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

© Jingyu Liu, 2026
except where otherwise stated.
All rights reserved.

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

List of Figures

List of Tables

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.

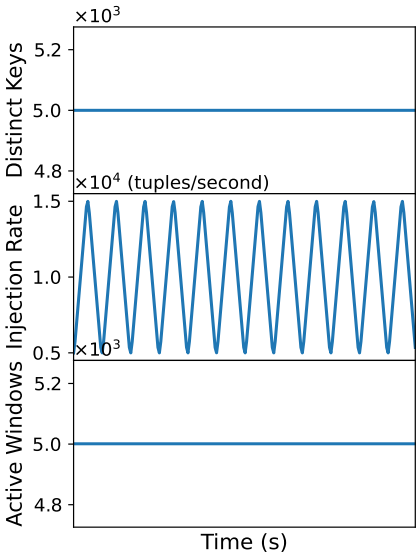
Depending on Aggregates' computational footprint and the available resources at each node, Aggregates may be deployed at different points in the pipeline. In data-center environments, SPEs typically handle time-varying workloads by executing tasks in a distributed mode across multiple nodes or by leveraging 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.

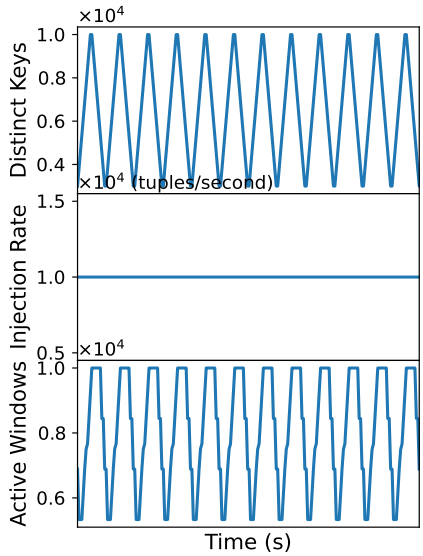
This motivates establishing mechanisms that can efficiently manage Aggregate resource usage while supporting safe and correct principled reconfiguration. Achieving this requires particular attention to two aspects of Aggregate behavior, which are especially critical in edge deployments:

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.

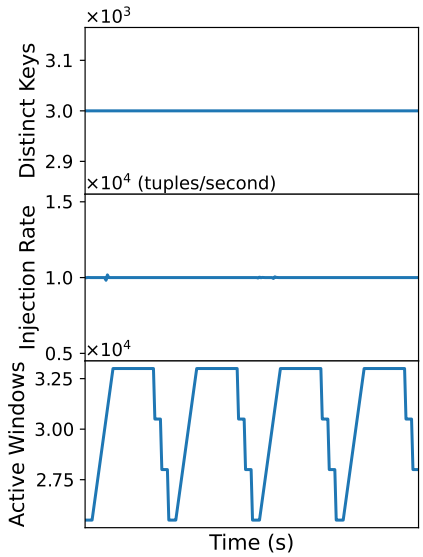
Window configuration. An alternative approach to relaxing Aggregates'



(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.

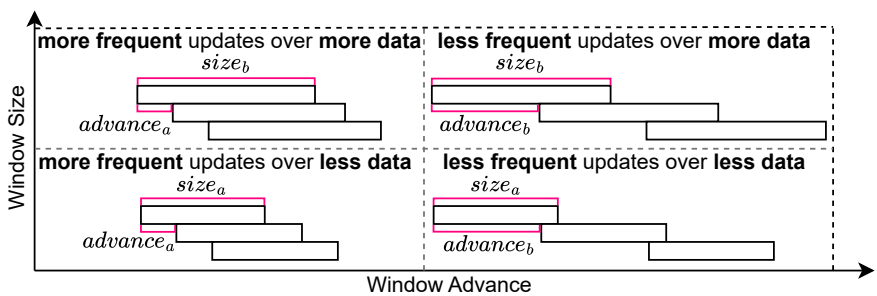


Figure 1.2: Computation cost under different window advances and window sizes.

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.