Balancing Performance and Memory

Investigating Compression Trade-offs for Stream Aggregates

Master's thesis in Computer science and engineering

YI LIU

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2025

Master's thesis 2025

Balancing Performance and Memory

Investigating Compression Trade-offs for Stream Aggregates

YI LIU

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2025

Balancing Performance and Memory
Investigating Compression Trade-offs for Stream Aggregates
YI LIU

© YI LIU, 2025.

Supervisor: Vincenzo Gulisano, Department of Computer Science and engineering
Examiner: Vincenzo Gulisano, Department of Computer Science and engineering

Master's Thesis 2025
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2025

Balancing Performance and Memory
Investigating Compression Trade-offs for Stream Aggregates
YI LIU
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Stream processing has become a cornerstone of real-time data analytics, particularly in edge-to-cloud computing environments where timely insights are crucial. Stream Aggregates, as fundamental stateful operators, maintain windowed state to compute summaries over continuous data streams. However, in resource-constrained edge deployments, managing memory efficiently while maintaining high performance remains a significant challenge.

Recent work has shown that compressing infrequently accessed window instances can reduce memory usage, but existing approaches rely on fixed, manually configured thresholds-parameters that are difficult to tune without prior knowledge of data characteristics such as reporting frequency and access patterns. Moreover, these methods often overlook the impact of different compression libraries on system behavior.

To address these limitations, this thesis presents a study on adaptive memory compression for stream aggregates. We first evaluate the performance trade-offs of three widely used compression libraries-Snappy, Zstandard (Zstd), and JZlib-within a stream processing context. Our results show that each library offers a distinct balance between compression efficiency and processing overhead, with Snappy providing superior throughput and latency at the cost of lower memory savings, while Zstd achieves better compression at the expense of higher CPU cost.

Building on these findings, we propose a dynamic, self-tuning mechanism that automatically adjusts the compression threshold D based on runtime feedback. Instead of requiring analysts to specify a fixed D, our approach allows them to define a target range for a performance metric-such as the non-compressed/compressed (n/c) ratio-and the system adapts D online using simple adjustment rules. This enables robust and predictable compression behavior under varying workloads, without requiring expert knowledge.

We implement and evaluate our approach using the Liebre stream processing engine and the Linear Road benchmark. Experimental results demonstrate that our adaptive mechanism effectively stabilizes the n/c ratio within user-defined bounds, with negligible performance overhead compared to an optimally tuned fixed-D configuration. The dynamic strategy proves resilient to workload fluctuations and configuration resets, making it suitable for real-world, unpredictable environments.

Keywords: stream processing, aggregate, memory compression.

Acknowledgements

I would like to express my deepest gratitude to my supervisor, Vincenzo Gulisano, for his invaluable guidance, continuous support, and encouragement throughout this journey. Research is never an easy task, but his insight, patience, and dedication made it not only manageable but truly rewarding. I am deeply grateful for the opportunity to work under his supervision.

I also thank Chalmers for providing an excellent academic environment and the resources necessary to carry out this work. I especially appreciate the departments study spaces-particularly the MT-TD datorsalar-where I spent many productive and inspiring afternoons working on this thesis.

Finally, I would like to extend my heartfelt thanks to my beloved grandparents. Your endless love, care, and unwavering support have been a constant source of strength. I could not have reached this point without you.

Yi Liu, Gothenburg, 2025-08-25

Contents

List of Figures **xi**

List of Tables **xiii**

1 Introduction **1**
1.1 Thesis Outline . . . 3

2 Preliminaries **5**
2.1 Stream Processing . . . 5
2.1.1 Batch Processing . . . 5
2.2 Streams . . . 6
2.2.1 Windows . . . 7
2.2.2 Aggregation . . . 8
2.2.3 Notions of Time and Watermark . . . 9
2.3 Memory Compression . . . 10
2.4 State Compression in Stream Aggregates . . . 10

3 Study Goals **13**

4 Methods **15**
4.1 Compression Libraries' Impact on Performance . . . 15
4.2 Dynamic Adjustment of D . . . 16
4.2.1 Compression and Decompression as D Changes . . . 16
4.2.2 Adapting D Based on User-Defined Metrics . . . 17
4.2.3 Adjustment Functions $A_m(D)$ and $A_M(D)$ . . . 17

5 Implementation **21**
5.1 System Architecture . . . 21
5.2 Integration of Compression Libraries . . . 22
5.3 Dynamic adjustment of D . . . 23
5.3.1 Compression and Decompression as D Changes . . . 23
5.3.2 Adapting D Based on User-Defined Metrics . . . 24

6 Results and Analysis **27**
6.1 Experimental Setup . . . 27
6.1.1 Hardware Configuration . . . 27

6.1.2 Dataset and Workload 27
6.2 Compression Libraries impact on Performance 28
6.2.1 General Trends Across Libraries 29
6.2.2 Performance Comparison Across Libraries 29
6.2.3 The 30-Second Threshold Effect 30
6.2.4 Quantitative Trade-off Analysis 30
6.3 Dynamic Adjustment of D 31
6.3.1 Stress-Testing the Adaptive Mechanism 32
6.3.2 Analysis of Control Behavior 32
6.3.3 Trends in Latency and Memory 33
6.3.4 Comparison with Fixed-D Baseline 33
6.3.5 Implications for Usability and Robustness 34

7 Related Work **37**

8 Conclusion and Future Work **39**

Bibliography **41**

List of Figures

2.1 Bounded data processing with a classic batch engine. A finite pool of unstructured data on the left is processed through a data pipeline, resulting in structured output on the right. *Note:* Adapted from "Streaming 101" by Tyler Akidau 6
2.2 Illustration of a sliding window: to count events over a 5-minute interval, a window of size 5 minutes is created. The window advances every 1 minute, resulting in overlapping intervals. 7
2.3 An aggregate operator maintains multiple window instances, grouped by key. Such operators can be distributed across multiple devices to facilitate scalable processing and reduce data transfer overhead. . . . 8
2.4 Illustration of compression eligibility: for a 4-hour window, if no update occurs in the last 2 hours (i.e., $D = 2$ hours), the window is eligible for compression. 11

4.1 Dynamic D adjustments are potentially applied every P time units or processed tuples, through A_m if the monitored metric falls below threshold m or A_M if it grows above threshold M. 19

6.1 Throughput, latency, n/c ratio, and memory metrics for the different compression libraries and increasing injection rates when D grows from 0 to 600 seconds. 28
6.2 Throughput, latency, and memory metrics ratio for Snappy over Zstd, for increasing injection rates and D values from 0 to 600 seconds. . . 31
6.3 Evolution of windows, n/c ratio, latency, and memory metrics over time for increasing throughput values, with dynamic D adjustments targeting range $[0.3, 0.4]$ for the n/c ratio and D reset to 0 or 30 (alternatively) every 3×10^6 processed tuples. Vertical lines show when such resetting is applied. 35

List of Tables

6.1 Performance comparison between dynamic D adaptation and fixed $D = 15$ configuration 33

1
Introduction

Stream processing has become a cornerstone of modern data-intensive applications, enabling real-time analysis and decision-making across a wide range of domains such as intelligent transportation, industrial IoT, financial trading, and smart city infrastructures. Unlike traditional database systems, where data is stored persistently and queries are executed on demand, stream processing systems operate under an inverted paradigm: queries are persistent and data is processed as it arrives. This shift allows for significantly reduced latency, efficient resource utilization, and continuous insight generation-advantages that are critical in time-sensitive applications.

The effectiveness of this model is evidenced by the widespread adoption of distributed stream processing engines (SPEs) such as Apache Flink [1] and RisingWave [2], which support complex, stateful computations over unbounded data streams. A key operator in these systems is the *stream aggregate*, which summarizes incoming data over time-based *windows*. These windows allow analysts to define meaningful time intervals (e.g., count vehicles per minute) and control the frequency of result generation. By deploying such operators across the *edge-to-cloud* spectrum, SPEs can perform computation close to the data source-reducing network bandwidth usage, improving response times, and enhancing privacy.

However, while stream processing excels in data center environments, its deployment on edge devices introduces significant challenges. Edge hardware such as embedded systems in vehicles, sensors, or roadside units often operates under strict constraints in terms of CPU power, memory capacity, and energy availability. In such environments, managing memory efficiently becomes a critical concern, especially for stateful operators like aggregates that maintain internal state over time. As the number of active windows grows, so does the memory footprint, potentially leading to performance degradation or even system failure if memory limits are exceeded.

An effective strategy to mitigate memory pressure is to compress the state of infrequently accessed window instances. This approach trades a small amount of CPU overhead (for compression and decompression) for substantial gains in memory savings. Recent research has demonstrated the feasibility of memory compression in stream aggregates [3], where a threshold parameter D determines when a window becomes eligible for compression based on its inactivity duration. However, existing approaches typically rely on a fixed, manually configured D, which requires analysts to have prior knowledge of data characteristicssuch as reporting frequency, arrival patterns, and workload dynamics-information that is often unavailable or variable

in real-world deployments.

Furthermore, these studies often overlook the impact of different compression algorithms on system behavior. Libraries such as Snappy, Zstandard (Zstd), and JZlib offer varying trade-offs between speed, compression ratio, and CPU cost. Selecting the right library and configuring it appropriately can have a significant effect on both performance and resource efficiency. Yet, there is limited empirical analysis comparing these options in the context of stream processing.

Motivated by these limitations, this thesis presents a study on memory compression for stream aggregates, with a focus on two key contributions:

1. **Evaluation of compression libraries:** We conduct an empirical comparison of three widely used compression techniques-Snappy [4], Zstd [5], and JZlib [6]-within a stream aggregate operator. Our analysis quantifies their impact on critical performance metrics such as throughput, latency, memory consumption, and compression frequency.

2. **Dynamic compression control:** We propose and evaluate a self-adaptive mechanism that dynamically adjusts the compression threshold D based on runtime feedback. Instead of requiring analysts to specify a fixed value, our approach allows them to define a desired target range for a chosen performance metric (e.g., the n/c ratio). The system then automatically tunes D using simple adjustment rules, making compression behavior robust to workload variations and accessible to users without expert knowledge.

Our work builds upon the baseline algorithm introduced in [3], extending it with both a comparative analysis across compression libraries and a novel adaptive control strategy. Through experiments using the Linear Road benchmark [7], we demonstrate that different libraries exhibit distinct performance profiles and that our dynamic approach effectively stabilizes compression behavior within user-defined bounds, with negligible performance cost.

The contributions of this thesis are therefore threefold:

1. A systematic evaluation of multiple compression libraries in the context of stream aggregate state management.

2. The design and implementation of an adaptive compression mechanism that dynamically tunes the compression threshold based on observed system metrics.

3. An empirical validation showing that adaptive tuning achieves stable and predictable memory behavior while incurring minimal overhead.

These results provide practical insights for developers and system designers seeking to balance memory efficiency and processing performance in edge-aware stream processing applications.

1.1 Thesis Outline

The remainder of this thesis is structured as follows:

1. Chapter 2 provides the necessary background on stream processing, focusing on stream aggregates, windowing semantics, watermarking, and the fundamentals of in-memory state compression. It also reviews the baseline compression algorithm upon which our work is built.

2. Chapter 3 presents our study goals and research questions.

3. Chapter 4 presents our research methodology, detailing the evaluation framework for compression libraries and the design of the dynamic D adjustment mechanism. We describe the control logic, adjustment strategies, and the rationale behind our design choices.

4. Chapter 5 outlines the implementation of our prototype system within the Liebre stream processing engine [8], including the integration of compression libraries, indexing structures, and the adaptive control module.

5. Chapter 6 presents the experimental evaluation using the Linear Road dataset. We analyze the performance trade-offs across different compression libraries and demonstrate the effectiveness of our dynamic tuning strategy under varying load conditions.

6. Chapter 7 discusses related work in the areas of memory compression, adaptive systems, and stream processing optimization, highlighting how our approach complements and extends existing solutions.

7. Finally, Chapter 8 summarizes the findings of this thesis, reflects on the limitations of the current work, and suggests directions for future research.

2

Preliminaries

This chapter introduces the foundational concepts necessary to understand the problem of memory-efficient stream processing and the design of adaptive compression mechanisms for stream aggregates. We begin by contrasting stream processing with traditional batch processing, then define core components such as streams, operators, windows, and time semantics. Finally, we present the concept of in-memory state compression and review its application in stream processing systems.

2.1 Stream Processing

Stream processing has emerged as a dominant paradigm for real-time data analysis in domains such as IoT, financial trading, and intelligent transportation systems. Unlike traditional systems that operate on static datasets, stream processing systems are designed to handle continuous, unbounded data flows, enabling timely insights and immediate responses to events.

2.1.1 Batch Processing

Traditional data processing systems, such as relational database management systems (DBMS) and frameworks like MapReduce, are based on a batch processing model. In this model:

- Data is collected and persisted in storage before any processing occurs.
- Users issue ad-hoc queries that are executed over the complete dataset.

A key characteristic of this paradigm is that data is persistent, while queries are temporary and reactive. Once a query completes, its results are produced, and the query itself is discarded.

While effective for historical analysis, batch processing has significant limitations in scenarios requiring low-latency responses. To approximate real-time behavior, users must repeatedly re-execute queries as new data arrives, which is inefficient and resource-intensive, especially when the underlying data changes only incrementally between runs.

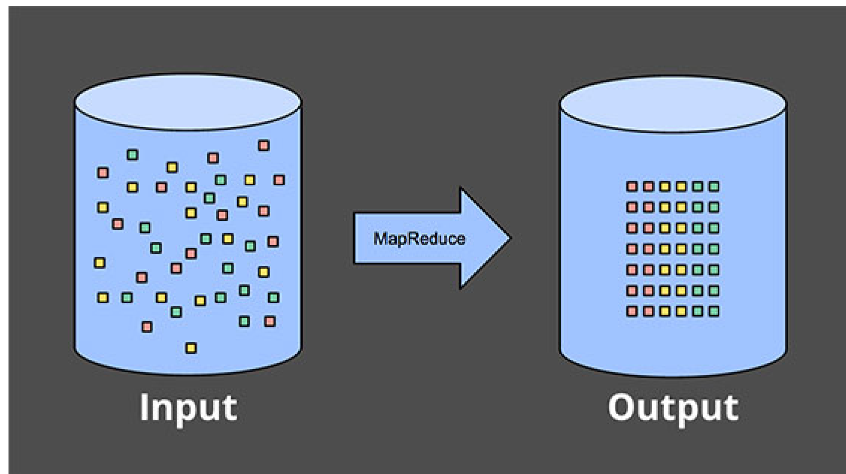

Figure 2.1: Bounded data processing with a classic batch engine. A finite pool of unstructured data on the left is processed through a data pipeline, resulting in structured output on the right. *Note:* Adapted from "Streaming 101" by Tyler Akidau .

2.2 Streams

In contrast to batch systems, stream processing reverses the roles of data and queries. In this model:

- Queries are defined in advance and remain *persistent*.
- Data arrives continuously and is processed *in real time*.

This shift enables immediate reaction to events as they occur, making it ideal for applications such as anomaly detection, live monitoring, and real-time recommendation systems.

A defining feature of stream processing is that the input data is *unbounded*-it does not have a predefined end and may continue indefinitely. This contrasts sharply with batch processing, where data is *bounded* and finite.

Formally, a *stream* S is an unbounded sequence of tuples, where each tuple t carries:

- A timestamp $t.\tau$, representing the event time of the data,
- A payload $t.\phi$, containing application-specific attributes [9].

A Stream Processing Engine (SPE) executes persistent queries composed of interconnected *operators*. These operators are arranged in a directed acyclic graph (DAG), forming a dataflow topology. Two special types of operators are:

- **Sources:** Inject tuples into the system (e.g., from sensors or message queues). A source has no input streams and one or more output streams.
- **Sinks:** Emit processed results to external systems (e.g., databases or dashboards). A sink has one or more input streams but no output streams.

General operators may have multiple inputs and outputs and perform transformations such as filtering, mapping, or aggregation.

Operators can be classified as:

- **Stateless operators:** Produce output based solely on individual input tuples (e.g., filtering or field transformation). They do not maintain internal state.
- **Stateful operators:** Maintain internal state that evolves over time based on multiple input tuples. Examples include aggregations, joins, and pattern detectors. These operators incur memory overhead and require mechanisms for fault tolerance.

Popular SPEs include Apache Flink [1], Kafka Streams[10], and RisingWave [2]. In this thesis, we use Liebre, a lightweight, research-oriented SPE implemented in Java, where each operator runs as a dedicated JVM thread. This architecture allows fine-grained control and instrumentation, making it well-suited for experimental evaluation.

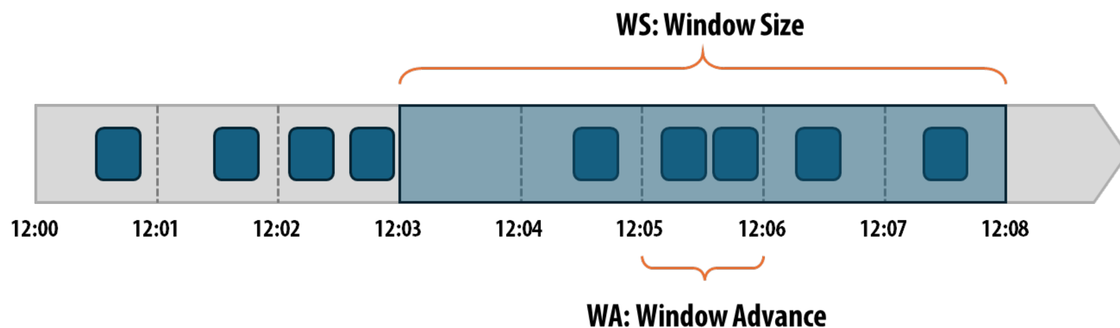

Figure 2.2: Illustration of a sliding window: to count events over a 5-minute interval, a window of size 5 minutes is created. The window advances every 1 minute, resulting in overlapping intervals.

2.2.1 Windows

Because streams are unbounded, direct aggregation (e.g., total count) is not meaningful. Instead, stream processing systems use *windows* to define finite subsets of data over which aggregations can be computed. A window partitions the stream into manageable segments, such as count over the last 5 minutes or sum of the most recent 100 tuples.

Common types of windows include:

- **Sliding windows:** Defined by two parameters: *window size* (WS) and *window advance* (WA). Each window covers a duration of WS, and the window moves forward by WA units of time (or count). If $WA < WS$, windows overlap; if $WA = WS$, they are non-overlapping.
- **Tumbling windows:** A special case of sliding windows where $WA = WS$. This results in contiguous, non-overlapping windows that partition the stream.

- **Session windows:** Group events based on periods of activity separated by gaps of inactivity. These windows are dynamically sized and are useful for tracking user sessions or transaction sequences.

2.2.2 Aggregation

The *Aggregate* is a fundamental stateful operator provided by most SPEs [1], [8], [11] to summarize data over time-based windows. An Aggregate A is defined by the following components:

Window Advance (WA), Window Size (WS): Define the event time intervals $[\ell \cdot WA, \ell \cdot WA + WS]$ covered by A, where $\ell \in \mathbb{N}$. If $WA < WS$, the windows are *sliding*; if $WA = WS$, they are *tumbling*; if $WA > WS$, they are *jumping*, and some tuples may not be assigned to any window.

Function $keyby(t)$ (optional): Enables partitioned processing by grouping tuples with the same key into independent window instances, allowing per-key aggregations.

Function $f_{add}(window, t)$: Adds the contribution of an input tuple t to the window instance $window$.

Function $f_{out}(window)$: Generates an output tuple t_o encapsulating the aggregated result from $window$.

Function $f_{rm}(window, t)$ (optional): Removes the contribution of tuple t from $window$. This is typically used in systems that support retractions or updates.

The f_{rm} function is optional because, in many implementations (especially for sliding windows), the system maintains multiple overlapping window instances per key and discards them after emission, rather than updating their contents.

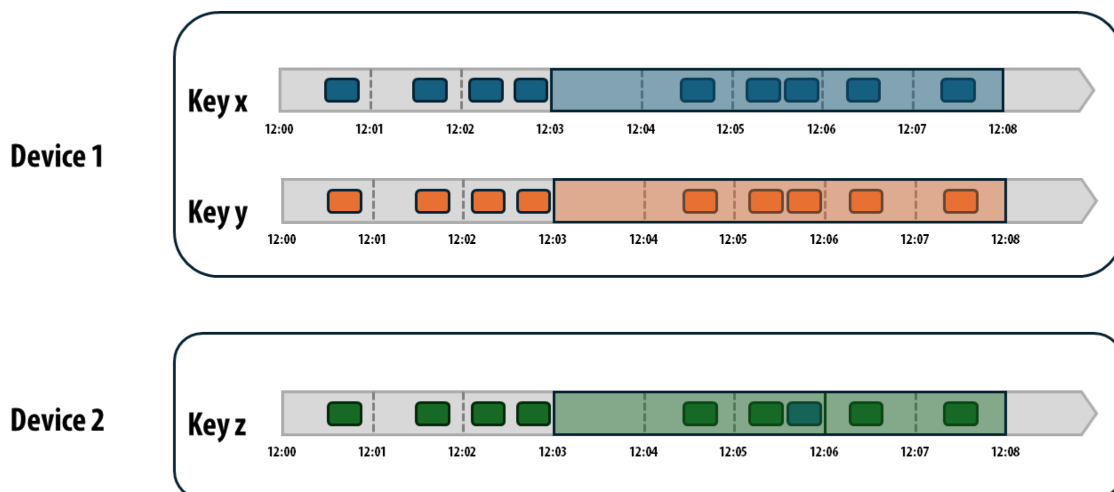

Figure 2.3: An aggregate operator maintains multiple window instances, grouped by key. Such operators can be distributed across multiple devices to facilitate scalable processing and reduce data transfer overhead.

2.2.3 Notions of Time and Watermark

Stream processing systems distinguish between two primary notions of time:

Processing time refers to the system clock time on the machine running the SPE. A window based on processing time includes all tuples that arrive within a given wall-clock interval.

Processing time is simple to implement and provides low-latency results. However, it lacks determinism: results may vary depending on system load or network delays. For example, replaying the same data at a different speed would produce different outputs. This makes it unsuitable for many analytical or regulatory applications.

Event time refers to the timestamp embedded in each tuple, typically assigned by the data source. Event-time processing enables correct results even when tuples arrive out of ordera common occurrence in distributed systems.

Event time decouples application logic from system timing, making it ideal for log replay, historical analysis, and consistent reporting.

A key challenge with event-time processing is handling out-of-order data. Since future tuples may carry earlier timestamps, the system must estimate when a window can be safely closed and its results finalized.

To solve this, SPEs use *watermarks*. A watermark $watermarkofA$ maintained by an operator A represents the systems estimate of the earliest event time that has not yet been observed. It is updated based on watermarks received from upstream sources or operators [12]. Sources emit watermarks as special control messages with monotonically increasing timestamps [1].

When $watermarkofA$ advances past the end of a window w (i.e., $watermarkofA \geq w.l + WS$), the operator emits the final result for w and discards its state. This indicates that no more tuples relevant to w are expected.

The choice of watermark strategy involves a trade-off:

- A **conservative** watermark (slow to advance) improves accuracy by allowing more time for late data but increases latency and memory usage.
- A **aggressive** watermark (fast advancing) reduces latency and memory pressure but risks missing late-arriving tuples.

To handle late data, systems like Flink allow users to specify an *allowed lateness*, a grace period during which late tuples can still be processed. However, retaining state for longer periods increases memory footprint, motivating the need for techniques like state compression.

2.3 Memory Compression

Memory compression is a technique used to reduce the physical size of data in RAM, thereby decreasing memory pressure and improving system efficiency. It is particularly valuable in resource-constrained environments such as edge devices.

There are two main types of compression:

- **Lossless compression:** Allows exact reconstruction of the original data from the compressed form. This is essential for applications where data fidelity is critical, such as databases, logs, and analytical systems.
- **Lossy compression:** Achieves higher compression ratios by discarding some information. It is commonly used in multimedia applications (e.g., images, audio) where minor quality loss is acceptable.

In the context of stream processing, where analytical accuracy is paramount, we focus exclusively on **lossless compression**. We evaluate three widely used libraries:

- **Zstandard (Zstd)** [5]: Developed by Meta, Zstd offers high compression ratios with fast decompression speeds. It balances compression efficiency and performance, making it suitable for a wide range of scenarios.
- **Snappy** [4]: A high-speed compression library developed by Google, based on the LZ77 algorithm. Snappy prioritizes low latency over high compression ratios, making it ideal for real-time systems.
- **JZlib** [6]: A pure Java reimplementation of the Zlib library, widely used in Java-based systems. Like Snappy, it is based on LZ77 variants and offers moderate compression performance.

All three libraries are optimized for fast decompression and are well-suited for in-memory use. While we do not delve into algorithmic details, their shared foundation in LZ77, a dictionary-based compression methodexplains their behavior in terms of speed and compression ratio.

2.4 State Compression in Stream Aggregates

We build upon the algorithm introduced in [3] for compressing the internal state of stream aggregates. The approach selectively compresses window instances based on their inactivity duration, controlled by a threshold parameter $D \in [0, +\infty]$.

Specifically, a window instance becomes eligible for compression if the time elapsed since its last update exceeds D, relative to the latest processed tuples event time. Once compressed, remains in a serialized and compressed form until it is accessed again (e.g., for an update or result emission), at which point it is decompressed.

The value of D controls the trade-off between memory usage and processing overhead:

- If $D = 0$: All windows are compressed immediately after each update. This maximizes memory savings but incurs high CPU cost due to frequent compression/decompression.
- If $D = +\infty$: No compression occurs, preserving fast access at the cost of higher memory usage.
- If $D \geq WS$: Since no window spans more than WS time units, no instance will ever qualify for compression under normal operation.

This mechanism is particularly effective when only a subset of windows (e.g., older or inactive ones) are accessed frequently. By compressing infrequently accessed state, the system reduces memory footprint with minimal impact on overall performance.

This baseline algorithm assumes a *static* D, which must be manually configured. As discussed in later chapters, our work extends this approach by introducing *dynamic* adjustment of D based on runtime metrics, eliminating the need for expert tuning.

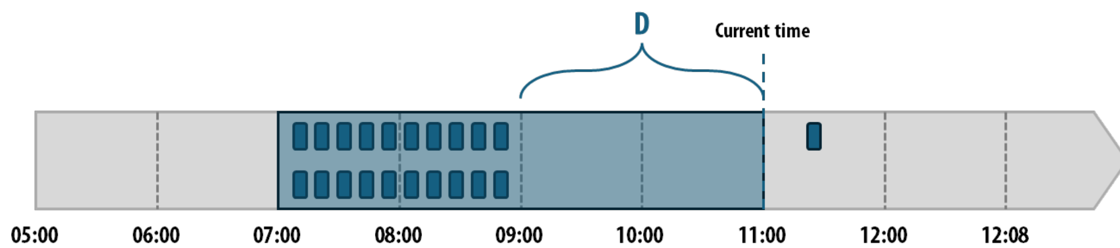

Figure 2.4: Illustration of compression eligibility: for a 4-hour window, if no update occurs in the last 2 hours (i.e., $D = 2$ hours), the window is eligible for compression.

3

Study Goals

This thesis focuses on improving memory efficiency in stream processing systems, particularly in resource-constrained environments such as edge computing. Stream Aggregates, a fundamental operator in modern Stream Processing Engines (SPEs), maintain internal state to compute summaries over time-based windows. As data streams grow and processing moves closer to the source, such as in vehicular networks or IoT systems, the memory footprint of these operators can become a critical bottleneck.

One promising approach to reduce memory usage is the selective compression of window instances that are infrequently accessed. Recent work [3] has demonstrated that such compression can significantly reduce memory consumption with acceptable performance overhead. However, two key limitations remain underexplored:

1. **Lack of comparative evaluation across compression libraries:** Most existing studies assume a single compression backend (e.g., Snappy) without comparing alternatives. In practice, libraries like Zstandard (Zstd) and JZlib offer different trade-offs between speed, compression ratio, and CPU cost. Understanding these differences is essential for making informed decisions in real-world deployments.
2. **Reliance on static, manually configured parameters:** The effectiveness of compression depends on a threshold parameter D, which determines when a window becomes eligible for compression. However, setting D requires prior knowledge of data characteristics, such as reporting frequency and access patterns, information that is often unavailable or variable in dynamic environments. A fixed D may perform well under certain conditions but degrade under workload shifts.

To address these limitations, this thesis aims to answer the following research questions:

RQ1: How do different compression libraries (Snappy, Zstd, JZlib) impact the performance of stream aggregates in terms of throughput, latency, memory usage, and compression frequency?

RQ2: Can we design a mechanism to dynamically adjust the compression threshold D based on runtime feedback, eliminating the need for manual tuning?

RQ3: How does such a dynamic adjustment strategy affect system stability, responsiveness, and overall performance compared to a fixed-D configuration?

Based on these questions, the study is structured around two main goals:

Empirical Evaluation of Compression Libraries: We conduct a systematic comparison of Snappy, Zstd, and JZlib within a stream aggregate operator. This evaluation quantifies the trade-offs between compression efficiency and processing overhead under varying workloads and D values.

Design of an Adaptive Compression Mechanism: We propose a feedback-driven control loop that dynamically adjusts D based on user-defined performance metrics, such as the n/c ratio or memory usage. Instead of requiring analysts to specify a fixed D, our approach allows them to define a target range (e.g., $[0.3, 0.4]$ for the n/c ratio), and the system automatically tunes D to stay within that range.

These goals are realized through an implementation in the Liebre stream processing engine [8], using the Linear Road benchmark [7] for evaluation. The adaptive mechanism is designed to be lightweight, modular, and transparent to the analyst, making it practical for real-world use.

4

Methods

4.1 Compression Libraries' Impact on Performance

To comprehensively understand the trade-offs inherent in different compression techniques, we conducted a comparative analysis of three libraries: **Snappy** [4], **Zstandard (Zstd) zstd**, and **JZlib** [6]. These libraries were selected due to their diverse design philosophies and practical relevance in real-world systems.

Each library exhibits unique performance characteristics, which we summarize as follows:

1. **Snappy**: A lightweight library based on the LZ77 algorithm, primarily optimized for high compression and decompression speeds at the expense of slightly lower compression ratios. It is suited for latency-sensitive applications.

2. **Zstd**: Developed by Facebook, this library provides a tunable balance between compression efficiency and processing speed, achieving ratios comparable to Zlib while offering faster performance. Its flexibility makes it ideal for scenarios requiring adaptive compression levels.

3. **JZlib**: A pure Java re-implementation of the Zlib library, also leveraging a variant of LZ77. It emphasizes broad compatibility and reliability but tends to exhibit slower compression/decompression speeds compared to more modern alternatives, making it suitable for environments prioritizing standardization over raw performance.

The selection of these libraries is motivated by their prevalence in production-grade systems and their representation of key design paradigms: speed prioritization (Snappy), balanced efficiency (Zstd), and compatibility (JZlib). By assessing them within the context of compressing stream aggregate states, this study aims to deliver analysts seeking to optimize resource utilization in edge-to-cloud deployments. To evaluate these libraries rigorously, we focused on a set of four core performance metrics, aligned with those established in prior work [3] for fair comparability:

1. **Throughput**: The rate at which tuples are processed, measured in tuples per second (t/s). This metric captures the overall processing capacity under compression overheads.

2. **Latency**: The end-to-end time elapsed from the ingestion of an input tuple

to the generation and emission of corresponding output results, quantified in seconds (s). It highlights the temporal cost introduced by compression and decompression operations.

3. **Memory Usage**: The total heap memory consumption attributed to the aggregate operator's state, reported in bytes. This directly reflects the effectiveness of compression in reducing footprint.

4. **Non-compressed/Compressed (n/c) Ratio**: The proportion of uncompressed window instances relative to the total number of maintained instances. This ratio serves as an indicator of compression aggressiveness and is independent of the specific library used when D is fixed.

These metrics were measured across a spectrum of experimental conditions to isolate library-specific impacts. Specifically, we varied the data injection rates from 5×10^3 t/s to 2×10^5 t/s, the latter representing the maximum sustainable rate for an uncompressed aggregate baseline. Concurrently, we explored a range of fixed compression threshold values D from 0 to 600 seconds (10 minutes), as higher values were observed to yield negligible compression (approaching no compression when $D \geq WS = 20$ minutes, as detailed in Chapter 2).

This experimental design enables a systematic dissection of how each library influences system behavior under increasing load and compression intensity. For instance, low D values enforce aggressive compression (potentially all windows compressed immediately post-update), maximizing memory savings but incurring frequent overheads, while higher D values favor performance by delaying or avoiding compression. By plotting metrics against these parameters (as visualized in subsequent figures), we identify inflection points, such as the sharp n/c ratio transitions around $D = 30$ seconds due to the dataset's reporting periodicity, and derive recommendations for library selection based on workload profiles.

4.2 Dynamic Adjustment of D

4.2.1 Compression and Decompression as D Changes

As discussed in Chapter 3, we aim to provide a mechanism that dynamically adjusts the compression threshold D based on a user-defined performance metric. The original algorithm in [3] assumes a static D: once a window has been inactive longer than D, it becomes eligible for compression, and no further updates occur if D later changes.

This assumption leads to undesirable behavior: D is *increased*, windows that had previously been compressed may now fall below the new inactivity threshold, but the system does not decompress them, potentially introducing unnecessary latency.

To address this shortcoming, we extend the original algorithm by actively re-evaluating all windows whenever D is updated. Specifically:

- If D is decreased, the system scans active windows to identify additional candidates for compression.
- If D is increased, the system inspects previously compressed windows and selectively decompresses those that no longer qualify under the new threshold.

This ensures that the system behavior remains consistent with the current D value and that performance trade-offs (e.g., latency vs. memory) are continuously aligned with user expectations.

4.2.2 Adapting D Based on User-Defined Metrics

As noted in Chapter 3, we allow D to be adjusted automatically in response to performance metrics selected by the user, such as memory usage, CPU load, or compression ratio. This is illustrated in Figure 4.1.

The update policy for D can be triggered in two ways:

1. **Tuple-driven:** D is adjusted after processing every P tuples (Algorithm 2).
2. **Time-driven:** D is adjusted at fixed time intervals (Algorithm 1).

In the tuple-based variant, a local counter p is initialized to P and decremented with each processed tuple. When $p = 0$, the system queries the current value of the selected performance metric, denoted by s, and compares it with the user-defined performance range $[m, M]$:

- If $s < m$, this indicates underutilization (e.g., low memory usage), and D is reduced using an adjustment function $A_m(D)$.
- If $s > M$, indicating system pressure (e.g., memory nearing a critical threshold), D is increased using $A_M(D)$.

After adjustment, p is reset to P, and processing resumes.

In the time-based variant, instead of maintaining a local counter, the system tracks a timestamp variable `lastDUpdate`, which records the wall-clock time of the most recent D update. The parameter P now represents the time interval (in milliseconds or seconds) after which D should be reconsidered. Each time a tuple is processed, the system checks whether the time elapsed since `lastDUpdate` is greater than or equal to P. If so, the system retrieves the relevant performance metric, adjusts D according to the same logic as in the tuple-based variant, and updates `lastDUpdate` to the current system time.

4.2.3 Adjustment Functions $A_m(D)$ and $A_M(D)$

The adjustment functions A_m and A_M define how aggressively the system reacts to metric deviations:

- **Linear Adjustment:** The simplest choice is to modify D linearly, e.g., $A_m(D) = D - \delta$ and $A_M(D) = D + \delta$, where δ is a fixed step size.

- **Multiplicative Adjustment:** For faster convergence, a multiplicative factor may be used, e.g., $A_m(D) = D \cdot \alpha$ and $A_M(D) = D \cdot \beta$, where $0 < \alpha < 1 < \beta$.
- **Clamped Range:** To prevent instability, D is bounded within a minimum and maximum range, i.e., $D \in [D_{\min}, D_{\max}]$.

These strategies can be selected or tuned based on deployment needs. In our experiments (see Section 6), we evaluate both fixed-step adjustments.

This feedback-based control loop enables the system to dynamically rebalance memory consumption and query latency based on runtime conditions, without requiring deep prior knowledge from the user.

Algorithm 1: Dynamic D adjustment algorithm (P expressed as wallclock time)

```
   Local Variables:
 1 lastDUpdate // time of last check/update of D, initially null
   Auxiliary Methods:
 2 getMetricOfInterest () // retrieve metric based on which D is checked/updated
 3 getSystemTime () // Retrieve the current wallclock time
 4 Method process(t)
 5   if getSystemTime() − lastDUpdate ≥ P then
 6     s ← getMetricOfInterest();
 7     if s ≠ null ∧ s < m then D ← A_m(D) ;
 8     if s ≠ null ∧ s > M then D ← A_M(D) ;
 9     lastDUpdate ← getSystemTime();
10     // Rest of algorithm from [3] follows
```

Algorithm 2: Dynamic D adjustment algorithm (P expressed as tuples)

```
   Local Variables:
 1 p ← P // tuples left before next check/update of D, initialized to P
   Auxiliary Methods:
 2 getMetricOfInterest () // retrieve metric based on which D is checked/updated
 3 Method process(t)
 4   p ← p − 1;
 5   if p = 0 then
 6     s ← getMetricOfInterest();
 7     if s ≠ null ∧ s < m then D ← A_m(D) ;
 8     if s ≠ null ∧ s > M then D ← A_M(D) ;
 9     p ← P;
10     // Rest of algorithm from [3] follows
```

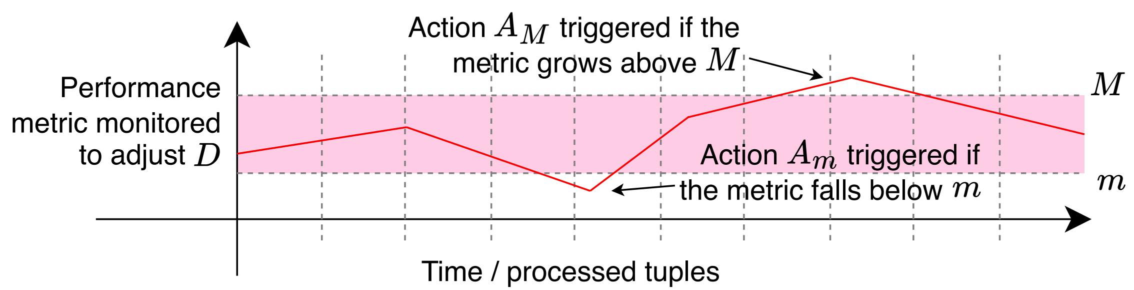

Figure 4.1: Dynamic D adjustments are potentially applied every P time units or processed tuples, through A_m if the monitored metric falls below threshold m or A_M if it grows above threshold M.

5

Implementation

This chapter details the implementation of the proposed enhancements, specifically the integration of multiple compression libraries and the adaptive adjustment mechanism for the compression threshold D. The algorithms are implemented within the Liebre stream processing engine [8], a lightweight, Java-based stream processing engine. In Liebre, each operator like source, aggregator, and sink executes as an Java Virtual Machine (JVM) thread.

5.1 System Architecture

This project builds upon the existing codebase from prior work [3], which already implements a baseline stream aggregate operator with tunable memory compression. The original architecture includes core components such as tuple ingestion, window management and basic compression logic. To extend this for our study, we introduced modular abstractions that decouple compression strategies from the aggregate's core logic, facilitating easy integration of diverse libraries and dynamic adaptations. At a high level, the system architecture comprises the following key elements:

- **Input Sources**: Generate or ingest tuples from streams, simulating real-time data (e.g., from the Linear Road dataset [7]).
- **Aggregate Operator**: Maintains window instances per key, applies aggregation functions, and triggers outputs based on timestamps. This is extended with a compression manager that selectively compresses/decompresses windows based on the event time distance threshold D.
- **Compression Module**: A pluggable layer (detailed in Section 5.2) that supports multiple libraries and handles state serialization/deserialization.
- **Adaptive Controller**: Periodically monitors specified metrics (e.g., n/c ratio) and adjusts D dynamically (detailed in Section 5.3).
- **Output Sinks**: Collect aggregated results for evaluation or further processing.

The architecture emphasizes modularity to ensure extensibility. For instance, the aggregate operator interacts with compression via an abstract interface, allowing

seamless swapping of libraries without altering the underlying window management logic. The full codebase, including these extensions, is available at `https://anonymous.4open.science/r/compression_debs` for reproducibility.

5.2 Integration of Compression Libraries

To incorporate multiple memory compression libraries into the existing codebase, we refactored the compression-related code. Specifically, we extracted the compression logic into a dedicated module and introduced a new interface called *WoostCompressor*. This interface defines two essential methods:

Algorithm 3: `WoostCompressor` Interface

window: A window instance in Liebre
compressedWindow: An object consists of the compressed byte array, the size in bytes of original window and the size of compressed byte array

```
1 compress (window) /* Compress the window and return the compressed byte array
    */
2 decompress (compressedWindow) /* Get the timestamp associated with a key     */
```

This abstraction ensures that the aggregate operator remains agnostic to the underlying compression implementation, promoting loose coupling and ease of maintenance. We implemented concrete classes for each evaluated library:

- **Snappy** and **Zstd**: Utilized the Java bindings from the *aircompressor* library [13], which provides efficient implementations. These were directly mapped to the *WoostCompressor* interface.

- **JZlib**: As a pure Java re-implementation of Zlib [6], it required a thin wrapping layer to align its API with our interface, ensuring consistent exception handling.

The integration process involved injecting the chosen compressor instance into the aggregate operator at initialization, configurable via parameters.

An additional note: During initial prototyping, we also benchmarked LZ4 [14], another high-speed compression library. However, its performance characteristics (throughput, latency, and compression ratios) were remarkably similar to Snappy's, as both are derived from comparable LZ77-inspired algorithms optimized for speed over ratio. To streamline the evaluation and avoid redundancy, we omitted LZ4 from the final experiments.

Decompression occurs symmetrically when accessing a compressed window (e.g., for output emission or further updates). This modular design not only supports the comparative study of libraries but also paves the way for future extensions, such as hybrid compression schemes.

5.3 Dynamic adjustment of D

5.3.1 Compression and Decompression as D Changes

In the original algorithm, two auxiliary data structures are used to efficiently identify windows that are eligible for compression. Since the window states are stored in a `HashMap<String, Window>`, the most straightforward approach is to iterate over the entire map and inspect the last update time of each window. However, this brute-force method incurs significant overhead, especially in systems with a large number of concurrent windows.

To avoid this, the original implementation maintains a secondary index using a `TreeMap<Long, Set<String»`, where each key represents a last-update timestamp, and the corresponding value is a set of window identifiers with that timestamp. Because `TreeMap` is ordered by key, the system can efficiently traverse only the windows whose last update time exceeds the compression threshold D, without scanning the entire state map.

However, because a window's last update time may change with every incoming tuple, the system also maintains a reverse index a `HashMap<String, Long>` to track the current update time of each window. This enables efficient removal and re-insertion of window keys in the `TreeMap` whenever updates occur.

This indexing strategy is effective, but in the original implementation, the logic for update-time tracking and compression decision-making is tightly coupled. This entanglement increases code complexity and the risk of implementation bugs. To address this, we modularize the update-time tracking logic by introducing a dedicated component: the `KeyTimestampTracker` class.

`KeyTimestampTracker` provides a clean interface for tracking keys (window identifiers) and their associated timestamps, independent of the compression logic. This abstraction not only improves code maintainability but also facilitates unit testing and reuse in other components of the system.

The interface is shown in Algorithm 4.

Algorithm 4: `KeyTimestampTracker` Interface

Data: Key-value pairs representing update timestamps for window keys

```
1 addOrUpdate (timestamp, key) /* Update the timestamp associated with a key;
     Remove old entry if key exists; Insert key under new timestamp        */
2 get (key) /* Get the timestamp associated with a key                    */
3 remove (key) /* Remove a key from tracking keys and timestamp            */
4 removeAll (keyList) /* Remove a list of keys and the corresponding timestamps
     from tracking                                                          */
5 getKeysBefore (t) /* Return all keys whose timestamp ≤ t Traverse TreeMap up to
     key t and return union of sets                                        */
6 getKeysAfter (t) /* Return all keys whose timestamp ≥ t Traverse TreeMap down
     to key t and return union of sets                                     */
```

With the `KeyTimestampTracker` in place, implementing window decompression based on a changing D value becomes straightforward. After processing a tuple—including

steps such as updating aggregation results and compressing stale windows—the system checks whether any previously compressed windows are now considered recent enough to be decompressed due to an increase in D.

This is achieved by querying the tracker for all keys (i.e., window identifiers) with update timestamps later than the threshold, defined as the current system time minus the updated D value. These windows have been updated recently enough that they no longer qualify as inactive under the new threshold. The system decompresses these windows and removes them from the tracker using `tracker.removeAll(keys)` to prevent redundant future checks.

It is also important to ensure that the tracker remains up-to-date. As each tuple is processed, the corresponding window's timestamp must be updated in the tracker to reflect its latest activity. This ensures correctness when the compression or decompression decision is made in subsequent iterations.

5.3.2 Adapting D Based on User-Defined Metrics

Having established the mechanisms for compressing and decompressing window instances in response to changes in the threshold D, we now describe how the system dynamically adjusts D itself based on runtime performance metrics specified by the user. This capability forms the core of our adaptive control framework, enabling the system to self-tune its behavior without requiring manual intervention or prior knowledge of data characteristics.

Metric Definition and Access The Liebre stream processing engine includes a flexible metrics module that allows users to define and expose arbitrary performance indicators. Rather than hardcoding a specific set of observable properties (e.g., memory usage or latency), our design follows a decoupled approach: the system provides the infrastructure for metric collection and access, while the responsibility for defining, computing, and updating relevant metrics is delegated to the application developer or analyst.

For instance, if the primary concern is memory efficiency, a natural choice of metric is the *compression ratio*, defined as the ratio of compressed window instances to the total number of windows maintained by the aggregate. A value close to 1 indicates aggressive compression and low memory footprint, while a value near 0 suggests minimal compression and higher memory consumption. To compute this metric efficiently, the operator can maintain two lightweight countersone tracking the number of compressed windows and another for uncompressed onesupdated atomically during compression and decompression operations.

Other possible metrics include estimated memory footprint, or even application-specific measures such as output delay or CPU utilization. This flexibility allows analysts to tailor the adaptation logic to their performance goals, whether prioritizing low latency, bounded memory usage, or energy efficiency in edge deployments.

Considerations for Memory-Based Metrics Special care must be taken when using memory consumption as a control metric within a Java Virtual Machine (JVM)

environment. Direct measurement of an objects memory footprint is inherently challenging due to the complexities of object layout, garbage collection, and internal JVM optimizations. Accurate estimation would require deep introspection of the object graph, which is both computationally expensive and potentially disruptive to real-time processing.

To avoid these overheads, we recommend using indirect, lightweight proxies for memory usage. For example, the total number of active window instances, combined with known average object sizes or compression ratios, can provide a sufficiently accurate approximation. Such estimations are inexpensive to compute and update, making them well-suited for frequent sampling in high-throughput scenarios.

Adjustment Strategy The adjustment of D is governed by two user-defined functions: $A_m(D)$, applied when the observed metric falls below a lower threshold m, and $A_M(D)$, applied when it exceeds an upper threshold M. These functions encode the control policy and determine how aggressively the system reacts to deviations from the desired operational range.

A common and intuitive strategy is *linear adjustment*, where D is incremented or decremented by a fixed step size δ:

$$A_m(D) = D + \delta, \quad A_M(D) = D - \delta$$

This approach is simple to configure and produces predictable, stable behavior. However, the optimal step size depends on the workload and the sensitivity of the system to changes in compression frequency. A large δ may lead to overshooting and oscillations, while a small δ could result in slow convergence.

To support more responsive adaptation, our implementation also allows for *multiplicative adjustment*, where D is scaled by a factor:

$$A_m(D) = D \cdot \beta \quad (\beta > 1), \quad A_M(D) = D \cdot \alpha \quad (0 < \alpha < 1)$$

This can accelerate convergence during periods of rapid change. Additionally, D is bounded within a user-specified range $[D_{\min}, D_{\max}]$ to prevent extreme values that could destabilize the system.

Integration into Tuple Processing With all components in place, integrating dynamic D adjustment into the operators execution flow is straightforward. The adaptation logic is embedded directly within the tuple processing method, executed either after every P processed tuples (tuple-driven) or at fixed time intervals (time-driven), as described in Chapter 4. After each update trigger, the system retrieves the current value of the selected metric, compares it against the target range $[m, M]$, and applies the appropriate adjustment function if needed.

This inline integration ensures low-latency feedback and maintains consistency with the stream processing model, where all state transitions are driven by incoming data or time progression. No external controllers or background threads are required, minimizing complexity and synchronization overhead.

6

Results and Analysis

6.1 Experimental Setup

Our evaluation framework was designed to measure the performance characteristics of compressed stream aggregates under realistic edge computing conditions. The following subsections detail our experimental methodology.

6.1.1 Hardware Configuration

All experiments were conducted on a dedicated server with an Intel Xeon E5-2637 v4 processor (4 cores, 8 threads @3.50GHz) and 64GB DDR4 RAM. This configuration was selected to represent capable edge computing hardware while maintaining consistency with the baseline established in prior work [3].

6.1.2 Dataset and Workload

We employed the Linear Road Benchmark [7], which simulates vehicle traffic across a 100-segment highway network. The dataset comprises approximately 44 million timestamped position reports over a 3-hour period (10,800 simulated seconds). Each record contains:

$$(Type = 0,\ Time,\ VID,\ Spd,\ XWay,\ Lane,\ Dir,\ Seg,\ Pos) \tag{6.1}$$

Where:

- $Type = 0$ indicates a position report
- $Time \in [0, 10799]$ marks the event timestamp (seconds)
- VID is the vehicle identifier
- Spd denotes current speed (0-100 mph)
- Remaining fields specify spatial coordinates

We normalized timestamps by adding 10,800 seconds to ensure immediate window emissions. The benchmark's natural 30-second reporting interval per vehicle creates inherent periodicity in the data stream. Our implementation focuses on detecting

stopped vehicles ($Spd = 0$) using sliding windows with $WS = 20$ minutes and $WA = 2$ minutes.

To ensure measurement accuracy, we implemented several safeguards:

- Excluded initial (1 minute) and final (several seconds) periods from measurements to account for warm-up and cool-down effects.
- Repeated each experiment configuration five times.

6.2 Compression Libraries impact on Performance

The experiments study varying injection rates: 5×10^3, 10×10^3, 20×10^3, 33×10^3, 50×10^3, 100×10^3, and 200×10^3 t/s. D values range from 0 to 600 seconds. We limit the max D value to 600 seconds since we observed in our experiments that higher values lead to almost no compression, with no actual compression once D is set to 20 minutes (i.e., when $D = WS$, see Section 2).

We evaluate three different compression techniques: *Snappy* [4], based on the LZ77 algorithm, and designed for high speed, *Zstandard (Zstd)* [5], which offers a compression ratio comparable to that of Zlib, and *JZlib* [6], a pure Java re-implementation of Zlib, which also relies on a variant of LZ77. By evaluating these different approaches, we aim to understand their impact on memory usage and processing efficiency within our stream Aggregate framework.

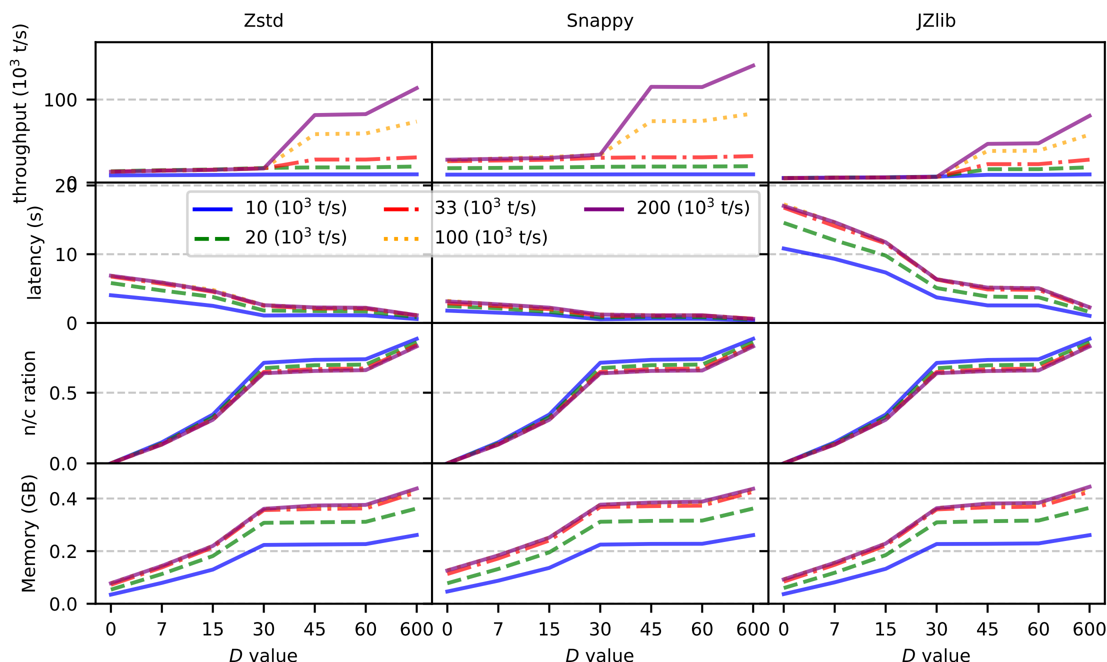

Figure 6.1: Throughput, latency, n/c ratio, and memory metrics for the different compression libraries and increasing injection rates when D grows from 0 to 600 seconds.

We now analyze how the choice of compression library affects the performance of the stream aggregate operator under varying configurations of the compression threshold D and input injection rates. The results, presented in Figure 6.1, provide a multi-dimensional comparison across four key metrics: throughput, latency, the non-compressed/compressed (n/c) ratio, and memory usage. Each column in the figure corresponds to one of the three evaluated libraries−Snappy, Zstd, and JZlib−while each row represents a different performance metric. Within each subplot, multiple lines (distinguished by color and line style) depict results at different injection rates, ranging from 10^3 to 200×10^3 t/s.

The x-axis shows the value of D (in seconds), spaced evenly to emphasize trends as compression aggressiveness changes. Recall that a smaller D leads to more aggressive compression (windows become eligible for compression sooner), while larger D values reduce compression frequency.

6.2.1 General Trends Across Libraries

As expected and consistent with prior work [3], increasing D leads to higher throughput, lower latency, and higher memory consumption. This behavior is directly tied to the amount of compression activity: when D is small, more window instances are compressed, increasing CPU overhead due to frequent serialization and compression operations. Conversely, when D is large (e.g., 600 seconds), few windows qualify for compression, resulting in minimal CPU cost but higher memory footprint.

This trend is clearly reflected in the n/c ratiothe proportion of non-compressed window instances relative to the total number of windows maintained by the operator. As D increases, the n/c ratio also increases, indicating that fewer windows are compressed. When $D = 600$, the n/c ratio approaches 1.0 across all injection rates, confirming that compression is nearly inactive. Consequently, the differences in latency and memory usage between $D = 0$ and $D = 600$ become less pronounced at high D values, as the system converges toward the behavior of an uncompressed aggregate.

6.2.2 Performance Comparison Across Libraries

A critical observation is the significant performance gap between the libraries, particularly under high load and aggressive compression settings ($D \ll 30$ s).

- **JZlib** consistently exhibits the worst performance across all metrics. It achieves the lowest throughput and highest latency, especially at high injection rates. This is attributable to its relatively slow compression and decompression routines, which impose a heavy CPU burden when compression is frequent.
- **Snappy** and **Zstd** both outperform JZlib significantly. Between the two, Snappy delivers higher throughput and lower latency, making it the best choice for latency-sensitive applications. However, it provides weaker compression, leading to higher memory usage compared to Zstd.

- **Zstd**, while slower than Snappy, offers superior compression efficiency, achieving better memory savings at the cost of increased CPU overhead. This makes it suitable for memory-constrained environments where CPU resources are less critical.

It is important to note that the n/c ratio is identical across all three libraries for the same D value. This is because the n/c ratio depends solely on D and the data access pattern (i.e., when windows are updated), not on the compression algorithm itself. The actual compression ratio (in terms of byte reduction) does differ between libraries, but this is not captured directly in the n/c metric.

6.2.3 The 30-Second Threshold Effect

One of the most notable patterns in Figure 6.1 is the emergence of a distinct inflection point around $D = 30$ seconds. Below this threshold, performance degrades rapidly with decreasing D, while above it, the system stabilizes.

This behavior is directly linked to the characteristics of the Linear Road dataset: vehicles report their position approximately every 30 seconds of event time. Therefore, when $D < 30$, most active windows (i.e., those associated with currently reporting vehicles) become eligible for compression shortly after being updated. This triggers frequent compression and decompression cycles, significantly increasing CPU load and reducing throughput.

Conversely, when $D \geq 30$, only inactive or long-idle windows are compressed, which belonging to vehicles that have stopped reporting. This results in much less frequent compression activity and thus lower overhead.

Interestingly, the n/c ratio does not increase linearly with D. Instead, it rises sharply around $D \approx 15$ seconds and then plateaus. This suggests that even moderate values of D can effectively reduce compression pressure while still maintaining reasonable memory usagea finding that informs the design of adaptive strategies discussed later.

6.2.4 Quantitative Trade-off Analysis

To quantify the trade-offs between Snappy and Zstd, we compute the ratio of their performance metrics, as shown in Figure 6.2. This figure plots the Snappy-to-Zstd ratio for throughput, latency, and memory usage across different D values and injection rates.

Key observations include:

- At high injection rates (e.g., 2×10^5 t/s) and low D values (e.g., $D = 0$), Snappy achieves up to **twice the throughput** of Zstd and reduces latency by nearly **50%**.
- However, this performance advantage comes at a cost: Snappy uses approximately **60% more memory** than Zstd under the same conditions.

This highlights a fundamental trade-off: Snappy prioritizes processing efficiency at the expense of memory savings, while Zstd favors memory efficiency with higher CPU cost. The choice between them should therefore depend on the deployment context-edge devices with tight memory budgets may prefer Zstd, while systems with ample RAM but strict latency requirements may benefit from Snappy.

These results underscore the importance of not only choosing the right compression algorithm but also dynamically adapting its use based on runtime conditions—a motivation we explore in the next section.

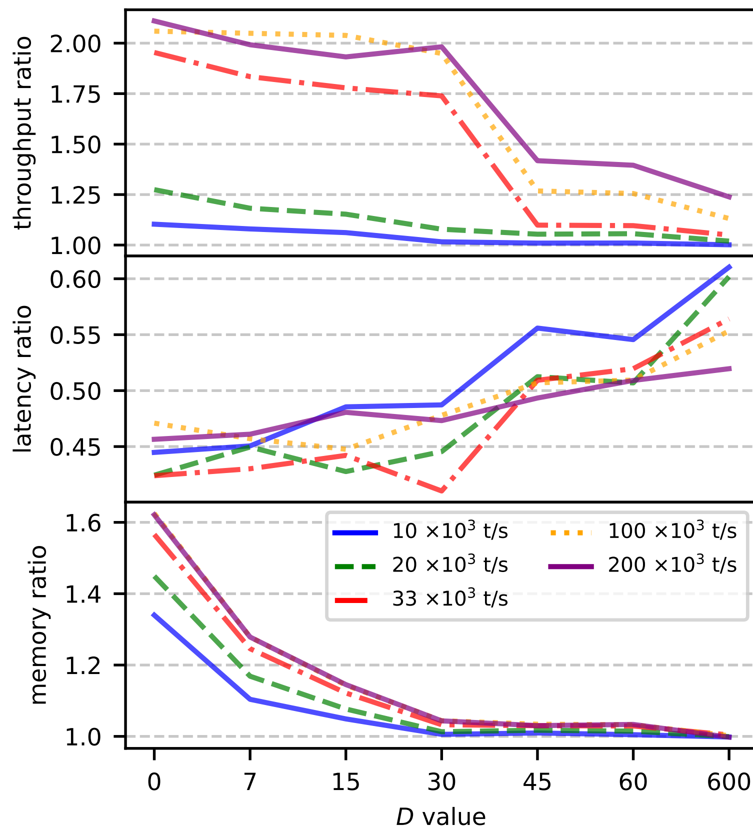

Figure 6.2: Throughput, latency, and memory metrics ratio for Snappy over Zstd, for increasing injection rates and D values from 0 to 600 seconds.

6.3 Dynamic Adjustment of D

We now evaluate the effectiveness of our adaptive compression mechanism from the perspective of a realistic analyst who lacks detailed knowledge of the input data characteristics—such as exact vehicle count, reporting frequency, or access patterns-but still wishes to maintain predictable system behavior.

In this scenario, the analyst can observe high-level workload indicators (e.g., incoming data volume). However, they do not know the precise periodicity (every 30 seconds) or the full distribution of vehicle activity. Rather than requiring them

to manually tune D, our adaptive mechanism allows them to express their intent through high-level control parameters:

- A target range for the **n/c ratio**: $[m, M] = [0.3, 0.4]$,
- An adjustment periodicity: $P = 10^4$ tuples (i.e., every 10,000 processed tuples),
- Adjustment functions: $A_m(D) = D - 1$, $A_M(D) = D + 1$ (linear step of 1 second),
- Compression backend: *Snappy*, selected for its favorable speed-compression trade-off.

This configuration instructs the system to keep the proportion of uncompressed window instances between 30% and 40% by dynamically increasing or decreasing D as needed. The goal is to stabilize memory usage and processing overhead without requiring expert-level insight into the data.

6.3.1 Stress-Testing the Adaptive Mechanism

To rigorously evaluate the robustness and responsiveness of the adaptive controller, we introduce artificial, periodic disruptions to D. Specifically, every 3×10^6 input tuples, we forcibly reset D to either 0 seconds (maximal compression) or 30 seconds (minimal compression), alternating between the two values. This intervention simulates sudden changes in workload or misconfigurations and tests whether the system can recover and re-stabilize within the desired n/c ratio range.

The experiment runs for a 10-minute excerpt of the Linear Road dataset under three different sustained throughput levels: approximately 10×10^3, 18×10^3, and 28×10^3 t/s. These rates were chosen to represent light, moderate, and heavy but sustainable workloads. Results are presented in Figure 6.3, which shows the evolution of four key metrics over time: total number of window instances, n/c ratio, latency, and memory usage.

Vertical dashed lines indicate the points at which D is forcibly reset, serving as perturbation events.

6.3.2 Analysis of Control Behavior

The most critical result is that, despite repeated and extreme external disruptions, the adaptive mechanism consistently brings the n/c ratio back into the target range $[0.3, 0.4]$ within a short recovery period. This demonstrates the controllers robustness and self-stabilizing behavior, which are key qualities for deployment in unpredictable edge environments.

When D is reset to 0, a sharp downward spike occurs in the n/c ratio (indicating aggressive compression), followed by a gradual increase as the controller detects under-utilization ($s < m$) and applies $A_m(D) = D + 1$ to reduce compression aggressiveness. Conversely, when D is reset to 30, an upward spike appears (fewer

windows compressed), prompting the system to apply $A_M(D) = D - 1$ until the ratio falls back into the target band.

Notably, the duration of these transients (spikes) decreases with higher throughput. At 28×10^3 t/s, the system recovers faster because the adjustment trigger ($P = 10^4$ tuples) is reached more frequently, enabling more rapid feedback. This highlights a beneficial side effect of high load: faster control loop responsiveness.

6.3.3 Trends in Latency and Memory

As expected, both latency and memory usage exhibit a general upward trend over time due to the increasing number of active vehicle keys and corresponding window instances maintained by the aggregate. This reflects the growing state footprint as more vehicles enter the monitored highway network.

Superimposed on this trend are oscillations synchronized with the D resets:

- When $D = 0$, frequent compression increases CPU overhead, slightly elevating latency.
- When $D = 30$, fewer windows are compressed, leading to higher memory consumption.

However, the magnitude of these oscillations remains small. Latency fluctuations are particularly well-damped, indicating that the system quickly absorbs performance disturbances. This is crucial for real-time applications where consistent response times are essential.

6.3.4 Comparison with Fixed-D Baseline

To assess the overhead of dynamic adaptation, we compare the results against a fixed-D baseline. We select $D = 15$ seconds because it yields an n/c ratio of approximately 0.35, which is centered within the target range $[0.3, 0.4]$, and was identified during the fixed-parameter evaluation (Section 6.2) as a reasonable manual configuration.

Injection Rate (t/s)	**Latency (s)**	**Memory (GB)**
	Dynamic / Fixed	*Dynamic / Fixed*
10×10^3	1.1 / 1.2	0.13 / 0.14
20×10^3	1.5 / 1.6	0.19 / 0.20
30×10^3	2.0 / 2.2	0.25 / 0.25

Table 6.1: Performance comparison between dynamic D adaptation and fixed $D = 15$ configuration

Table 6.1 compares the average latency and memory usage under dynamic control versus the fixed-D configuration at similar injection rates.

The results show that the dynamically adjusted system achieves nearly identical performance to the optimally tuned fixed-D case, with only negligible differences in

both latency and memory. In fact, in some cases (e.g., at 30×10^3 t/s), the dynamic version uses slightly less memory due to fine-grained tuning.

This confirms a key claim of our work: adaptive control incurs negligible runtime cost while providing significant usability benefits. Analysts no longer need to perform error-prone manual tuning or rely on prior knowledge of data characteristics. Instead, they can specify desired behavior, and the system automatically maintains it under varying conditions.

6.3.5 Implications for Usability and Robustness

These findings have practical implications:

- The adaptive mechanism is resilient to configuration errors and workload shifts.
- It enables predictable resource usage even in the face of limited uncertainty.
- It reduces the cognitive load on analysts, making compression accessible to non-experts.

While this experiment uses the n/c ratio as the control metric, the framework supports other metrics (e.g., memory usage, CPU load), paving the way for future multi-objective controllers.

In summary, the dynamic adjustment of D successfully decouples high-level performance goals from low-level system parameters, achieving stable, efficient, and user-friendly memory management in stream aggregates.

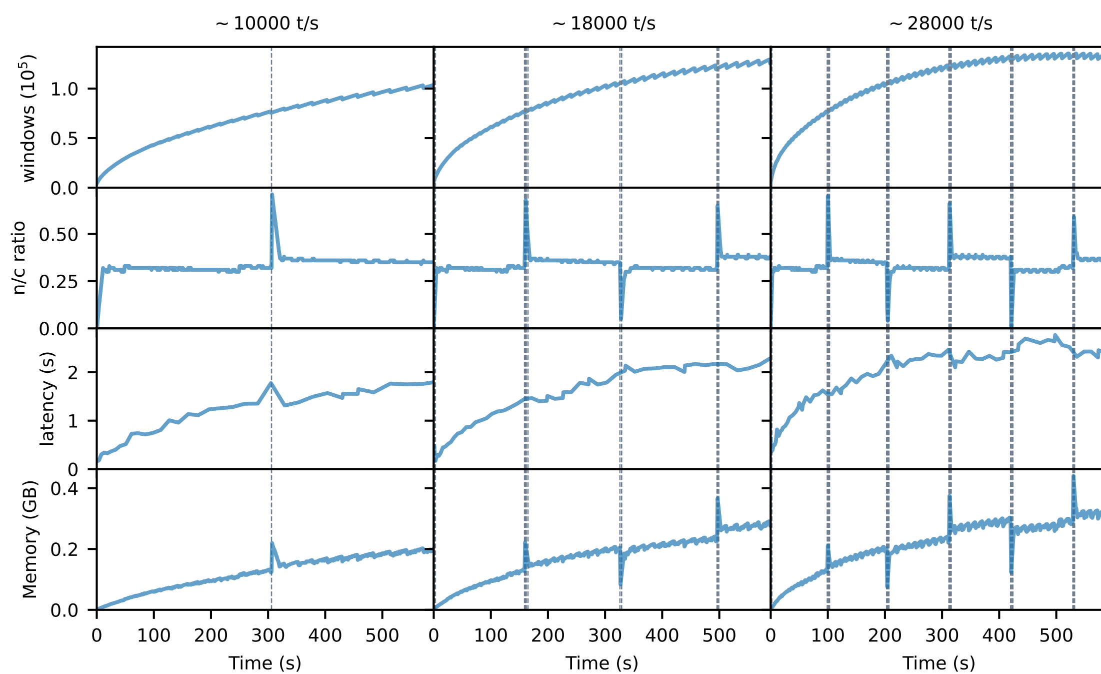

Figure 6.3: Evolution of windows, n/c ratio, latency, and memory metrics over time for increasing throughput values, with dynamic D adjustments targeting range $[0.3, 0.4]$ for the n/c ratio and D reset to 0 or 30 (alternatively) every 3×10^6 processed tuples. Vertical lines show when such resetting is applied.

7

Related Work

To the best of our knowledge, this is the first work that proposes a threshold-based, self-adaptive mechanism for compressing the internal state of stream Aggregates. Specifically, the window instances that store intermediate aggregation results over time-based intervals.

The most closely related work to our study is the algorithm presented in [15], which explores the use of Reinforcement Learning (RL) to automate the tuning of the compression threshold D in stream aggregates. While this approach can learn optimal policies without user-defined parameters, it requires a training phase and introduces complexity in terms of stability and deployment. In contrast, our feedback-driven mechanism offers a lightweight, transparent, and immediately deployable solution that allows analysts to define intuitive performance goals. The two approaches are thus complementary: our method provides a practical alternative for scenarios where simplicity and predictability are prioritized, while RL-based control offers a path toward fully autonomous tuning in more dynamic environments.

Both this work and [15] are based on [3]. This algorithm introduces the idea of selectively compressing window instances in a stream Aggregate based on a time threshold D: a window is eligible for compression if no update has occurred within the last D units of event time. However, the original algorithm assumes that D is a static, pre-configured parameter that remains unchanged throughout the execution of the application. This means that the analyst must manually select an appropriate value for D before deployment, based on prior knowledge of the data characteristics, such as reporting frequency, arrival patterns, and workload dynamics.

Another relevant piece of research is [16], which explores a different but complementary direction in memory-efficient stream processing. The authors propose a technique called Tersecades, which enables direct operations on compressed data without requiring full decompression. This approach aims to minimize both memory footprint and CPU overhead by avoiding the costly step of decompressing data before processing. Tersecades focuses on operating on compressed data streams rather than managing the internal state of stateful operators like Aggregates. Our work differs in that we focus on when and how often to compress state, not on how to compute over compressed representations. The two approaches are therefore not mutually exclusive and could potentially be combined in future systems to achieve even greater efficiency.

Expanding the scope beyond state compression in Aggregates, there are several works that investigate memory compression applied to other components of streaming applications. For instance, some studies focus on compressing the input data stream itself rather than the operator state [17], [18]. These approaches aim to reduce network bandwidth and ingestion overhead by compressing tuples before they enter the processing pipeline. These research applying compression on streams rather than on the states maintained by stateful operators.

Other research efforts have explored the use of compression for fault-tolerance mechanisms in stream processing engines. For example, Scabbard [19] applies compression to the state that is periodically checkpointed to persistent storage, reducing the amount of data that needs to be written to disk. While this improves the efficiency of fault-tolerant operations, it does not alleviate real-time memory pressure during normal operation. Our approach, on the other hand, operates on in-memory state during active processing, providing continuous memory relief rather than just optimizing storage costs.

Additionally, there are works that explore compression techniques tailored for specialized hardware platforms, such as Field-Programmable Gate Arrays (FPGAs) [20]. These approaches leverage hardware acceleration to perform compression at high speeds, often integrated directly into the data path of stream processing pipelines.

In summary, while several works have investigated compression in different aspects of stream processing, our work is the first to apply adaptive, metric-driven control to the compression of windowed state in stream Aggregates. By building upon the foundation laid by [3] and extending it with dynamic adjustment capabilities, we offer a more flexible, usable, and robust solution for memory-constrained environments. The techniques discussed in related work are largely complementary to our approach and could be integrated in future extensions. For example, by combining our adaptive mechanism with compressed-domain computation or hardware-accelerated compressionto further enhance the efficiency of stream processing systems.

8

Conclusion and Future Work

As data generation moves closer to the source, such as vehicles, sensors, and mobile devices, centralized processing in the cloud is no longer sufficient due to limitations in bandwidth, latency, and privacy. This shift has led to a growing interest in deploying stream processing applications directly on edge devices, where computations can be performed locally before any data is transmitted upstream.

Stream Aggregates are a fundamental building block in such systems, allowing analysts to summarize continuous data streams over time-based windows. These operators maintain internal state to store intermediate results, which can grow significantly over time, especially under high data ingestion rates. In resource-constrained environments like edge devices, managing memory efficiently becomes a critical challenge. One promising approach to reduce memory pressure is to compress the state of infrequently accessed window instances, trading a small amount of CPU overhead for significant gains in memory savings.

Recent work has introduced techniques for memory compression in stream aggregates, where a parameter D controls when a window becomes eligible for compression based on how long it has been inactive. However, these approaches typically rely on a fixed, manually configured D, which requires the analyst to have prior knowledge of data characteristics such as reporting frequency, arrival patterns, and workload dynamics. In practice, this information is often unknown, variable, or difficult to estimate, making manual tuning both challenging and suboptimal.

To address this limitation, we have presented a study that extends existing compression techniques in two key ways. First, we have evaluated the impact of different compression libraries on the performance of a stream aggregate. Our results show that each library offers a different trade-off between compression efficiency and processing overhead. For example, Snappy provides faster compression and decompression, leading to higher throughput and lower latency, but achieves less memory reduction. In contrast, Zstd offers better compression ratios at the cost of increased CPU usage, while JZlib performs worse across most metrics. These findings highlight the importance of selecting an appropriate compression backend depending on the specific constraints of the deployment environment.

Second, and more importantly, we have proposed a dynamic, self-adaptive mechanism that automatically adjusts the compression threshold D based on runtime feedback. Instead of requiring the analyst to set a fixed value, our approach allows

them to define a desired target range for a performance metric, such as the n/c ratio (the proportion of non-compressed windows), and the system continuously monitors this metric and modifies D accordingly. This enables the system to maintain stable and predictable behavior even when workload conditions change, without requiring expert-level knowledge or manual intervention.

Our experimental evaluation demonstrates that this adaptive strategy is effective. When subjected to artificial disruptions, such as sudden resets of D to extreme values, the system is able to recover quickly and stabilize the n/c ratio within the user-defined bounds. Furthermore, the performance overhead of dynamic adjustment is negligible compared to a well-tuned fixed-D configuration, with only minor differences in latency and memory usage.

These results suggest that adaptive compression can significantly improve the usability and robustness of stream processing systems, particularly in edge scenarios where conditions are unpredictable and users may not have deep system expertise. By shifting the focus from low-level parameter tuning to high-level behavioral goals, our approach makes memory optimization more accessible and reliable.

Looking forward, there are several directions in which this work could be extended. One possibility is to support multi-objective adaptation, where the system considers multiple metrics simultaneously, such as memory usage, CPU load, and latency, when adjusting D. This would allow for more sophisticated trade-off management in complex environments. Another direction is to explore the use of predictive models to anticipate changes in workload and proactively adjust D, rather than reacting after changes occur.

Additionally, future research could investigate the integration of lossy compression techniques or methods that allow computations to be performed directly on compressed data, such as those explored in Tercedes [16]. While these approaches may introduce some approximation error, they could offer even greater memory savings, especially for applications where perfect accuracy is not required.

In summary, this thesis contributes to the ongoing effort to make stream processing more efficient and adaptive, particularly in heterogeneous edge-to-cloud environments. By combining empirical evaluation of compression libraries with a user-friendly, feedback-driven control mechanism, we take a step toward smarter, self-tuning systems that can automatically balance performance and resource usage with minimal human intervention.

Bibliography

[1] *Apache Flink*, https://flink.apache.org, Accessed: 2025-02-14.

[2] *RisingWave*, https://risingwave.com/, Accessed: 2025-02-14.

[3] 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.

[4] *Snappy*, https://github.com/xerial/snappy-java, Accessed: 2025-02-14.

[5] *Zstandard*, https://facebook.github.io/zstd/, Accessed: 2025-02-14.

[6] *JZlib*, https://github.com/ymnk/jzlib, Accessed: 2025-02-14.

[7] A. Arasu, M. Cherniack, E. Galvez, *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.

[8] *Liebre SPE*, https://github.com/vincenzo-gulisano/Liebre, Accessed: 2025-02-14.

[9] T. Akidau, R. Bradshaw, C. Chambers, *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.

[10] *Kafka Stream*, https://kafka.apache.org/documentation/streams/, Accessed: 2025-02-14.

[11] *Apache Storm*, http://storm.apache.org, Accessed: 2025-02-14.

[12] 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.

[13] *aircompressor*, https://github.com/airlift/aircompressor, Accessed: 2025-02-14.

[14] *lz4*, https://lz4.org/, Accessed: 2025-02-14.

[15] 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*, ser. ICPE '25, Toronto ON, Canada: Association for Computing Machinery, 2025, pp. 240–252, ISBN: 9798400710735. DOI: 10.1145/3676151.3719369. [Online]. Available: https://doi.org/10.1145/3676151.3719369.

[16] G. Pekhimenko, C. Guo, M. Jeon, P. Huang, and L. Zhou, "Tersecades: Efficient data compression in stream processing," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018, pp. 307–320.

[17] Y. Zhang, F. Zhang, H. Li, S. Zhang, and X. Du, “Compressstreamdb: Fine-grained adaptive stream processing without decompression,” in *2023 IEEE 39th International Conference on Data Engineering (ICDE)*, 2023, pp. 408–422. DOI: 10.1109/ICDE55515.2023.00038.

[18] X. Zeng and S. Zhang, “Parallelizing stream compression for iot applications on asymmetric multicores,” in *2023 IEEE 39th International Conference on Data Engineering (ICDE)*, 2023, pp. 950–964. DOI: 10.1109/ICDE55515.2023.00078.

[19] G. Theodorakis, F. Kounelis, P. Pietzuch, and H. Pirk, “Scabbard: Single-node fault-tolerant stream processing,” *Proceedings of the VLDB Endowment*, vol. 15, no. 2, pp. 361–374, 2021.

[20] P. R. Geethakumari and I. Sourdis, “Stream aggregation with compressed sliding-windows,” *ACM Transactions on Reconfigurable Technology and Systems*, 2023.