capabilities, and more realistic deployment environments, future research could move closer to stream processing systems capable of continuously, efficiently, and accurately adapting to operational environments.

Chapter 2

On-demand Memory Compression of Stream Aggregates through Reinforcement Learning

---

J. Liu, V. Gulisano

Proceedings of the 16th ACM/SPEC International Conference on Performance Engineering, 2025, pp. 240-252.

PAPER A

This is a formatted and adapted version of the paper published in *Proceedings of the 16th ACM/SPEC International Conference on Performance Engineering, 2025, pp. 240-252.* Any performed changes serve only to retain the consistency of this thesis and adapt to the layout.

Abstract

Stream Aggregates are crucial in digital infrastructures for transforming continuous data streams into actionable insights. However, state-of-the-art Stream Processing Engines lack mechanisms to effectively balance performance with memory consumption – a capability that is especially crucial in environments with fluctuating computational resources and data-intensive workloads.

This paper tackles this gap by introducing a novel on-demand adaptive memory compression scheme for stream Aggregates. Our approach uses Reinforcement Learning (RL) to dynamically adapt how a stream Aggregate compresses its state, balancing performance and memory utilization under a given processing latency threshold. We develop a model that incorporates the application- and data-specific nuances of stream Aggregates and create a framework to train RL Agents to adjust memory compression levels in real-time. Additionally, we shed light on a trade-off between the timeliness of an RL Agent training and its resulting behavior, defining several policies to account for this trade-off.

Through extensive evaluation, we show that the proposed RL Agent supports well on-demand memory compression. We also study the effects of our policies – providing guidance on their role in RL applied to stream Aggregates – and show our framework supports lean execution of such RL jobs.

2.1 Introduction

In distributed systems like Smart Grids [10] and Vehicular Networks [11], [14], stream Aggregates (or simply Aggregates) are used by Stream Processing Engines (SPEs) to analyze continuous streams of data, summarizing data and extracting insights over time windows (e.g., every hour). However, state-of-the-art SPEs are designed for data centers where hardware can be dynamically adjusted [23], [40]. In contrast, their deployment at the edge, with fixed and limited computational resources, presents significant challenges. A modern electric vehicle, e.g., might mount a powerful device [41] but give limited room to run custom Aggregates, since such a device must prioritize essential safety/driving functions and limit, to the extent possible, its battery consumption [42]. In this context, trading off performance for computational cost, such as reducing memory consumption via memory compression at the price of increased processing latency, enables the live-tuning of Aggregates' computational footprints by, e.g., lowering memory usage to prioritize higher-priority applications or migrate an Aggregate's state to another device.

2.1.1 Challenges

The study in [43] showed that when accounting for the evolving rates/distributions of the data fed to an Aggregate, such live-tuning would require continuous adjustments. Nonetheless, the study did not investigate such adjustments, which is our focus.

We hypothesize that the process of adaptively adjusting the memory compression of an Aggregate while accounting for a desired latency threshold can be automated through Reinforcement Learning (RL), which has been proposed for similar decision-making processes in relevant work such as [44], [45], [46], [47], [48]. For the problem at hand, RL would train an Agent that, based on the rewards it gets by acting on the SPE environment running an Aggregate, would learn to dynamically adapt the compression of the Aggregate's memory under a processing latency threshold. Designing an RL model for this problem is challenging. It is best to train the Agent on a live system to capture all the subtleties of the Aggregate's environment. In such a system, though, the effects of an action could take seconds or tens of seconds to manifest. It is thus key to understand the trade-offs between fast-paced rewards (i.e., faster feedback that may partially reflect an action's effects) and slow-paced rewards (i.e., slower but more comprehensive feedback on an action's effects). Each learning job, furthermore, would be made longer by the need to set up the Aggregate to be controlled, a task whose time is proportional to the size of the state maintained by such an Aggregate (e.g., an Aggregate maintaining the last hour of data would need to receive the data covering an hour to be in steady state). To the best of our knowledge, no model nor framework exists that can support the efficient and effective training of an Agent for the problem at hand.

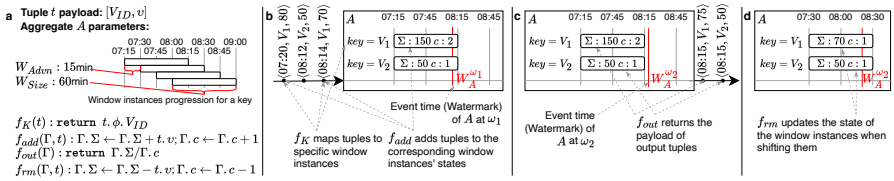


Figure 2.1: Sample Aggregate A computing the average per-vehicle speed from their position reports over a window of W_{Advn} and W_{Size} of 15 and 60 minutes, respectively. The execution excerpt shows how the Aggregate functions contribute to processing input tuples, maintaining A's window instances, and producing output tuples.

2.1.2 Contributions

This paper tackles these open challenges by introducing a novel on-demand adaptive memory compression scheme for stream Aggregates. We first introduce several policies for RL jobs that can be used to trade the timeliness with which an Agent can be trained with the fitness of its resulting behavior (e.g., leading to Agents applying conservative rather than aggressive compression actions). Then, we propose a complete model and also discuss the design of a framework that can be used for *lifelong* live training of Agents on top of an SPE. Finally, we provide an extensive evaluation of the proposed techniques based both on Linear Road [39], the de-facto benchmark for streaming applications, and on a Synthetic usecase designed to stress-test our proposed solution, and show how the resulting Agent is capable of offering the on-demand memory compression motivating our work. Our code is available at `https://github.com/appalachia-xx/compAgg`.

The rest of the paper is organized as follows. Section 2.2 introduces stream processing, RL, and the compression algorithm we use. Section 2.3 covers our problem formulation. Section 2.4 discusses the trade-offs in the timeliness with which states are shared with an Agent and the information carried by such states, introducing four different state reporting policies. Our RL model and learning framework are discussed in Section 2.5 and Section 2.6, respectively. Section 2.7 covers our empirical evaluation. We discuss related work in Section 2.8 and conclude in Section 2.9.

2.2 Preliminaries

2.2.1 Stream Processing/Stream Aggregates

According to DataFlow [16], a *stream* S is an unbounded sequence of *tuples*. Each tuple t carries a timestamp $t.\tau$ and a payload $t.\phi$ with application-specific attributes. Stream processing *queries* are composed of *sources*, *operators*, and *sinks*. Sources forward tuples (e.g., events reported by sensors) to operators. Operators connected in a Directed Acyclic Graph process and forward/produce

tuples. Eventually, tuples are fed to sinks, which deliver results to end-users or other applications. Operators are either *stateless* – they do not maintain a state that evolves with the tuples they process – or *stateful* – they produce results from a state dependent on one or more tuples.

When a query is deployed within an SPE, multiple copies of the same operator can be instantiated to analyze different portions of the operator's input stream(s). In state-of-the-art SPEs [23], [24], such instances run in shared-nothing environments. Hence, each instance of a stateful operator has a dedicated state exclusively managed by one thread. The evolution of such a state – i.e., the raw or aggregated tuples forming such a state and how such tuples are kept within the data structures maintained by the thread – progresses exclusively based on the tuples processed and produced by said thread.

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 δ increments (e.g., milliseconds [23]).

We focus on *Aggregates*, stateful operators defined over *time-based windows* (or simply windows) which are commonly provided by SPEs [23], [24], [25]. An Aggregate A is defined by:

Window Advance (W_{Advn}) and Window Size (W_{Size}): which determine the event time periods $[jW_{Advn}, jW_{Advn} + W_{Size})$, with $j \in \mathbb{N}$, covered by A. Each event time period is referred to as a window *instance* Γ. If $W_{Advn} < W_{Size}$, consecutive *sliding* window instances overlap and a tuple can fall into several window instances.

Key-by Function $f_K(t)$ (opt.): which can be used to extract a key from a tuple t and maintain dedicated window instances for tuples that share the same key.

Function $f_{add}(\Gamma, t)$ to add to window instance Γ the contribution of the input tuple t.

Function $f_{out}(\Gamma)$ to define the payload of the tuple t_o output from window instance Γ.

Function $f_{rm}(\Gamma, t)$ (opt.) to remove from window instance Γ the contribution of the input tuple t.

Note f_{rm} is optional since an A can also maintain the dedicated window instances to which a tuple contributes in parallel and discard a certain window instance once an output is produced from it [49].

In state-of-the-art SPEs [23], [26], when an output tuple t_o is created from a window instance Γ starting at l, referred to as $\Gamma.l$, $t_o.\tau$ is set by Aggregate A to $\Gamma.l + W_{Size} - \delta$ (i.e., the max event time observable in Γ). The values of the event times carried by the tuples output by A are in the form $nW_{Advn} + W_{Size} - \delta$. If such outputs are produced by an A while processing tuples whose event and wallclock times advance at the same pace, then A outputs tuples every W_{Advn} time units (both in event and wallclock time). An SPE can determine whether the result for a given window instance of A can be correctly produced by

consistently maintaining A's *watermarks* [28]. A's watermark W_A^ω at wallclock time ω is the earliest event time a tuple t_i fed to A can have from ω on (i.e., $t_i.\tau \geq W_A^\omega, \forall t_i$ fed to A from ω on).

Watermarks are maintained assuming sources periodically emit them as special tuples with monotonically increasing timestamps [23], [28], to notify how event time advances each source's perspective. Upon receiving a watermark W, Aggregate A stores W's time and updates W_A^ω to the smallest value among those in the set comprised of the latest watermarks from each input stream. Upon reception of a watermark that increases W_A^ω, A outputs the results of all window instances Γs whose right boundary is smaller than or equal to W_A^ω (i.e., A invokes f_{out} on any Γ whose starting event time $\Gamma.l$ is so that $\Gamma.l + W_{Size} \leq W_A^\omega$) since no more tuples will fall in such window instances, and then forwards W_A^ω to its downstream peers.

Example. Figure 3.2a presents the setup of a sample Aggregate A that computes, on a per-vehicle basis, the average speed from vehicles' position reports (whose payloads carry the vehicle ID - V_{ID} and speed - υ). A defines a window of W_{Advn} and W_{Size} of 15 and 60 minutes, respectively. Hence, the window instances A maintains for a given key evolve as in Figure 3.2a. As shown in Figure 3.2b, the SPE assigns each of the three input tuples to its corresponding window instance based on the f_K function. In Figure 3.2b, A's Watermark $W_A^{\omega_1}$ falls approaching 08:15, and A's window instances cover the period [07:15-08:15). Through f_{add}, the state of each window instance is updated based on the position reports falling in it. When A's watermark grows beyond 08:15 at ω_2, in Figure 3.2c, the function f_{out} is invoked by the SPE to compute the payload of each output tuple. Eventually, in Figure 3.2d, the window instances are advanced and their states are updated by the f_{rm} function.

2.2.2 Reinforcement learning

RL is inspired by biological learning, with an Agent that interacts with an environment through trial and error to maximize rewards from its actions and determine the optimal policy [31]. A RL model is represented as a quadruple $\langle \mathbb{S}; \mathbb{A}; P; R \rangle$, where $\mathbb{S}$ and $\mathbb{A}$ are the set of states and actions, respectively, P is the state transition model, and R is the reward function. The Agent-environment interaction is modeled after a Markov-Decision Process: at step i, the Agent in state $s_i \in \mathbb{S}$ chooses the action $a_i \in \mathbb{A}$ according to its policy, receives a reward $r_i = R(s_i, a_i)$, and enters the next state s_{i+1} under the $P(s'|s_i, a_i)$.

The goal of the Agent in RL is to find, through continuous experimentation and learning, a policy that will maximize the long-term cumulative reward $\sum_{k=0}^{\infty} \gamma^k r_{i+k+1}$, where $\gamma \in [0, 1]$ is the discount factor. The state-action function, $Q^\pi(s, a)$, denotes the expectation of the return received by an Agent that starts from the state $s_i = s$ with the action $a_i = a$, and continues to follow the policy π for decision-making. $Q^\pi(s, a)$ can be expressed using the Bellman expectation equation in terms of the expected values of future states: $Q^\pi(s, a) = \mathbb{E}_\pi [r_{i+1} + \gamma Q^\pi(s_{i+1}, a_{i+1}) | s = s_i, a = a_i]$.

One of the common methods for estimating the state-action function, which we use, is Temporal Difference (TD) [36]. It estimates the state-action function through the TD information obtained from two steps before and after the Agent-environment interaction. Q-learning [37], an RL algorithm on top of TD learning, fits the optimal state-action function $Q^*(s, a)$ using the Bellman optimality equation:

$$Q(s_i, a_i) \leftarrow Q(s_i, a_i) + \alpha \left[r_{i+1} + \gamma \max_a Q(s_{i+1}, a) - Q(s_i, a_i) \right].$$

We rely on a common extension of Q-learning, Deep Q-Network (DQN), which uses neural networks to approximate $Q^\pi(s, a)$ [38].

2.2.3 Aggregation/Compression Algorithms

The aggregation and the compression algorithms we rely on were originally presented in [43]. The compression algorithm is designed to compress specific window instances maintained by an Aggregate A depending on the event time passed since their last update. More concretely, the algorithm can be tuned through a single parameter D which specifies the minimum distance (inclusive) in event time between the timestamp of the latest tuple that contributed to a window instance Γ and that of the timestamp of the latest tuple processed by A, after which Γ should be compressed. The D value can range in $[0, +\infty]$. If set to 0, each window instance is immediately compressed after being updated upon the reception of tuple t (the event time distance between a window instance's latest contributing tuple and $t.\tau$ is equal to 0). That is, if $D = 0$, all window instances maintained by A are compressed. If $D = +\infty$, no window instance maintained by A is compressed. Note that, in general, an A receiving watermarks continuously and at the same pace with which regular tuples are received, maintains a state that does not extend back in time more than W_{Size} event time units. Hence, setting $D \geq W_{Size}$ usually suffices to have no window instances compressed by A.

2.3 Problem Formalization

As discussed in [43], the compression achieved for an Aggregate A by a given D (see Section 2.2) depends on several factors. On the one hand, it depends on A's parameters (e.g., W_{Size}, which affects the number of tuples that can fall in a window instance and that can be thus compressed, or f_K, which affects the total number of distinct window instances maintained by A). On the other hand, it depends on the data fed to A (e.g., the varying number of distinct keys observed over time). While A's parameters are usually fixed at deployment time, live changes in A's input data imply changing compression levels for the same D over time.

Based on these observations, our goal is to automate, through RL (see Section 2.2.2), the live-tuning of the memory compression level of an Aggregate A. More concretely, the goal is to define and train an Agent that can be

activated at any point in time during the execution of an A (i.e., offering an on-demand service) to continuously adjust the D value that optimizes the A's non-compressed/compressed window instances ratio (n/c ratio) while meeting a given latency threshold ℓ over a given number of steps, where:

- The *n/c ratio*, ranging in $[0, 1]$, represents the portion of non-compressed window instances over the overall window instances A maintains (i.e., it is 0 when all window instances are compressed, 1 when none is compressed).
- The *latency* (s) is measured as the average of the wallclock time interleaving the production of an output tuple and the ingestion of the last input tuple that contributed to the window instance from which such output is produced, for all the output tuples received during each monitoring period.

Since streaming applications handle continuously evolving data, we pursue *lifelong* learning, where an Agent starts without prior experience and continuously learns by interacting with an environment that evolves at the same pace as the SPE and A. We thus do not differentiate between offline and online learning. As such, besides the challenge of defining the appropriate states, actions, and rewards of the Agent (see Section 2.2.2), we also aim at creating a learning environment that efficiently supports the exploration of training episodes (discussed in Section 2.6), an environment that, to our knowledge, was not proposed before in the literature.

Assumptions. Besides the n/c ratio and latency, we consider for an Aggregate A the following metrics:

- the *input rate*, the tuples/second (abbreviated to t/s) fed to A by its upstream source(s)/operator(s),
- the *throughput*, the t/s A can process,
- the *output rate* , the t/s A outputs, and
- the *CPU consumption* of the thread running A (see Section 2.2.1).

All metrics are measured periodically in wallclock time during A's execution. As in state-of-the-art SPEs [23], such measuring is carried out by dedicated threads potentially deployed within the same CPUs used to process A's tuples and run other operators belonging to A's query. Hence, the execution of threads in charge of such measuring is affected by the workload of said CPUs. For ease of exposition, and without lack of generality, we assume this measuring period is 1 second. A metric might not be available for a given monitoring period (e.g., no latency measurement is available for a given second if no output tuple has been produced in said second). In this case, the metric carries the `NULL` value for that second.

To ensure our solution is compatible with state-of-the-art SPEs [23], [24], [25], we assume an Aggregate A runs in a shared-nothing setup. Hence, each *transition* from one D value to another in A should not be concurrently handled by threads other than that running A. Instead, changes should be notified

to the latter through tuple-based notifications (i.e., through dedicated tuples, similar to how special tuples are used to notify the progress of event time with watermarks).

For ease of exposition, we assume watermarks (see Section 2.2.1) are continuously received and that the wallclock time interleaving the last tuple contributing to a window instance Γ and the watermark triggering the output of a tuple from Γ is negligible (i.e., the state maintained by an Aggregate A at any point during its execution covers the last W_{Size} event time units and, if $D = W_{Size}$, then no Γ is compressed), and also assume there is only one A instance whose compression is being controlled by the Agent in a given streaming query.

2.4 Wallclock/event time and metrics alignment in state measurements

In a system where an Agent interacts with an Aggregate A, the two components operate in a cyclical dependency: the Agent computes its next action only after receiving the state measurement and reward, while A waits for the Agent's action before sharing a new state and reward. Note, though, a fundamental difference: the Agent's actions progress in discrete steps tied to state-reward updates while A's environment evolves continuously in both wallclock and event time, driven not only by the Agent's actions but also by the data fed to A.

A trade-off arises when reporting states after an action. If state metrics are collected at arbitrary points in wallclock or event time, the timing of the report affects how completely it reflects the effects of the action. Reporting too soon might result in a state that predominantly reflects conditions before the action, potentially distorting the reward and impeding effective learning. This highlights the need for careful alignment between the timing of state reports and the dynamics of an Aggregate's environment, critical in scenarios like ours, where metrics such as latency indicate the quality of the Agent's actions (see Section 2.3), as discussed next and evaluated in Section 2.7.

When comparing the state of an Aggregate A before an action to the state sensed after such an action, we may observe that in the latter: (1) wallclock time has progressed, (2) event time has advanced, or (3) a new latency measurement is available. Here, (2) implies (1) because event time progresses as A processes input tuples, which occurs with advancing wallclock time. Similarly, (3) implies (2) since new latency values appear only when event time advances enough for A to generate new output tuples. Based on these dependencies, four state reporting policies can be established:

- *Wallclock time, Event time, and Latency OBlivious (WEL-OB)*: a state is immediately sensed and returned after each action, independently of whether wallclock/event time have advanced or new latency values are available.
- *Event time and Latency OBlivious (EL-OB)*: a state measurement is returned only if its metrics are measured at a wallclock time greater than that at

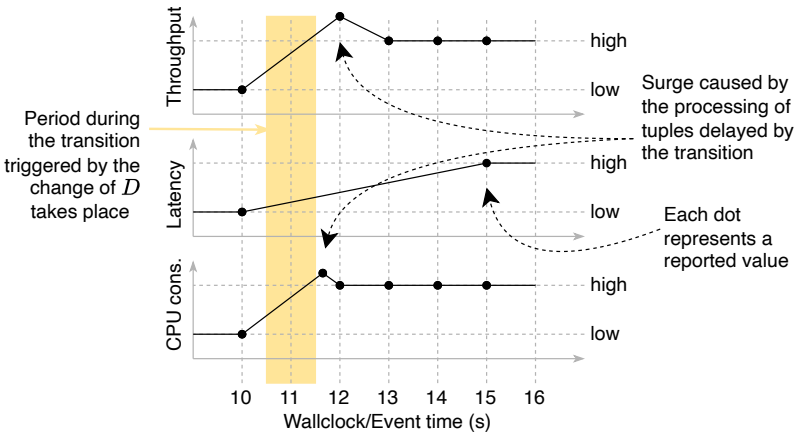


Figure 2.2: Throughput, latency, and CPU consumption metrics reported by a sample Aggregate A during a period in which a change in D triggers a transition (from seconds 10.5 to 11.5).

which the action has been received by A.

- *Latency OBlivious (L-OB)*: a state measurement is returned only if its metrics are measured at a wallclock and an event time greater than those at which the action has been received by A.

- *Wallclock time, Event time, and Latency AWare (WEL-AW)*: a state measurement is returned only if its metrics are measured at a wallclock time and an event time greater than those at which the action has been received by A, and latency values measured after the action and that are not `NULL` (see Section 2.3) are part of such a state.

Note that, since an update in the D value occurs through tuple-based notifications (see Section 2.3) whose processing is interleaved with that of ongoing input/output tuples, policies L-OB and WEL-AW imply that at least some of the metrics of a state are recorded after a transition is completed because event time can only increase once A resumes the processing its input tuples after updating D. Also, note an action decreasing D may temporarily increase the computation workload of A before completing the processing of the first tuple t after the D value change since, based on the algorithm from Section 2.2.3, a high number of window instances might need to be compressed once t is processed.

Example. To exemplify the effects of the different policies, let us consider, as in Figure 2.2, an Aggregate A processing live data (i.e., an A for which wallclock and event time evolve at a similar pace) and sustaining a high throughput with low latency and low CPU consumption. The A defines a W_{Advn} of 5 sec, and, thus, produces updated latency results every 5 seconds (see Section

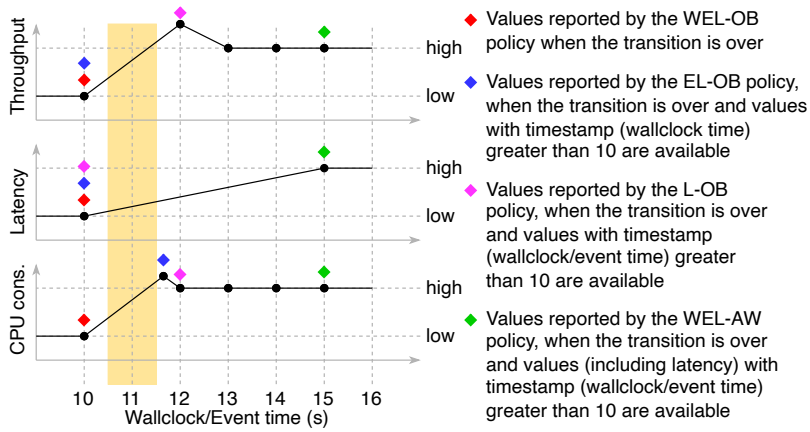


Figure 2.3: Throughput, latency, and CPU consumption values reported in the state measured after a transition triggered by a change in D depending on the chosen policy.

2.2.1). Together with latency, the state measured and returned to the Agent also contains the throughput and CPU consumption metrics.

At time 10.5, Aggregate A enters a transition that lasts 1 second (e.g., to compress window instances based on the new D value). After the processing load surge caused by the processing of tuples delayed by the transition, A can again sustain a high throughput but with both higher latency and CPU consumption. Note that, the delay in tuple processing caused by the transition results in a CPU consumption surge and in a throughput measurement that, accounting for the tuples to be processed at seconds 11 and 12, is higher than the others. Acknowledging that whether the increased latency/CPU consumption should be positively or negatively rewarded is problem-specific, the timeliness with which a state is reported after the transition can contain values steering the reward in different directions. To illustrate this, we show in Figure 2.3 how the values reported for the throughput, latency, and CPU consumption metrics depend on the chosen policy[1]. In the figure, the markers of a given color indicate which values (for each metric) are reported by the corresponding policy. For WEL-OB, the state contains the values available as soon as the transition is over. In this example, these are all values sensed before the action took place. Such a situation might occur when the transition results in a surge in CPU consumption that slows down the SPE threads in charge of measuring and reporting metrics. For EL-OB, the reported CPU consumption is the one observed and reported after the transition. Since EL-OB accounts only for the growth of wallclock time in the reported values, the reported throughput and latency values are the same as reported by the WEL-OB policy. This is not the case for L-OB, since the new values must refer to newer event times.

[1]This dependency can exist for other metrics too. Throughput, latency, and CPU consumption are discussed here in relation to the examples from Figure 2.2 and Figure 2.3.

Table 2.1: Values of the metrics contained in the next reported state based on the chosen state reporting policy.

Policy	Throughput	Latency	CPU consumption
WEL-OB	low	low	low
EL-OB	low	low	very high
L-OB	low	very high	high
WEL-AW	high	high	high

With L-OB, both throughput and CPU consumption reflect the new values associated with the chosen D value. Finally, when adopting the WEL-AW policy, values are only returned once a latency measurement with a greater event time than the one observed before the transition is measured, at second 15. With respect to timeliness in state measurements, this example shows how the reporting of a new state can be instantaneous (WEL-OB policy) or completed more than 4 seconds after a transition has been triggered (WEL-AW policy). Table 2.1 summarizes the values reported by each policy in the given example.

Based on the values contained in the state shared with the Agent after its action, the metrics used by the reward function to compute the next reward could lead to different rewards. As mentioned, how much different rewards affect the Agent's learning depends on the problem at hand. As we empirically show in Section 2.7, they can have negligible effects, they can lead to an Agent taking more conservative actions, or an Agent whose actions lead to aggressive compression that does not meet the given latency threshold.

From an implementation perspective (see Section 2.6), the different policies can be enforced through barriers not reporting a state depending on the wallclock/event time carried measurements in it.

2.5 Reinforcement Learning Model

We here present our RL model, aimed at choosing the D value and optimizing an Aggregate A's n/c ratio while satisfying a given latency threshold ℓ over a given number of steps. We proceed by covering the action space, the state space, the reward function, and the model hyperparameters and dynamics.

Action Space. Observing $D \in [0, +\infty]$ (see Section 2.2.3) and that the n/c ratio assumes continuous values within $[0, 1]$, our first step is to discretize the action space into a manageable set of actions that the Agent can learn. We do this by allowing D to assume values in a set of fractions of the Aggregate A's W_{Size}, e.g., $0, 0.1, \ldots, 1$. As mentioned in Section 2.2.3, if $D = 0\%$ of A's W_{Size}, all window instances maintained by A are compressed, while if $D = 100\%$ of A's W_{Size}, no window instance is compressed. To avoid abrupt perturbations in A's computation load, e.g., by changing D from 100% of W_{Size} to 0% in a single action, and to avoid having a too-large action space to be learned by the Agent, we define 3 actions: *Compress*, *Stay*, and *Decompress*, and a value d (fraction of A's W_{Size}) specifying by how much D should be

changed for actions *Compress* and *Decompress* (e.g., if $d = 0.1$ the Agent can, at each action, decrease D by 10% of A's W_{Size}, keep the same D, or increase it by 10% of A's W_{Size}).

Note the learning process has a stochastic nature because a fixed D/a D changing in one direction do not imply a fixed n/c ratio or an n/c ratio that changes in the same direction as D. While keeping a steady D value, for instance, a change in the distribution of the data fed to the Aggregate A might result in a higher or lower compression depending on the number of keys that were not updated in the last D time units (see Section 2.2.3). Such a distribution change could also lead to a lower compression when the D value is increased.

State Space. In RL, the state space needs to encapsulate relevant operating environment features for the Agent to successfully learn which actions are best depending on each state instance. In the study in focus, such features should indicate how the n/c ratio and latency change based on the Agent actions, but also be informative about the Aggregate A's status, workload, and possible saturation (due to a too high workload caused by a high input rate or a too aggressive compression). To account for all these aspects, we define a state that comprises all the metrics from Section 2.3, noting that:

- the input rate carries information about A's workload (a higher input rate implies a higher workload),
- the throughput carries information about A's workload too, but also hints at a possible saturation when it does not match the input rate (a saturated A will process tuples at a lower speed with which such tuples are fed to it),
- the output rate carries information about A's workload and the number of window instances maintained by A, since one output tuple is produced every W_{Advn} time units (when A is not saturated) for each window instance maintained by A,
- the latency carries information about A's workload and about whether the Agent meets the given threshold ℓ,
- the n/c ratio carries information about the compression achieved by the D values chosen by the Agent, and
- the CPU consumption carries information about A's workload, also hinting at a possible (imminent) overload of A.

To enable the Agent to observe how these metrics evolve based on its actions, the state covers these metrics based on the past N measurements in wallclock time. For each metric, the collected data is used to calculate a linear regression, yielding the slope – indicating the rate of change over time – and intercept – the initial value within the observation period and thus the metric's baseline state – as representative values. This aggregation structure supplies the Agent with a concise, trend-based view of each metric, facilitating more informed decision-making based on recent system behavior.

Reward and early termination criteria. The reward function and early termination criteria are designed to guide the Agent toward three main goals: (1) to avoid exceeding a specified latency threshold ℓ, (2) to maximize the number of steps completed in each episode (i.e., the duration of on-demand compression actions), and (3) to maximize the compression of the Aggregate A's window instances.

Since our setup involves exploratory learning, the aim is not to hard-code rewards to enforce highly specific behaviors, which could otherwise be directly implemented. Instead, the objective is for the Agent to explore combinations of the available actions – *Compress*, *Stay*, and *Decompress* – to determine the most effective policy. To encourage this exploration, rewards are structured as follows:

For latency control (1), we apply an early termination criterion where episodes are ended if a maximum latency threshold ℓ_{max} is exceeded. Note that ℓ_{max} may be set above the target latency ℓ, allowing the Agent some flexibility for short latency bursts while it learns an effective policy. To encourage the completion of steps (2), we give the Agent a fixed positive reward R_F after every η steps. To maximize compression (3), the Agent can get a positive reward at each step i, if the latest valid n/c ratio at step i is less than that of the previous step $i-1$ while the latency is below ℓ, according to the formula $\lfloor\sqrt{100\left[(n/c)_i - (n/c)_{i-1}\right]}\rceil$, within the range of $[1, 10]$.

Furthermore, while we do not wish to enforce specific action sequences, we want to prevent the Agent from learning clearly suboptimal policies. A policy overly favoring a single action is generally inefficient: consistently choosing *Compress* either implies the compression overhead is negligible for a given latency threshold ℓ and thus that no learning is required or leads to a too-aggressive Agent; while always choosing *Stay* or *Decompress* leads to minimal or no compression. To address this, we track the entropy of action probabilities, computed by the Boltzmann decay scheduler [32], as $H = -\sum_{a=1}^{n} p_a \log p_a$ where n is the number of actions and p_a is the corresponding probability of the action a, and monitor its moving average for every H_η steps, denoted as $\bar{H}$. When $\bar{H}$ falls below a threshold T_H, indicating an overly repetitive action pattern, we impose a fixed negative reward R_H at each step and reduce the R_F reward given after η steps by a factor R_η, discouraging stagnant policies and encouraging balanced action selection.

Model hyperparameters and dynamics. Besides the problem-specific actions, states, and rewards, the model hyperparameters and dynamics are chosen according to other state-of-the-art RL solutions for stream processing [50], [51]. More concretely, the network structure for the Agent follows the DQN framework (see Section 2.2.2), using fully connected layers with ReLU activations for nonlinear transformations. Training is conducted in batches to balance computational and training efficacy. Additionally, the model employs an optimizer alongside an appropriate learning rate to mitigate issues such as divergence or excessively slow convergence. An experience buffer is employed to ensure sufficient diversity in training samples while minimizing unnecessary computational overhead [52]. The target network in the DQN framework is

also updated at regular intervals to strike a balance between frequency and stability [53]. Finally, a relatively large discount factor emphasizes long-term rewards, essential for effective decision-making in dynamic stream processing environments.

Since we aim to support lifelong learning (see Section 2.3), we rely on a Boltzmann decay scheduler [32] rather than an epsilon-greedy approach for the exploration probabilities to be dynamically influenced by the changing rewards, allowing the Agent to better adapt to evolving environments and maintain a balance between exploration and exploitation over extended periods. Also, we initialize the weights and bias of network layers to enhance training outcomes by balanced weight – to ensure they are equally likely to start with positive or negative values, thereby preventing biased gradient update during early training, and positive bias – to guarantee the neurons remain active when using ReLU activation and avoid dead neurons [54]. Further configuration details are provided in Section 2.7.

2.6 Learning Framework

While related works have explored the use of RL in stream processing [55], [56], [57], [58], this study is, to the best of our knowledge, the first to focus specifically on tuning an Aggregate As' memory compression. In the absence of an existing framework, we aimed to design and implement a new, general-purpose framework. There are two key challenges such a framework must account for to be general:

- The interleaving of the processing of tuples within the A and other sources/operators/sinks within the SPE and the query to which the A belongs takes place in a shared-nothing environment (see Section 2.2.1). Hence, concurrent state changes should not alter or break the internal state of an A and must be handled carefully by the framework itself. More concretely, the thread in charge of A's analysis (see Section 2.2.1) and A's upstream/downstream peers' threads should not operate on a state that is concurrently modified by the framework but rather be notified of changes they later apply themselves (in a similar spirit in which broadcasted streams are used in Flink to notify operators and steer their analysis at execution time [59]).

- The time to prepare each episode and to transition through episodes should be minimal, to minimize the overhead introduced in the learning process by the framework itself. This is particularly so when accounting for the stateful nature of an A, whose setup must initialize A's state, which is proportional to its underlying W_{Advn} and W_{Size}. Note the initialization of an A's state is achieved in the same way in which an A is then run, i.e., by feeding data to it. From the framework perspective, it is thus important to distinguish tuples contributing to the initialization of an A's state from those sent during the normal execution of such an A.

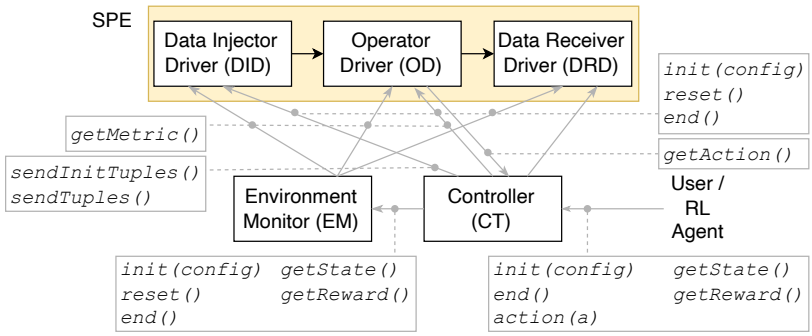


Figure 2.4: Architecture of the proposed framework.

Figures 2.4 and 2.5 overview the main components of the framework and the interaction between the stream components related to an Aggregate A and the Agent, showing how the framework encapsulates the common steps of RL processes (see Section 2.2.2). The framework provides, for a given SPE, three encapsulating units called Drivers:

- the Data Injector Driver (DID), which encapsulates the sources/operators of the query to which A belongs that precede (in topological order) and feed tuples to such an A,
- the Operator Driver (OD), which encapsulates the A whose compression is to be controlled by the Agent, and
- the Data Receiver Driver (DRD), which encapsulates all the operators/sinks that follow A (in topological order).

Finally, the framework defines an Environment Monitor (EM), computing/sharing states/rewards with the Agent, and a Controller (CT), exchanging information between the SPE and the Agent/user.

A user interested in an Agent acting on a given Aggregate A can then encapsulate such an A in the OD and A's upstream/downstream peers in the DID and the DRD, respectively. Once deployed, the framework can monitor and act on the data exchanged between them whenever an Agent is activated. More concretely, once a query is deployed in the given SPE with its sources, operators (including A), and sinks encapsulated in the DID, OD, and DRD units, the `init(config)` and `end()` methods can be invoked by the user to activate/deactivate a given Agent. Subsequent invocations of the `init(config)`/`end()` method do not require the re-deployment of the SPE nor the framework components.

For the `init(config)` method, the `config` object contains the information on A's configuration (e.g., its W_{Advn} and W_{Size} parameters, see Section 2.2.1), the parameters of the Agent (e.g., the discount factor γ, see Section 2.2.2), and the parameters concerning the framework setup (e.g., the state reporting policy to be used, see Section 2.4). Internally, the CT invokes `init(config)` on all other modules (DID, OD, DRD, and EM).

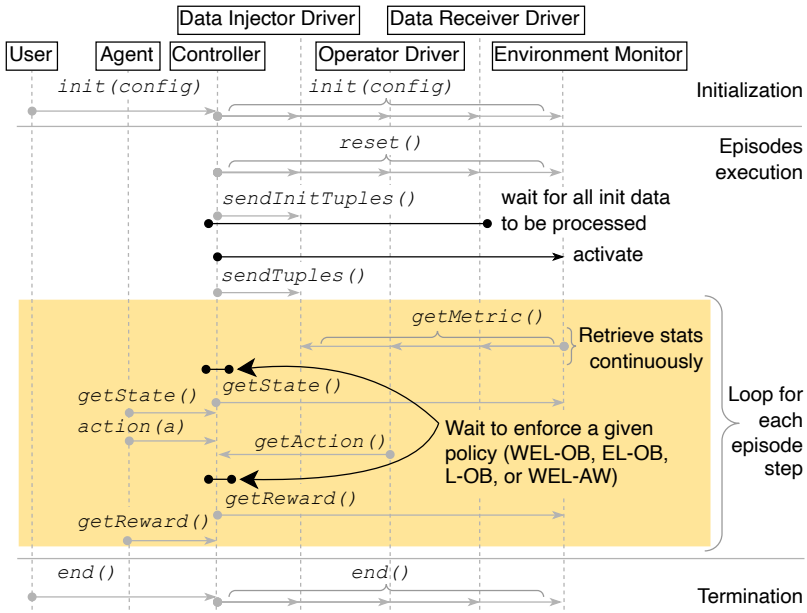


Figure 2.5: Main steps performed by the framework during the training/execution of an Agent.

Upon the beginning of an episode, the CT notifies all components by invoking `reset()`. Subsequently, it invokes `sendInitTuples()` on the DID to notify the latter the initial data required to initialize A's state can now be forwarded. As aforementioned, the framework needs to distinguish which tuples are fed to A to initialize A's state from those sent during A's normal execution. To do so, once `sendInitTuples()` is invoked, the CT waits until the completion of `sendInitTuples()` (i.e., until all the initialization data has been sent from the DID to the OD). To know that the OD has processed such data, the framework inspects the stream between the DID and the OD for pending tuples to be processed. If no tuple is found in such a stream, the framework can then assess whether all the output tuples that should be forwarded from A to its downstream peers at the DRD have been processed by the latter by inspecting the tuples/watermarks flowing from the DID to the OD. More concretely, based on the timestamp of the latest tuple sent from the DID to the OD, and based on A's W_{Advn}/W_{Size} (shared through the `config` object), the framework can compute the highest timestamp that can be observed for the output tuples produced by the OD during the initialization phase and wait until the first watermark with an equal or greater timestamp is observed in the stream between the OD and the DRD.

Note that, while the method `sendInitTuples()` is invoked, no states/actions/rewards are being shared between the Agent and the A. Hence, during this phase, the DID can reduce the state initialization time by feeding tuples to A at a higher rate than that used during A normal execution (e.g., by sending

tuples as fast as A can process them). This mechanism can be leveraged when training an Agent through previously sensed data to shorten the initialization phase of learning jobs, as we show in Section 2.7. Once an episode setup has been finalized, the episode starts with the CT invoking `sendTuples()` on the DID. During the episode, the CT will forward the actions from the Agent to the A and the state/rewards from the EM to the Agent.

The actions are passed from the Agent to the CT through the `action(a)` method. The A deployed within the OD can pull any incoming action through the `getAction()` method. The states/rewards are internally prepared by the EM through the continuous retrieval of metrics from other components through the `getMetric()` method. Once the next action has been retrieved from the OD, the CT will poll the EM to get the new state and reward. Internal barriers will ensure the state is pulled according to the predefined state reporting policy. The exchange of actions, states, and rewards continues until all episodes are completed or until the user terminates the Agent execution through the invocation of the `end()` method, which is then forwarded by the CT to all the other components.

2.7 Evaluation

We assess the different state reporting policies (see Section 2.4), the proposed RL model (see Section 2.5), and our RL environment (see Section 2.6) using the de-facto standard benchmark for streaming applications [39]. To complement such a usecase with a study investigating how our solution behaves under more extreme variations in data rates and distributions, we also define a complementary synthetic usecase. Besides studying how the Agent supports the memory compression of the Aggregate under study, we also discuss the scalability of the proposed solution by assessing (1) the time required to prepare the state of the Aggregate at the beginning of an episode and (2) similarly to [50], the cost incurred by the Agent itself, in this case in terms of CPU and Memory consumption overhead. We first cover the experimental setup. All our code is available at `https://github.com/appalachia-xx/compAgg`.

Hardware. As mentioned in Section 2.3, we assume the Agent is acting on a single Aggregate A instance. Hence, all experiments are conducted within a single server. In detail, the machine we use is an Intel Xeon E5-2637 v4@3.50GHz, with 4 cores, 8 threads, and 64 GB RAM. As discussed in [43], this server is a device suitable for deployment at the edge of the cloud-edge continuum, specifically linked to the Linear Road usecase discussed next.

Software. Our framework uses both Java (OpenJDK version 17.0.7) and Python 3.7.6. We rely on Liebre [24], an SPE previously used for assessing memory compression techniques for stream Aggregates [43] and operator scheduling [60]. The compression library used within Liebre to compress/decompress the window instances of the Aggregates under study is *snappy-java*, a port of the snappy compressor/decompressor developed by Google [61]. The Agent runs in the OpenAI Gym [62] environment in Python.

Data and usecases. The first usecase, based on the Linear Road bench-

mark [39], extends the usecase from [43]. Linear Road simulates a real-time traffic monitoring system. Data from each vehicle comes periodically, every 30 seconds. Each position report carries a timestamp and the vehicle's ID, position, and speed in its payload. The vehicle's ID is returned by the f_K function (see Section 2.2.1) of the Aggregate under study to maintain per-vehicle window instances. Each vehicle produces reports for a few to tens of minutes. Overall, the data covers 3 hours (in event time) and consists of ∼44 million tuples. The Aggregate under study computes the number of non-consecutive stops of each vehicle (i.e., the sequences of position reports with 0 speed in between reports with speed greater than 0, also considering sequences occurring at the beginning or the end of a window instance) over a W_{Advn} and W_{Size} set to 5 and 600 seconds, respectively. For a given window instance Γ, functions f_{add}/f_{rm} maintain the tuples falling in Γ while f_{out} computes the number of consecutive stops.

To complement Linear Road with a usecase having more extreme variations in data rates and distributions, our second usecase – referred to as *Synthetic* – defines an Aggregate A that spends a fixed per-tuple processing time (performing math operations on a random value carried by each tuple) over a window with W_{Advn} and W_{Size} set to 1 and 900 seconds, respectively. The data is generated following a sawtooth wave whose peaks' values and distances are chosen randomly. Besides the random value, each tuple carries a key attribute, returned by the f_K function of the A, generated from a Gaussian distribution with changing μ/σ^2. A's computations are encapsulated in f_{add}/f_{rm}, while f_{out} outputs the last computed value.

Experimental setup. For each policy and usecase combination, and one of the baselines presented next, we run 120 episodes, without previous training (see Section 2.3). Each episode runs for a maximum of 1000 steps. Given our goal of training an Agent that, through lifelong learning, supports on-demand compression once activated at any execution point, each episode starts at a random event time, excluding the first W_{Size} event time units (so that enough previous data exists to load the state of the Aggregate before any episode starts). At the beginning of an episode, D is set to W_{Size}, implying no compression is initially applied (see Section 2.2.3). During each episode, the input rate is chosen so that both the wallclock and the event time advance at the same pace (i.e., the input data is being fed as if live). This does not hold for the data used to fill an Aggregate's state which, in our framework, can instead be fed as fast as possible to minimize the episode setup time (see Section 2.6). The d, N, ℓ, ℓ_{max}, R_F, η, H_η, T_H, R_H, and R_η parameters of the Agent (see Section 2.5) are set to 0.1, 7, $1.5s$, $2s$, 30, 10, 1000, 0.6, -1, and -5, respectively. The ℓ parameter is set to 1.5s since, in the original contribution [43], such a value represents a latency threshold that can be met for any of the D values taken into account and for low-enough input rate values ($\sim 1000\ t/s$). An episode terminates early once 1 reported latency value exceeds ℓ_{max}. The Agent relies on DQN (see Section 2.2.2 and Section 2.5) with 2 hidden layers, each with 128 units, whose weights are initialized uniformly between -0.03 to 0.03 and with bias 0.05, except for the final layer, whose bias is set to 0.03 once reaching the first training batch size. Updates are done with a batch size of 128 using the

Adam optimizer, with a learning rate of 0.01. The experience replay buffer has a capacity of 10000. The target network is updated every 1000 steps, and the discount factor γ is set to 0.99. For the exploration strategy, the Boltzmann decay scheduler, starts from 5, with a decay factor of 500, and ends at 1.

We evaluate an Agent by comparing (1) its behavior for each state reporting policy and (2) the performance of each policy with several DX baselines. Each DX baseline always sets the D value to XW_{Size}, with $X \in [1, 0.9, 0.8, \ldots, 0]$. The rationale is that DX baselines will choose, no matter the underlying Aggregate's data volumes and distribution, a safe yet suboptimal compression or a more aggressive but excessive compression. Note that $D1.0$ also represents the baseline from the original contribution with no compression [43] (see Section 2.3).

Results. Figures 2.6 and 2.8 show how an Agent performs, based on its state reporting policy, and also contrast the performance of an Agent and state reporting policy combination against other baselines for the Linear Road and Synthetic usecases, respectively.

The upper plot shows the input rate against the event time, measured in thousands of t/s. Since wallclock and event times align, this plot reflects the real-time input rates during the experiments. For each episode within an experiment, we collect the n/c ratio, latency (s), and CPU consumption every second. We then compute the respective mean values for the episode. The 3 underlying plots, for each state reporting policy, show then the moving average of such mean n/c ratio, latency (s), and CPU consumption, respectively, of the individual episodes based on their start event time. The green line of the n/c ratio for the L-OB policy in Figure 2.6, for instance, indicates that episodes starting before the event time $\sim$ 2500s achieve a mean n/c ratio below 0.5, while episodes starting after $\sim$ 3500s achieve a mean n/c ratio of approximately 0.9. A dashed red line across the latency plot denotes the ℓ threshold. In the bottom part, violin plots summarize the metrics for the different baselines and the Agent/state reporting policy combinations, based on the mean values observed across all episodes. In between green vertical lines, *safe* DX baselines have values that are always below the ℓ threshold.

Linear Road (Figure 2.6). When comparing the behavior of the different state reporting policies, we observe that: (1) for WEL-OB, EL-OB, and L-OB, the n/c ratio is lower for lower input rates, while WEL-AW achieve an almost constant n/c ratio. This is an indication that waiting to observe the effects of the most recent action on the latency metric results in a more conservative Agent; (2) all policies except WEL-OB maintain similar, acceptable levels for the latency – with WEL-OB also resulting in a higher CPU consumption; and (3) shorter episodes are observed for WEL-OB which proves overly aggressive, while the conservative Agent behind WEL-AW can sustain each compression episode for longer times. For EL-OB and L-OB policies, lower input rates are associated with longer episode durations. Note that, for these experiments, the times required for a new state to observe a new wallclock time (EL-OB), event time (L-OB), or latency value (WEL-AW), are of approximately 1, 1.8, and 5 seconds, respectively.

While observing that all policies compress the Aggregate's window instances

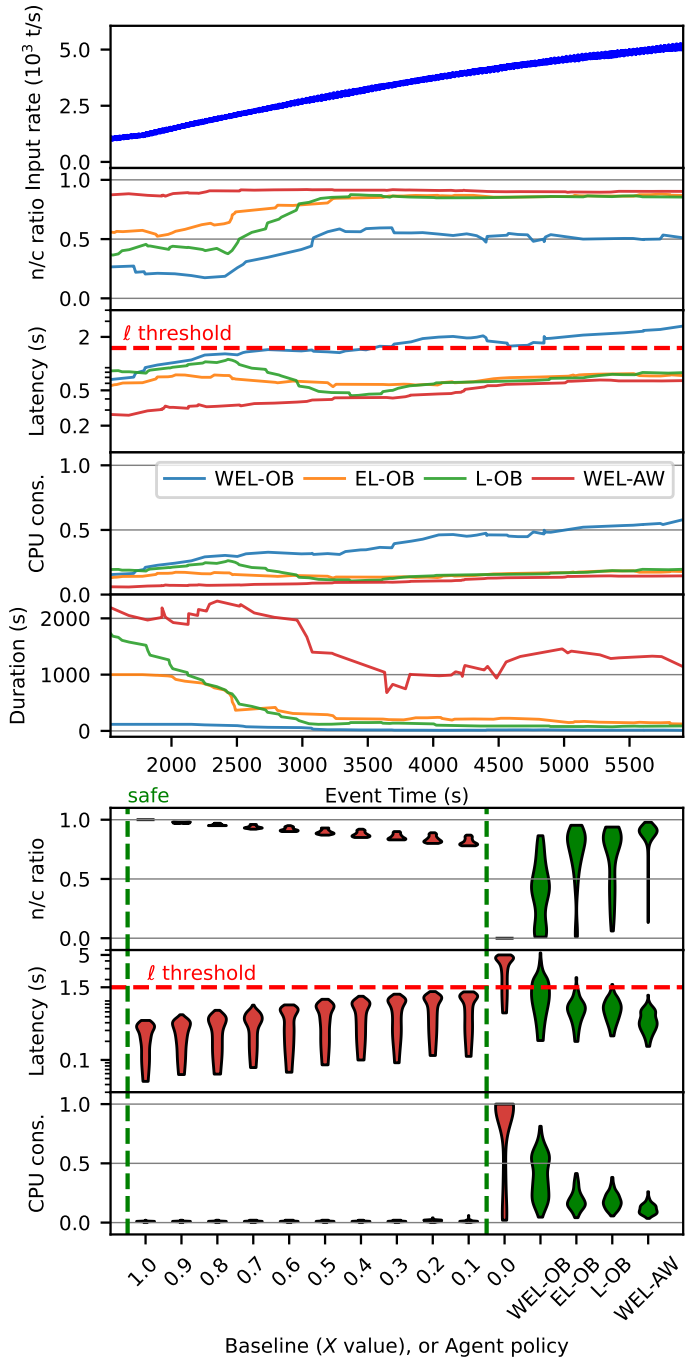


Figure 2.6: Agents performance vs. input rate, and Agents vs. baselines performance for n/c ratio, latency, and CPU consumption and different state reporting policies – Linear Road.

to different levels and with different latency and CPU consumptions, we can assess whether the Agent is effectively learning by comparing its results with those achievable by the constant compression of DX baselines. As shown in the bottom part of Figure 2.6, all DXs except for $D0.0$ are safe. Lower D values correlate with better n/c ratios, typically not falling below ∼0.80. For $D0.0$, though, compression reaches its maximum by significantly surpassing the ℓ threshold and resulting in a high CPU consumption. The abrupt change in the n/c ratio is due to the underlying data, in which each vehicle reports its position every 30 seconds. Down to $D0.1$ (i.e., 0.1 of W_{Size}, or 60 seconds) means compressing only the window instances of vehicles that have stopped reporting positions, which are less than those actively reporting. In contrast, $D0.0$ means compressing all window instances, including those of actively reporting vehicles.

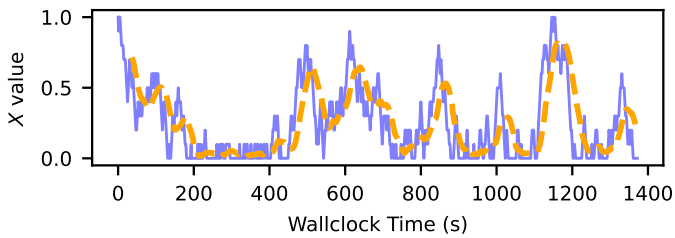


Figure 2.7: X values resulting from the actions taken by the Agent (Linear Road, L-OB). The moving average – dashed line – is added to better appreciate the actions' fluctuations.

If, on the one hand, these baselines indicate that most of the actions are safe, they, on the other hand, imply the Agent should efficiently explore its state space to learn to get closer to lower D values to minimize the n/c ratio. To provide an example of the Agent's ability to do so, we show in Figure 2.7 the X values chosen by the agent for one of the episodes and the L-OB policy (the dashed orange line shows a moving average and is added only to better appreciate the Agent decisions).

All Agent/state reporting policies, except for the too-aggressive WEL-OB, achieve n/c ratio values lower than those of the safe baselines while having almost all latencies below ℓ and CPU consumptions lower than the $D0.0$ baseline. Note though that the WEL-OB policy results in fewer latency violations than the $D0.0$ baseline, indicating it can be still beneficial if e.g., ℓ represents a soft latency threshold.

Synthetic (Figure 2.8). This usecase defines a W_{Advn} of 1 second. Hence, the time needed for a state to include a new latency value is smaller than that needed by the Linear Road usecase. In fact, for Synthetic, the times measured for a new state to observe a new wallclock time (EL-OB), a new event time (L-OB), or a new latency (WEL-AW), are of approximately 1, 1.6, and 1.7 seconds, respectively. The effects of such closer values can be observed in the resulting behavior of the Agent/state reporting policy combinations. As shown in the upper part of Figure 2.8, all policies have homogeneous behaviors.

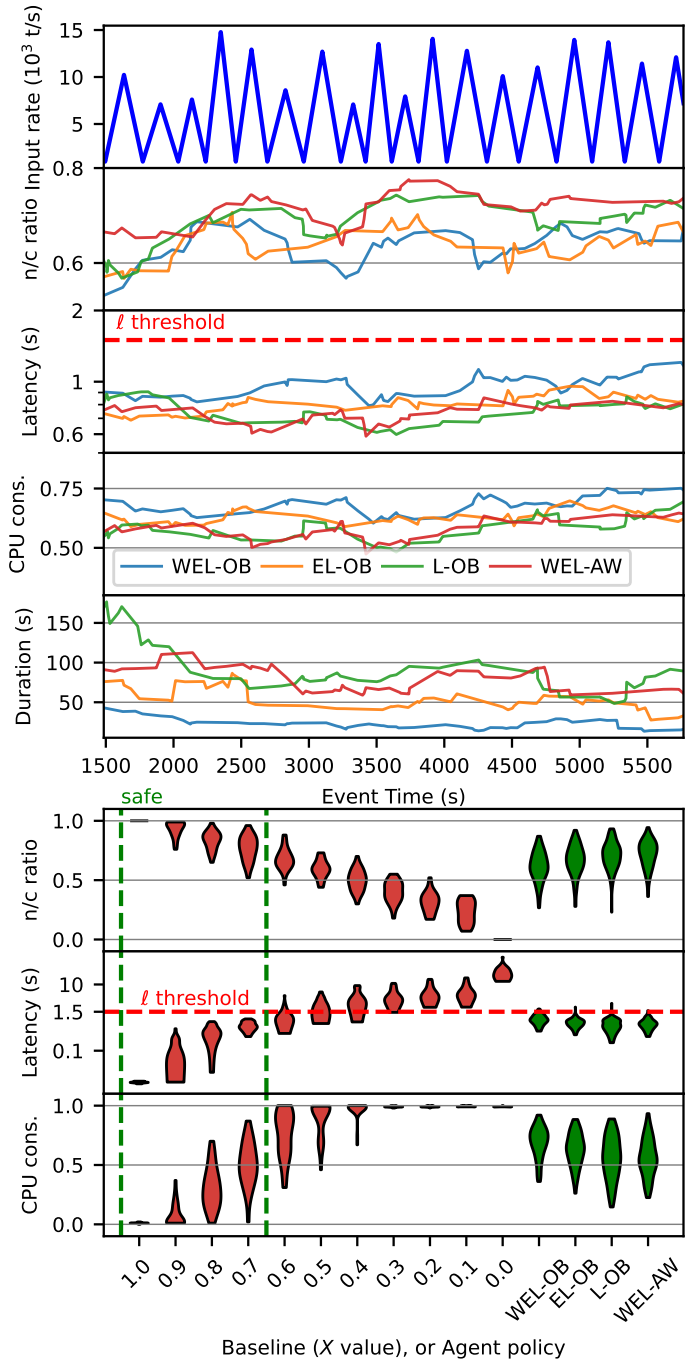


Figure 2.8: Agents performance vs. input rate, and Agents vs. baselines performance for n/c ratio, latency, and CPU consumption and different state reporting policies – Synthetic.

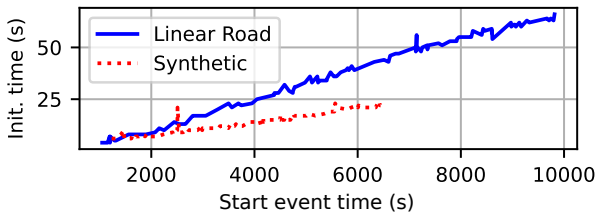


Figure 2.9: State initialization times (Linear Road/Synthetic).

Moving now to the comparison with the baselines, we observe that when D values are fixed, the n/c ratio decreases linearly with lower D values, but these are coupled with higher latency and CPU consumption, and baselines from $D0.6$ downward are not safe. In this case, the baselines indicate that a wrong sequence of actions has a higher probability of breaching the ℓ threshold than for Linear Road, making learning more challenging. Nonetheless, we see the Agent/state reporting policy combinations demonstrate an ability to fine-tune their strategy to maximize compression without compromising on latency or too-high CPU consumption.

For both usecases, increasing the wait time between actions (ordered by policies from longest to shortest wait time: WEL-OB, EL-OB, L-OB, and WEL-AW) results in more conservative Agent behavior concerning compression. At the same time, longer wait times lead to lower latency and CPU consumption, as the Agent takes fewer, more cautious actions. In both usecases, WEL-OB results in an Agent behaving more aggressively compared to the other policies, highlighting a trade-off between action frequency and resource efficiency.

Scalability discussion. To conclude our evaluation, we now focus, for each of the two usecases, on the time required to load the Aggregates' states at the beginning of each episode, as well as the computational overhead introduced by the framework while exchanging states, actions, and rewards with the Aggregate under study.

State initialization times. As discussed in Section 2.6, our framework allows for the tuples sent to initialize the state of an Aggregate A to be sent as fast as possible. As shown in Figure 2.9, this can significantly decrease the duration of the overall learning process. More concretely, it requires no more than 1 minute (instead of 10) to load A's state for Linear Road, and no more than 25 seconds (instead of 15 minutes) to load A's state for Synthetic. Note the growing trend, especially for Linear Road, is due to the growing rate and number of keys over event time.

Computational overhead. To evaluate the computational overhead introduced by our framework, we measure the CPU usage and latency of the Aggregate across both usecases and 20 random episodes, comparing scenarios where the Aggregate operates independently to those where it is connected to an Agent. To isolate the overhead caused by the framework communication, and exchange of states, actions, and rewards, we use an Agent that consistently selects a D value as 50% and 70% of A's W_{Size} for Linear Road and Synthetic, respectively. This ensures no additional overhead from the compression or

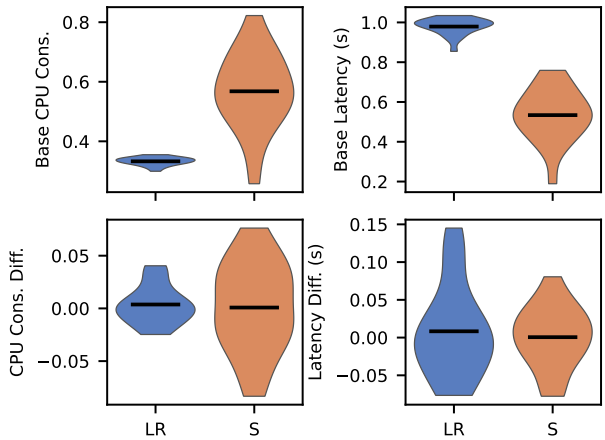


Figure 2.10: CPU/latency (top) and overheads of the RL framework measured as differences for Linear Road (LR)/Synthetic (S) Aggregates alone vs. connected to the Agent (bottom).

decompression of window instances, as these costs were analyzed earlier in the evaluation. Furthermore, since the Agent runs in a separate Python process, we measured CPU/memory of such process for a complete view of the framework's resource demands.

Results are presented in Figure 2.10. The violin plots include horizontal lines indicating average values. When the Aggregate operates without a connected Agent (top plots), it incurs average CPU consumption values of 0.33 and 0.59, and average latencies of 0.98s and 0.53s for Linear Road and Synthetic, respectively. With the Aggregate connected to an Agent (bottom plots), the observed differences in CPU consumption exhibit first-quartile, median, and third-quartile values of -0.01, 0.0, 0.01 for Linear Road, and -0.02, 0.0, 0.03 for Synthetic. In terms of latency differences, these values are -0.02s, 0s, 0.03s for Linear Road, and -0.02s, 0s, 0.03s for Synthetic. For the Python process running the Agent, we measure an average CPU usage of 0.02 and an average memory footprint of approximately 400 MB.

Overall, these results underscore the minimal overhead and resource footprint introduced by our Agent, whose implementation ensures it does not become a scaling bottleneck for the Aggregate.

2.8 Related Work

To the best of our knowledge, ours is the first work to apply RL to tackle memory compression for stream Aggregates. In [43], which introduces the compression algorithm we adopt, selecting an appropriate D value (whether statically or dynamically) is noted as a challenging task but remains unaddressed.

Note that our choice for RL is motivated by complementary work in which RL has been used in stream processing to automate different types of decision-

making processes, most notably in operator placement (also referred to as operator scheduling) [55], [57], [63], [64], elasticity [65], [66], [67], [68], [69], and resource allocation [48], [70], [71] problems.

In [57], [63], the authors model the operator placement and operator reconfiguration as an RL problem to automate decisions about where the operators of a query are deployed within the cloud-to-edge spectra. Similarly, RL is leveraged in [64] for task placement in heterogeneous stream processing environments with Apache Flink, and in [55] to enable the system to make scheduling decisions dynamically, guided by RL to jointly learn from limited runtime statistics and optimize the task scheduling process.

In [48], the proposed RL model exploits transfer learning in the context of resource allocation to apply knowledge from previous environments to new tasks, thus enabling the deep RL model to learn representations and generalize allocation policies across different stream processing scenarios by employing a graph-aware encoder-decoder framework. In contrast, the complexity of resource allocation in large-scale diverse stream processing graphs, using a coarsening strategy with the help of an RL-based approach, is tackled in [70].

In [65], RL is applied in conjunction with elasticity control policies on heterogeneous resources to handle uncertainties in system parameters. Similar studies are discussed in [66], [67], where the authors focus on adapting the application parallelism at runtime while meeting a given QoS and accounting for time-evolving workloads. In [68], the authors tailor the use of RL in connection with the Apache Spark SPE to steer auto-scaling decisions while meeting execution time constraints. Traffic-aware scheduling mechanisms such as [69], based on Q-learning, are also proposed to ensure optimal task distribution across heterogeneous cluster nodes by accounting for both the workload's characteristics and the available resources of each node.

Complementary work relies on RL to configure distributed SPEs and key-value stores to minimize failure impacts [72], to optimize checkpointing intervals in streaming applications to balance fault tolerance and resource efficiency [71], to achieve a balance between diverse objectives through dynamically reconfiguration data stream analytics tasks [57], to address the concept drift by leveraging continuous event streams to adaptively discover and update process models as the process evolves [73], and to automate decisions about whether incoming data, over time, should be processed locally, transmitted for processing at another node, or stored for later processing [74].

2.9 Conclusions and Future Work

We propose an RL-based scheme in which an Agent can be activated to act on a live Aggregate to adjust its memory compression by applying a given number of compression actions while meeting a given latency threshold. More concretely, we introduce an RL model whose actions, states, and rewards are modeled for an Agent to efficiently learn the appropriate behavior in a lifelong learning setup. To support such learning, we design and implement a framework that allows such an Agent to interact with an Aggregate running live in a real

SPE, and propose several policies to trade off fast-paced rewards that could nonetheless reflect the effects of an action only partially, with rewards that can provide more comprehensive feedback on the effects of an action but at a slower pace. We assess our solution with an extensive evaluation, showing the resulting Agent can support on-demand memory compression for stream Aggregates. To our knowledge, ours is the first solution to such a problem in the literature.

We believe this work can stimulate novel research in several directions. The learning process can be extended to account for other Quality-of-Service metrics besides latency, as well as other compression schemes, possibly accounting for different compression techniques to be adopted – possibly at the same time – depending on the Aggregate at hand and the data fed to it. Also, the proposed framework can be extended to support Agents adapting other aspects of Aggregates, such as adaptively adjusting their parallelism degree or load shedding. Furthermore, transfer learning can be leveraged to improve the Agent's knowledge with training performed on other Aggregate instances/data fed to an Aggregate, thus reducing the time required to adapt to new scenarios. It would also be valuable to implement the proposed approach within SPEs, such as Apache Flink, widely used in real-world deployments. Finally, our framework can be extended to further improve the fitness of an Agent by training it over a mixture of live and simulated episodes, further trading episodes' duration with state/reward's accuracy.

Chapter 3

Chill: Runtime Reconfiguration of Windowed Stream Aggregates with Semantic Guarantees

---

J. Liu, V. Gulisano

Proceedings of the 20th ACM International Conference on Distributed and Event-based Systems, 2026, pp. 157-168.

PAPER B

This is a formatted and adapted version of the paper published in *Proceedings of the 20th ACM International Conference on Distributed and Event-based Systems, 2026, pp. 157-168.* Any performed changes serve only to retain the consistency of this thesis and adapt to the layout.

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.

3.1 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 [10] and vehicular networks [11], [14]) and transformed into insights. Executed by Stream Processing Engines (SPEs) [23], [24], [26], 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.

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 [75], parallelization [76], wavelets [18], sketches [19], sampling [20], compression [21], or shedding [77] – 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 Section 3.2), and they steer costs either by manipulating the input tuples (e.g., discarding them through load shedding [77]) or by modifying the aggregation state (e.g., using approximate structures such as sketches [19]). 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?*

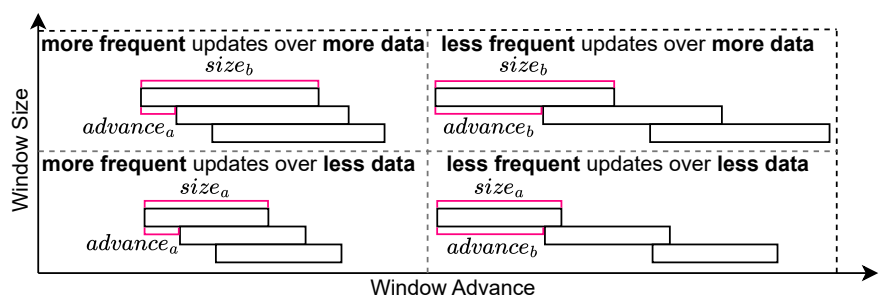
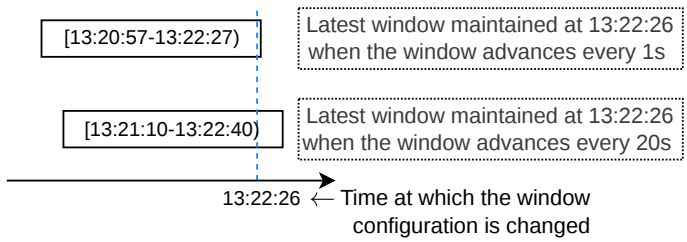


Figure 3.1: Computation cost under different advances and sizes.

Contributions. We make the following contributions with a focus on practical deployment. By building on pane-based aggregation [78] (see Section 3.2), 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 3.1.

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.



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.

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 [23] (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. Section 3.2 covers preliminaries. Section 3.3 introduces our system model and problem definition. Section 3.4 details our algorithms. Section 3.5 covers the empirical assessment of our approach. Section 3.6 reviews related work. Section 3.7 concludes and suggests future directions.

3.2 Preliminaries

A *stream* S is an unbounded sequence of *tuples* [16]. 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 δ increments (e.g., milliseconds [23]).

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 [23], [24] 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 $[m\,wa, m\,wa + ws)$, with $m \in \mathbb{N}$, $wa > 0$, and $ws > 0$, covered by A (right boundary exclusive). Each interval is a window *instance* γ; $\gamma.l$ 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 γs for tuples sharing the same key. Although optional in SPEs [23], [24], 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 [23], [24], 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-γ state based on its execution strategy [29], where single-window, multi-window, and pane-based (also referred to as slicing [30]) are among the main strategies discussed in the literature.

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

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

(iii) Pane-based (the approach we use). Panes, whose instances we denote as ps, are consecutive, non-overlapping intervals that may vary in length. Each γ 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 3.2 in connection with the example that follows.

Correctness. In state-of-the-art SPEs [23], [26], when an output tuple t_o is produced for a window instance γ, A sets $t_o.\tau = \gamma.r - \delta$ (the maximum event time within γ). Hence, output event times take values of the form $n\,wa + ws - \delta$ (for integer n). If A processes streams where event time advances at the same pace as wall-clock time, A emits one output every approximately wa time units (in both event and wall-clock time).

To decide when the correct result for a given γ can be produced, accounting for all γ's tuples, an SPE must know that no more tuples falling into γ are still to be processed. This can be ensured in two ways. If the SPE assumes timestamp-sorted streams [24], [27], correctness of the output of a γ 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* [23], [28], special tuples that signal event-time progress from each source's perspective. W_A^ω, the watermark of A at wall-clock time ω, is the earliest event time any future tuple t fed to A can have from ω onward. Sources periodically emit watermarks as special tuples with monotonically increasing timestamps [23], [28], indicating event-time progress from each source's perspective. Upon receiving a watermark W, A records it and updates W_A^ω to the minimum of the latest watermarks observed on all its input streams. When W_A^ω increases, A outputs all windows whose right boundary is not greater than W_A^ω and forwards the updated W_A^ω 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-Part 2, which we also include in our evaluation in Section 3.5 with a heavier aggregation function, Figure 3.2 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 Section 3.2), each γ spans 9 10-second ps, as shown in Figure 3.2(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 γ's output. In the example, A relies on watermarks to determine when to output each γ. For clarity, timestamps are shown using only their seconds component; thus, the first window is depicted as covering the interval $[180, 270)$. In Figure 3.2(f), watermark $W_A^{\omega_1}$ approaches 270, the right boundary of the γ whose output is to be returned, making tuples contributing to the γ spanning $[180, 270)$ ready for output. Once the watermark $W_A^{\omega_2}$ 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 γ spanning $[180, 270)$. Upon returning the output, γ advances by wa, and the corresponding pane state is updated to process subsequent tuples.

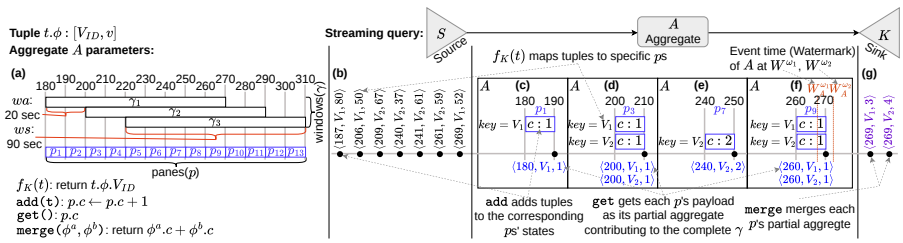


Figure 3.2: Sample A building on Example-Part 1 that counts per-vehicle reports over a window with wa and ws of 20s and 90s, respectively. The execution excerpt shows how A's functions process input tuples, maintain A's ps, and produce output tuples by merging the partial aggregates of all ps within the same γ.

3.3 Problem Formalization

As mentioned in Section 3.1, and illustrated in Figure 3.1, 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 Section 3.2, 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 γ lies entirely after time 0, and that $\delta = 1$ (see Section 3.2), as in state-of-the-art SPEs [23].

3.4 Algorithms

As mentioned in Section 3.1, 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 γs provides only coarse control, whereas a pane-based strategy (see Section 3.2) decomposes γs 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 γs it contributes to and, more importantly, the p where it is stored under the chosen semantics. This tuple-to-γ/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 γs and semantics-dependent p. For event time τ, we denote by $\Omega(\tau, (wa, ws))$ (or simply Ω, when notation can be simplified) the number of γs of advance wa and size ws to which a tuple t so that $t.\tau = \tau$ contributes. It holds that:

$$\Omega = \begin{cases} \left\lceil \frac{ws}{wa} \right\rceil, & \text{if } (\tau \bmod wa) < (ws \bmod wa) \ \vee \ (ws \bmod wa) = 0, \\ \left\lceil \frac{ws}{wa} \right\rceil - 1, & \text{otherwise.} \end{cases} \tag{3.1}$$

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

$$\gamma^i.l = \left(\left\lfloor \frac{\tau}{wa} \right\rfloor - (\Omega-1) + i\right) wa \tag{3.2}$$

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

$$\begin{aligned} p.l &= \max\left(\gamma^{\Omega-1}.l,\ \gamma^{-1}.r\right) \\ &= \max\left(\left\lfloor \frac{\tau}{wa} \right\rfloor wa,\ \left(\left\lfloor \frac{\tau}{wa} \right\rfloor - \Omega\right) wa + ws\right) \end{aligned} \tag{3.3}$$

In the following, `get_`γ`.l`$(\tau,(wa,ws))$ and `get_`p`.l`$(\tau,(wa,ws))$ denote the methods that compute the left boundary of a γ or a p, respectively, based on Eq.3.2 and Eq.3.3.

At-most-once (AMO)/exactly-once (EO) semantics and window configuration changes. According to our problem definition (see Section 3.3), 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 ps 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 γs 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, ps 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 ps 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., ps) that falls entirely in γs defined by the new configuration. For EO, we continuously maintain ps that can tile γs for every $(wa, ws) \in \mathbb{WA} \times \mathbb{WS}$, and produce outputs by merging the partial aggregates of those ps matching the current epoch's configuration.

While the γ 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:

$$p.l = \max_{(wa,ws)\in\mathbb{WA}\times\mathbb{WS}} p^{(wa,ws)}.l \tag{3.4}$$

In our algorithms, this logic is implemented by the method `get_EO_p.l`.

Since Eq.3.4 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 $O\Big(L \sum_{(wa',ws') \in \mathbb{WA} \times \mathbb{WS}} \frac{1}{wa'}\Big) \subseteq O\Big(L \frac{|\mathbb{WA} \times \mathbb{WS}|}{\min(\mathbb{WA})}\Big)$. Hence, EO requires up to $O\Big(\max(\mathbb{WS}) \frac{|\mathbb{WA} \times \mathbb{WS}|}{\min(\mathbb{WA})}\Big)$ panes in memory (per key) and up to $O\Big(ws \frac{|\mathbb{WA} \times \mathbb{WS}|}{\min(\mathbb{WA})}\Big)$ panes merged per output for the current epoch. These are analytical worst-case bounds; in practice, as we also show in Section 3.5, 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 Section 3.3), 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 $t_b.\tau \leq t_c.\tau$ could already have been processed, again breaking the cut. When `change` is invoked, A^* therefore tracks τ_{max}, the highest tuple timestamp seen, and sets τ^*, the new epoch's start time, to $\tau_{max} + 1$. Thus, τ^* 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 τ^* set to the latest processed timestamp plus one, any output at τ^* already conforms to $(wa, ws)^i$ (γs' right boundaries are exclusive, see Section 3.2), and any output with a larger timestamp to $(wa, ws)^{i+1}$. For WB implementations, though, since τ^* may be far from the last tuple's timestamp, outputs for γs with $\gamma.r \leq \tau^*$ 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 τ^* 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 τ^* 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 γs are fully populated. If Ω^{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 τ^*, the convergence time – i.e., the time during which result correctness is potentially affected by a reconfiguration – is:

$$\begin{cases} ws^{i+1}, & \text{if } \gamma^{\Omega^{i+1}-1}.l = \tau^*, \\ \gamma^{\Omega^{i+1}-1}.l + wa^{i+1} + ws^{i+1} - \tau^*, & \text{otherwise.} \end{cases} \tag{3.5}$$

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 Γ, where the c-th wa and ws are retrieved as $\Gamma[c].wa$ and $\Gamma[c].ws$, respectively. We also assume that `merge`(ϕ^a, ϕ^b), which combines the partial aggregates of two panes, returns the non-`null` value if exactly one of ϕ^a or ϕ^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 γ 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 γ 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).

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, τ_{max} is updated and t is added to its corresponding pane. The earliest left boundary of any γ maintained by A^* is also updated if the left boundary of the γ into which t falls precedes the current γ_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 τ^*. The method `getEarliestLeftBoundary` invokes `get_`γ`.l` and `get_AMO_`$p.l$ using index c. The method `reconfigAndAdvance` checks whether τ^* matches the timestamp of the current tuple (if τ^* is set). If so, it unsets τ^*, updates c to j, and updates the current γ_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

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
```

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
```

of any window under the new configuration.

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

3.5 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 variants (see Section 3.4). 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

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')
```

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
```

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')
```

state-of-the-art baselines. The first baseline is Liebre [24], 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 [23] 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 Section 3.2); 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 [39], representing a realistic use case with data generated by vehicles moving on highways. As stated in Section 3.3, 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 [43], 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 [39], 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 Section 3.2). 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 γ, `add(`t`)` incorporates incoming tuples and `get()` returns the number of non-consecutive stops within γ. 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

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 Section 3.1, 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 3.3 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 3×10^5 t/s at $ws = 90$ to roughly 7×10^4 t/s at $ws = 150$, whereas for $wa = 20$ it consistently exceeds 6.3×10^5 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.3.4). 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.3.5. 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 3.4 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 4×10^4 t/s for $wa = 1$ to approximately 2.3×10^5 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

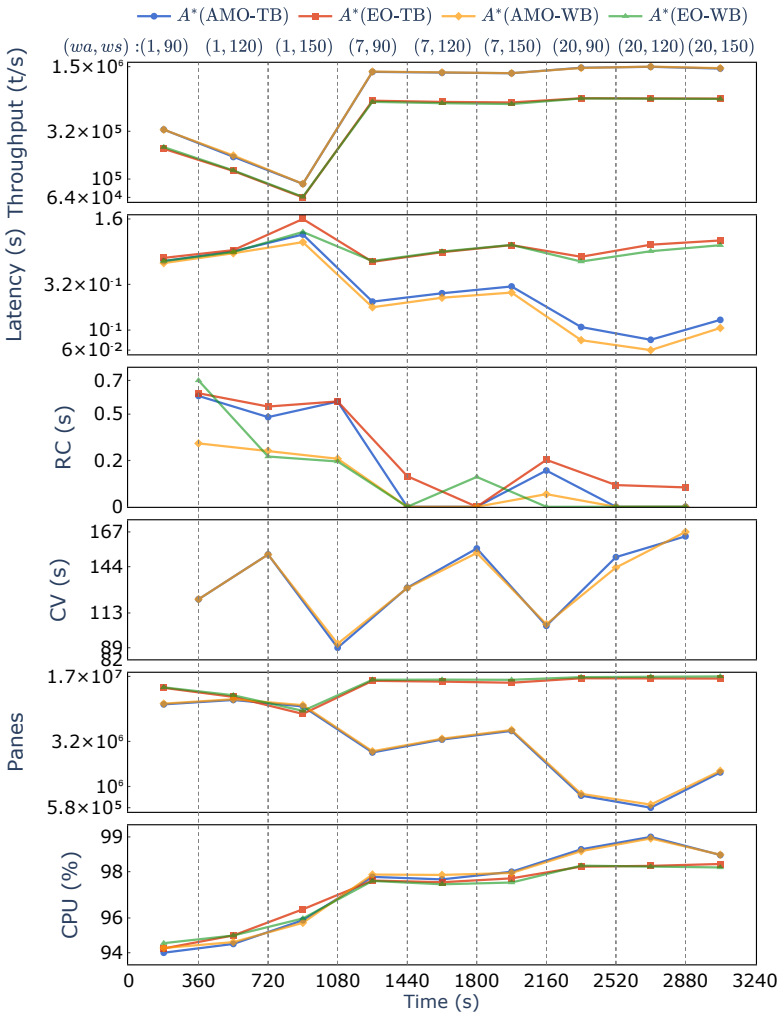


Figure 3.3: Window reconfiguration overhead for Synthetic.

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

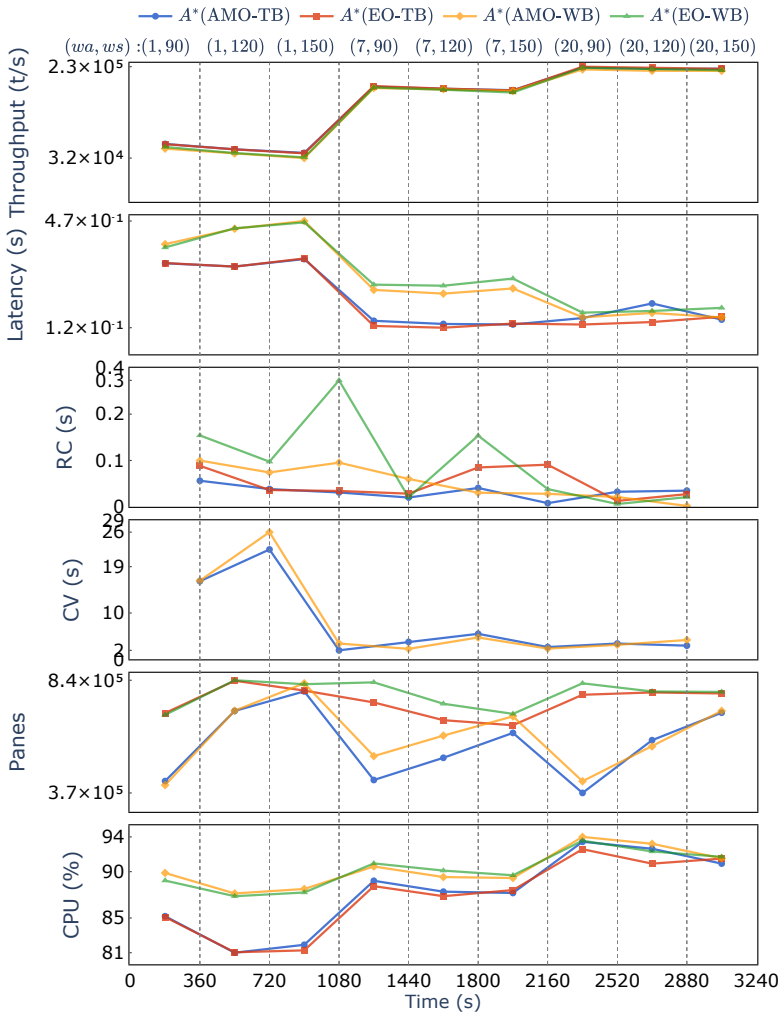


Figure 3.4: Window reconfiguration overhead for Linear Road.

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.

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 γs for every $(wa, ws) \in \mathbb{WA} \times \mathbb{WS}$ (see Section 3.4). 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 3.5 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.

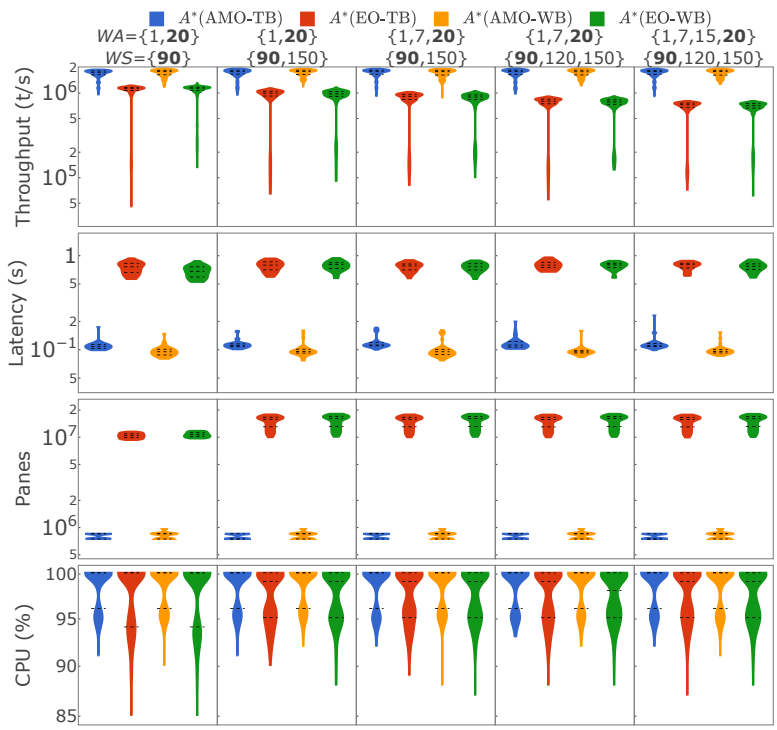


Figure 3.5: Performance impact of different $|\mathbb{WA} \times \mathbb{WS}|$ for Synthetic.

AMO exhibits limited sensitivity to the configuration space size, with metrics remaining relatively constant: Throughput hovering above 1.5×10^6 t/s, Latency remaining around 0.1 seconds, CPU close to saturation, and Panes stabilizing at approximately 7.5×10^5. 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 3.6 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.

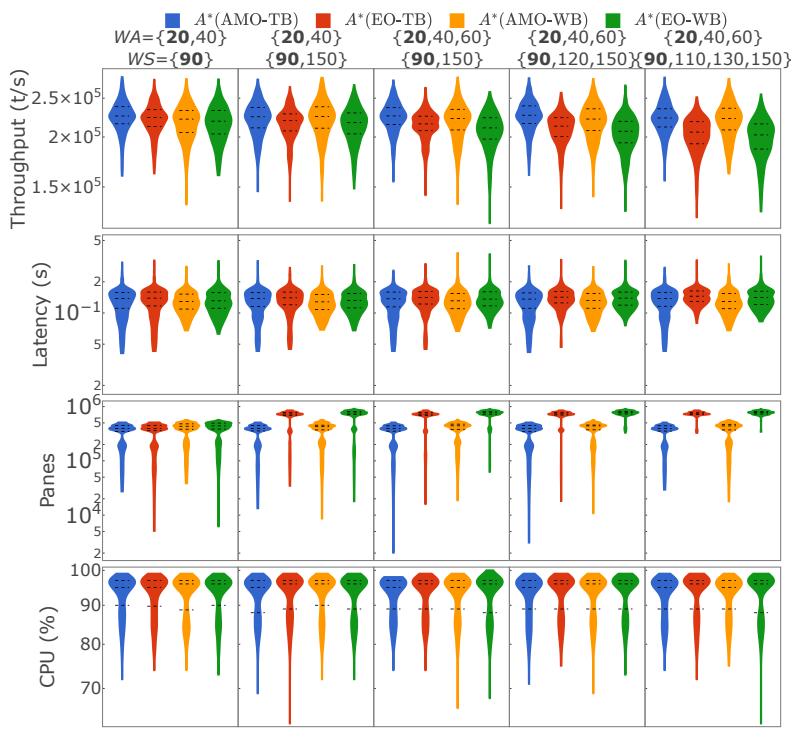


Figure 3.6: Performance impact of different $|\mathbb{WA} \times \mathbb{WS}|$ for Linear Road.

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 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 3.7 and 3.8 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 Section 3.2), and neither baseline relies on panes.

Figure 3.7 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 3×10^5 and 4×10^5 t/s, whereas Flink remains around 10^4 t/s. For $(20, 150)$, Liebre and A^* move into the 10^6–2×10^6 t/s range, while Flink reaches about 2×10^5 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 3.8 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 4×10^4 and 5×10^4 t/s, whereas Flink stays around 7×10^3 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 3×10^5 t/s, A^* around 2×10^5 t/s,

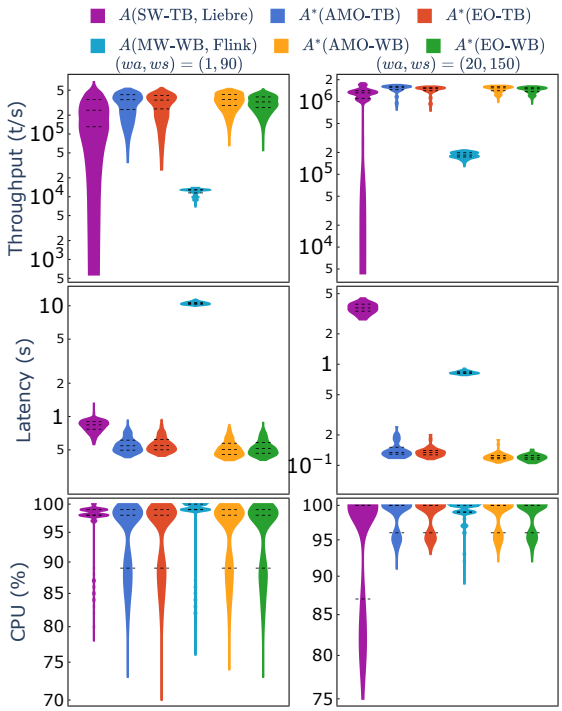


Figure 3.7: Violin plots of Throughput, Latency, and CPU across baselines and A^* variants for Synthetic.

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.3.5 in Section 3.4), 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

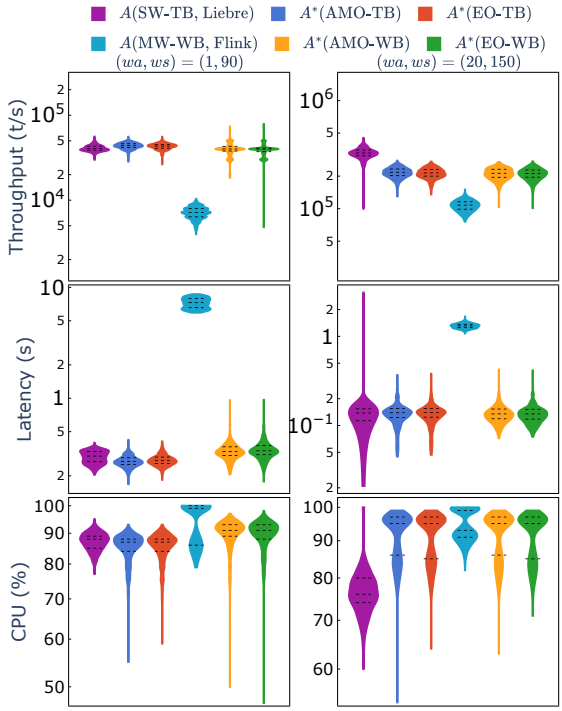


Figure 3.8: Violin plots of Throughput, Latency, and CPU across baselines and A^* variants for Linear Road.

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.

3.6 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** [75]: 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** [76]: 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** [18]: 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** [19]: 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** [20]: 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** [21]: 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** [22]: drop tuples or entire γs 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 [23]): 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 [79]; 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.

3.7 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 Section 3.4). 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.

Appendix A

Symbols and Abbreviations

Table A.1 collects all the symbols and abbreviations used in the thesis. Highlighted items indicate that they carry different meanings in Chapter 2 and Chapter 3.

Table A.1: Symbols and abbreviations used in the thesis.

Chapter 2	Chapter 3	Meaning
A	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
CT		Controller
	CV	Convergence time
DID		Data Injector Driver
DQN		Deep Q-Network
DRD		Data Receiver Driver
DX		Fixed-compression baselines, setting D to XW_{Size} of A
d		Fraction of A's W_{Size} the Agent uses to increase/-decrease compression
δ	δ	SPE-specific event time increments
EL-OB		Event time and Latency OBlivious state reporting policy
EM		Environment Monitor
	EO semantics	Exactly-once semantics
	epoch	A contiguous interval of event time during which a specific (wa, ws) pair is used by A^*
η		Interval (in number of steps) for granting positive/negative rewards
$f_{add}(\Gamma, t)$		Function to update window instance Γ with tuple t's contribution
$f_K(t)$	$f_K(t)$	Key extraction function for tuple t

Continued on next page

Table A.1 – Symbols and Abbreviations (*Continued from previous page*)

Chapter 2	Chapter 3	Meaning
$f_{out}(\Gamma)$		Function to define the tuple t_o output from window instance Γ
$f_{rm}(\Gamma, t)$		Function to remove tuple t's contribution from window instance Γ
Γ	γ	Window instance
γ		Discount factor of the cumulative reward for the Agent
	$\gamma.l$	Left boundary of the window instance γ
	$\gamma.r$	Right boundary of the window instance γ
	Γ	List of (wa, ws) combinations in $\mathbb{WA} \times \mathbb{WS}$
H		Moving average of the entropy of action probabilities
H_η		Monitoring period (in steps) for H
L-OB		Latency OBlivious state reporting policy
ℓ		Target latency threshold
ℓ_{max}		Maximum latency threshold allowed during the Agent execution
	MW	Multiple-Window execution strategy
N		Number of past measurements in wallclock time used in a reported state
n/c ratio		Non-compressed / compressed window instances ratio
OD		Operator Driver
	Ω	The set of γs of wa and ws to which a tuple $t.\tau = \tau$ contributes
	p	Pane instance
	$p.l$	Left boundary of the pane instance p
	$p.r$	Right boundary of the pane instance p
p_a		Action a's probability
$Q^\pi(s, a)$		State-action function
	RC	Reconfiguration time
R_F		Fixed positive reward given to the Agent for completing η steps
R_H		Fixed negative reward given to the Agent if H falls below T_H
RL		Reinforcement Learning
S	S	Stream
$\langle \mathbb{S}; \mathbb{A}; P; R \rangle$		Quadruple representing a RL model: $\mathbb{S}$ is the set of states, $\mathbb{A}$ is the set of actions, P is the state transition model, and R is the reward function
	SW	Single-Window execution strategy
SPE	SPE	Stream Processing Engine
T_H		Minimum entropy threshold
TD		Temporal Difference
t	t	Generic tuple
	t_i	Generic input tuple
$t.\phi$	$t.\phi$	Payload carried by tuple t
$t.\tau$	$t.\tau$	Timestamp carried by tuple t
t_o	t_o	Generic output tuple
	τ_{max}	The highest tuple timestamp observed by A^* during its execution
	τ^*	The event-time from which the new epoch starts when a reconfiguration is triggered
t/s		tuples/second

Continued on next page

Table A.1 – Symbols and Abbreviations (*Continued from previous page*)

Chapter 2	Chapter 3	Meaning
	TB implementation	Timestamp-based implementation
W	W	Generic watermark
W_A^ω	W_A^ω	Aggregate A's watermark W at wall-clock time ω
W_{Advn}	wa	Window Advance
	$\mathbb{WA}$	Set of acceptable wa values
	WB implementation	Watermark-based implementation
W_{Size}	ws	Window size
WEL-AW		Wallclock/Event time, and Latency AWare state reporting policy
WEL-OB		Wallclock/Event time, and Latency OBlivious state reporting policy
	$\mathbb{WS}$	Set of acceptable ws values

Bibliography

[1] F. Khan, N. Hosein, S. Ghiasi, C.-N. Chuah and P. Sharma, "Streaming solutions for fine-grained network traffic measurements and analysis," *IEEE/ACM Transactions On Networking*, vol. 22, no. 2, pp. 377–390, 2013.

[2] Y. Qiu, S. Papadias and K. Yi, "Streaming hypercube: A massively parallel stream join algorithm.," in *EDBT*, 2019, pp. 642–645.

[3] Q. Xiao, S. Chen, Y. Zhou and J. Luo, "Estimating cardinality for arbitrarily large data stream with improved memory efficiency," *IEEE/ACM Transactions on Networking*, vol. 28, no. 2, pp. 433–446, 2020.

[4] Y. Zhu and D. Shasha, "Statstream: Statistical monitoring of thousands of data streams in real time," in *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*, Elsevier, 2002, pp. 358–369.

[5] G. Luo et al., "Piecewise linear approximation of streaming time series data with max-error guarantees," in *2015 IEEE 31st international conference on data engineering*, IEEE, 2015, pp. 173–184.

[6] X. Yu, H. Xu, D. Yao, H. Wang and L. Huang, "Countmax: A lightweight and cooperative sketch measurement for software-defined networks," *IEEE/ACM Transactions on Networking*, vol. 26, no. 6, pp. 2774–2786, 2018.

[7] J. D. Taft and P. De Martini, "Sensing and measurement architecture for grid modernization," Pacific Northwest National Laboratory (PNNL), Richland, WA (United States), Tech. Rep., 2016.

[8] G. Weiss, "Data mining in the telecommunications industry," in *Encyclopedia of Data Warehousing and Mining, Second Edition*, IGI Global Scientific Publishing, 2009, pp. 486–491.

[9] N. Kayastha, D. Niyato, E. Hossain and Z. Han, "Smart grid sensor data collection, communication, and networking: A tutorial," *Wireless communications and mobile computing*, vol. 14, no. 11, pp. 1055–1087, 2014.

[10] D. Palyvos-Giannas, B. Havers, M. Papatriantafilou and V. Gulisano, "Ananke: A streaming framework for live forward provenance," *Proceedings of the VLDB Endowment*, vol. 14, no. 3, pp. 391–403, 2020.

[11] B. Havers, R. Duvignau, H. Najdataei, V. Gulisano, M. Papatriantafilou and A. C. Koppisetty, "DRIVEN: A framework for efficient data retrieval and clustering in vehicular networks," *Future Generation Computer Systems*, vol. 107, pp. 1–17, 2020.

[12] R. Duvignau, B. Havers, V. Gulisano and M. Papatriantafilou, "Querying large vehicular networks: How to balance on-board workload and queries response time?" In *2019 IEEE Intelligent Transportation Systems Conference (ITSC)*, IEEE, 2019, pp. 2604–2611.

[13] S. Lu and W. Shi, "Vehicle computing: Vision and challenges," *Journal of Information and Intelligence*, vol. 1, no. 1, pp. 23–35, 2023.

[14] D. Palyvos-Giannas, K. Tzompanaki, M. Papatriantafilou and V. Gulisano, "Erebus: Explaining the outputs of data streaming queries," in *Very Large Data Base*, vol. 16, 2023, pp. 230–242.

[15] B. Havers-Zulka, "Distributed and communication-efficient continuous data processing in vehicular cyber-physical systems," M.S. thesis, Chalmers Tekniska Hogskola (Sweden), 2020.

[16] T. Akidau et al., "The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing," *Proc. Endowment*, vol. 8, no. 12, pp. 1792–1803, 2015.

[17] M. Fragkoulis, P. Carbone, V. Kalavri and A. Katsifodimos, "A survey on the evolution of stream processing systems," *The VLDB Journal*, vol. 33, no. 2, pp. 507–541, 2024.

[18] M. Datar and R. Motwani, "The sliding-window computation model and results," in *Data Stream Management: Processing High-Speed Data Streams*, Springer, 2016, pp. 149–165.

[19] V. Q. Ngo and M. Papatriantafilou, "Cuckoo heavy keeper and the balancing act of maintaining heavy hitters in stream processing," *Proceedings of the VLDB Endowment*, vol. 18, no. 9, pp. 3149–3161, 2025.

[20] E. Cohen, G. Cormode and N. Duffield, "Structure-aware sampling on data streams," in *Proceedings of the ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, 2011, pp. 197–208.

[21] J. Liu and V. Gulisano, "On-demand memory compression of stream aggregates through reinforcement learning," in *Proceedings of the 16th ACM/SPEC International Conference on Performance Engineering*, 2025, pp. 240–252.

[22] N. Tatbul, "Qos-driven load shedding on data streams," in *International Conference on Extending Database Technology*, Springer, 2002, pp. 566–576.

[23] *Apache Flink*, `https://flink.apache.org`, Accessed:2024-11-01.

[24] *Liebre SPE*, `https://github.com/vincenzo-gulisano/Liebre`, Accessed:2024-11-01.

[25] *Apache Storm*, `https://storm.apache.org/`, Accessed:2024-11-01.

[26] *Apache Beam*, `https://beam.apache.org/`, Accessed:2024-11-01.

[27] V. Gulisano, Y. Nikolakopoulos, D. Cederman, M. Papatriantafilou and P. Tsigas, "Efficient data streaming multiway aggregation through concurrent algorithmic designs and new abstract data types," *ACM Transactions on Parallel Computing (TOPC)*, vol. 4, no. 2, pp. 1–28, 2017.

[28] V. Gulisano, D. Palyvos-Giannas, B. Havers and M. Papatriantafilou, "The role of event-time order in data streaming analysis," in *Proceedings of the 14th ACM International Conference on Distributed and Event-based Systems*, 2020, pp. 214–217.

[29] P. R. Geethakumari, V. Gulisano, B. J. Svensson, P. Trancoso and I. Sourdis, "Single window stream aggregation using reconfigurable hardware," in *2017 International Conference on Field Programmable Technology (ICFPT)*, IEEE, 2017, pp. 112–119.

[30] J. Traub et al., "Scotty: General and efficient open-source window aggregation for stream processing systems," *ACM Transactions on Database Systems (TODS)*, vol. 46, no. 1, pp. 1–46, 2021.

[31] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[32] L. Graesser and W. L. Keng, *Foundations of deep reinforcement learning: theory and practice in Python*. Addison-Wesley Professional, 2019.

[33] Y. Rizk, M. Awad and E. W. Tunstel, "Decision making in multiagent systems: A survey," *IEEE Transactions on Cognitive and Developmental Systems*, vol. 10, no. 3, pp. 514–529, 2018.

[34] D. P. Bertsekas and J. N. Tsitsiklis, "Neuro-dynamic programming: An overview," in *Proceedings of 1995 34th IEEE conference on decision and control*, IEEE, vol. 1, 1995, pp. 560–564.

[35] M. L. Puterman, "Markov decision processes," *Handbooks in operations research and management science*, vol. 2, pp. 331–434, 1990.

[36] R. S. Sutton, "Learning to predict by the methods of temporal differences," *Machine learning*, vol. 3, pp. 9–44, 1988.

[37] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, pp. 279–292, 1992.

[38] J. Fan, Z. Wang, Y. Xie and Z. Yang, "A theoretical analysis of deep q-learning," in *Learning for Dynamics and Control*, PMLR, 2020, pp. 486–489.

[39] A. Arasu et al., "Linear road: A stream data management benchmark," in *Proc. of the Thirtieth Int'l Conf. on Very large data bases-Volume 30*, VLDB Endowment, 2004, pp. 480–491.

[40] V. Gulisano, H. Najdataei, Y. Nikolakopoulos, A. V. Papadopoulos, M. Papatriantafilou and P. Tsigas, "Stretch: Virtual shared-nothing parallelism for scalable and elastic stream processing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 12, pp. 4221–4238, 2022.

[41] *NVIDIA Drive AGX Orin*, `https://nvidianews.nvidia.com/news/nvidia-introduces-drive-agx-orin-advanced-software-defined-platform-for-autonomous-machines`, Accessed:2024-11-01.

[42] B. Havers-Zulka, *Enhancing Localization, Selection and Processing of Data in Vehicular Cyber Physical Systems*. Chalmers University of Technology, 2024.

[43] V. Gulisano, "An algorithm for tunable memory compression of time-based windows for stream aggregates," in *European Conference on Parallel Processing*, Springer, 2023, pp. 18–29.

[44] Y. Wei, F. R. Yu, M. Song and Z. Han, "Joint optimization of caching, computing, and radio resources for fog-enabled iot using natural actor–critic deep reinforcement learning," *IEEE Internet of Things Journal*, vol. 6, no. 2, pp. 2061–2073, 2018.

[45] A. Staffolani, V.-A. Darvariu, P. Bellavista and M. Musolesi, "Rlq: Workload allocation with reinforcement learning in distributed queues," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 3, pp. 856–868, 2023.

[46] A. Mirhoseini et al., "Device placement optimization with reinforcement learning," in *International Conference on Machine Learning*, PMLR, 2017, pp. 2430–2439.

[47] R. Addanki, S. B. Venkatakrishnan, S. Gupta, H. Mao and M. Alizadeh, "Placeto: Learning generalizable device placement algorithms for distributed machine learning," in *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, 2019, pp. 3981–3991.

[48] X. Ni, J. Li, M. Yu, W. Zhou and K.-L. Wu, "Generalizable resource allocation in stream processing via deep reinforcement learning," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, 2020, pp. 857–864.

[49] P. R. Geethakumari, V. Gulisano, P. Trancoso and I. Sourdis, "Time-swad: A dataflow engine for time-based single window stream aggregation," in *2019 International Conference on Field-Programmable Technology (ICFPT)*, IEEE, 2019, pp. 72–80.

[50] G. Russo Russo, V. Cardellini and F. Lo Presti, "Hierarchical auto-scaling policies for data stream processing on heterogeneous resources," *ACM Transactions on Autonomous and Adaptive Systems*, vol. 18, no. 4, pp. 1–44, 2023.

[51] D. N. Doan, D. Zaharie and D. Petcu, "Auto-scaling for a streaming architecture with fuzzy deep reinforcement learning," in *Euro-Par 2019: Parallel Processing Workshops: Euro-Par 2019 International Workshops, Göttingen, Germany, August 26–30, 2019, Revised Selected Papers 25*, Springer, 2020, pp. 476–488.

[52] S. Bock, J. Goppold and M. Weiß, "An improvement of the convergence proof of the adam-optimizer," *arXiv preprint arXiv:1804.10587*, 2018.

[53] K. Arulkumaran, M. P. Deisenroth, M. Brundage and A. A. Bharath, "Deep reinforcement learning: A brief survey," *IEEE Signal Processing Magazine*, vol. 34, no. 6, pp. 26–38, 2017.

[54] T. Szandała, "Review and comparison of commonly used activation functions for deep neural networks," *Bio-inspired neurocomputing*, pp. 203–224, 2021.

[55] T. Li, Z. Xu, J. Tang and Y. Wang, "Model-free control for distributed stream data processing using deep reinforcement learning," *Proceedings of the VLDB Endowment*, vol. 11, no. 6, pp. 705–718, 2018.

[56] M. Hassani and A. Bouzeghoub, "Efficient constraint learning for stream reasoning," in *2023 IEEE 35th International Conference on Tools with Artificial Intelligence (ICTAI)*, IEEE, 2023, pp. 204–211.

[57] A. da Silva Veith, F. R. De Souza, M. D. de Assuncao, L. Lefèvre and J. C. S. Dos Anjos, "Multi-objective reinforcement learning for reconfiguring data stream analytics on edge computing," in *Proceedings of the 48th International Conference on Parallel Processing*, 2019, pp. 1–10.

[58] J. Xu and B. Palanisamy, "Model-based reinforcement learning for elastic stream processing in edge computing," in *2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, IEEE, 2021, pp. 292–301.

[59] *Apache Flink - The Broadcast State Pattern*, `https://nightlies.apache.org/flink/flink-docs-release-1.19/docs/dev/datastream/fault-tolerance/broadcast_state/`, Accessed:2024-11-01.

[60] D. Palyvos-Giannas, V. Gulisano and M. Papatriantafilou, "Haren: A Framework for Ad-Hoc Thread Scheduling Policies for Data Streaming Applications," in *Proceedings of the 13th ACM International Conference on Distributed and Event-Based Systems*, ACM, 2019, pp. 19–30.

[61] *Snappy-Java*, `https://github.com/xerial/snappy-java`, Accessed:2024-11-01.

[62] G. Brockman et al., "Openai gym," *arXiv preprint arXiv:1606.01540*, 2016.

[63] A. da Silva Veith, M. D. de Assunçao and L. Lefevre, "Monte-carlo tree search and reinforcement learning for reconfiguring data stream processing on edge computing," in *2019 31st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, IEEE, 2019, pp. 48–55.

[64] X. Huang et al., "Tata: Throughput-aware task placement in heterogeneous stream processing with deep reinforcement learning," in *2021 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)*, IEEE, 2021, pp. 44–54.

[65] G. Russo Russo, M. Nardelli, V. Cardellini and F. Lo Presti, "Multi-level elasticity for wide-area data streaming systems: A reinforcement learning approach," *Algorithms*, vol. 11, no. 9, p. 134, 2018.

[66] V. Cardellini, F. Lo Presti, M. Nardelli and G. Russo Russo, "Auto-scaling in data stream processing applications: A model-based reinforcement learning approach," in *New Frontiers in Quantitative Methods in Informatics: 7th Workshop, InfQ 2017, Venice, Italy, December 4, 2017, Revised Selected Papers 7*, Springer, 2018, pp. 97–110.

[67] G. R. Russo, V. Cardellini and F. L. Presti, "Reinforcement learning based policies for elastic stream processing on heterogeneous resources," in *proceedings of the 13th ACM International Conference on Distributed and Event-based Systems*, 2019, pp. 31–42.

[68] K. Thonglek, K. Ichikawa, C. Sangkeettrakarn and A. Piyatumrong, "Auto-scaling system in apache spark cluster using model-based deep reinforcement learning," *Heuristics for Optimization and Learning*, pp. 347–360, 2021.

[69] H. Hadian, M. Farrokh, M. Sharifi and A. Jafari, "An elastic and traffic-aware scheduler for distributed data stream processing in heterogeneous clusters," *The Journal of Supercomputing*, vol. 79, no. 1, pp. 461–498, 2023.

[70] L. Nie, Y. Qiu, F. Meng, M. Yu and J. Li, "Generalizable reinforcement learning-based coarsening model for resource allocation over large and diverse stream processing graphs," in *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, IEEE, 2023, pp. 435–445.

[71] Z. Zhang, T. Liu, Y. Shu, S. Chen and X. Liu, "Dynamic adaptive checkpoint mechanism for streaming applications based on reinforcement learning," in *2022 IEEE 28th International Conference on Parallel and Distributed Systems (ICPADS)*, IEEE, 2023, pp. 538–545.

[72] J. Tahir, "Config 2.0: Towards reinforcement learning based configuration of stream processing systems," in *Proceedings of the 22nd International Middleware Conference: Doctoral Symposium*, 2021, pp. 1–3.

[73] V. Pasquadibisceglie, A. Appice, G. Castellano, N. Fiorentino and D. Malerba, "Stardust: A novel process mining approach to discover evolving models from trace streams," *IEEE Transactions on Services Computing*, vol. 16, no. 4, pp. 2970–2984, 2022.

[74] S. Vaishnav and S. Magnússon, "Intelligent processing of data streams on the edge using reinforcement learning," in *2023 IEEE International Conference on Communications Workshops (ICC Workshops)*, IEEE, 2023, pp. 1265–1270.

[75] Y. Ahmad et al., "Distributed operation in the borealis stream processing engine," in *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, 2005, pp. 882–884.

[76] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, C. Soriente and P. Valduriez, "Streamcloud: An elastic and scalable data streaming system," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 12, pp. 2351–2365, 2012.

[77] N. Tatbul, U. Çetintemel, S. Zdonik, M. Cherniack and M. Stonebraker, "Load shedding in a data stream manager," in *Proceedings 2003 vldb conference*, Elsevier, 2003, pp. 309–320.

[78] C. Balkesen, N. Tatbul and M. T. Özsu, "Adaptive input admission and management for parallel stream processing," in *Proceedings of the 7th ACM international conference on Distributed event-based systems*, 2013, pp. 15–26.

[79] G. Cormode and S. Muthukrishnan, "An improved data stream summary: The count-min sketch and its applications," *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.