An Algorithm for Tunable Memory Compression of Time-Based Windows for Stream Aggregates

Downloaded from: https://research.chalmers.se, 2026-04-04 19:43 UTC

Citation for the original published paper (version of record):

Gulisano, V. (2024). An Algorithm for Tunable Memory Compression of Time-Based Windows for Stream Aggregates. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 14351 LNCS: 18-29. http://dx.doi.org/10.1007/978-3-031-50684-0_2

N.B. When citing this work, cite the original published paper.

research.chalmers.se offers the possibility of retrieving research publications produced at Chalmers University of Technology. It covers all kind of research output: articles, dissertations, conference papers, reports etc. since 2004. research.chalmers.se is administrated and maintained by Chalmers Library

(article starts on next page)

An Algorithm for Tunable Memory Compression of Time-Based Windows for Stream Aggregates

Vincenzo Gulisano(✉)

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

Abstract. Cloud-to-edge device continuums transform raw data into insights through data-intensive processing paradigms such as stream processing and frameworks known as Stream Processing Engines (SPEs). The control of resources in streaming applications within and across such continuums has been a prominent topic in the literature. While several techniques have been proposed to control resources like CPU, limited control exists for other resources such as memory.

Based on this observation, this work proposes an algorithm for streaming aggregation that allows for control of memory usage through lossless compression. The algorithm provides a "knob" to control the amount of state that should be compressed, prioritizing the compression of old over fresh data when performing streaming aggregation. Together with a detailed algorithmic description, this work presents preliminary results from a fully implemented prototype on top of the Liebre SPE, showing the effectiveness of the proposed approach.

Keywords: Stream Processing · Stream Aggregate · Compression

1 Introduction

Cloud-to-edge device continuums support pipelines transforming edge data into cloud-based insights/decisions. These pipelines often rely on data-intensive processing paradigms like stream processing where applications are defined as Directed Acyclic Graphs (DAGs) of operators and run by Stream Processing Engines (SPEs). SPEs allow control of how pipelines are deployed through the distribution and parallelization of the operators of a DAG.

Despite the increasing computational power of the cloud-edge continuum, resources are typically dedicated to critical tasks, with limited room for custom analysis, especially at the edge (e.g., as in modern vehicles, where powerful devices are primarily utilized for critical driving applications [18]).

The ability to control resource utilization of stream processing applications has been a prominent topic in the literature, e.g., with adaptive distribution and elasticity [8,9], shedding [20], and thread scheduling [22,23]. However, current

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024
D. Zeinalipour et al. (Eds.): Euro-Par 2023 Workshops, LNCS 14351, pp. 18–29, 2024.
https://doi.org/10.1007/978-3-031-50684-0_2

solutions offer limited control over memory usage, which is crucial for streaming applications with operators maintaining state over analyzed data to manage their memory footprint. In this context, this work introduces an algorithm for streaming aggregation that allows for control of memory usage through compression. By defining a single additional parameter to existing streaming aggregation parameters, the algorithm provides a "knob" to control the amount of state that should be compressed, prioritizing the compression of old over fresh data.

This work provides a detailed algorithmic description of the proposed solution, along with a fully implemented prototype on top of the Liebre SPE [21]. The results demonstrate the effectiveness of the approach in managing memory usage while maintaining the desired level of aggregation accuracy.

2 Preliminaries

2.1 Stream Processing Basics and Streaming Aggregation

According to DataFlow [2], a *stream* S is an unbounded sequence of *tuples*, each defined by its *type* $\langle \tau, v_1, \ldots, v_n \rangle$, where τ is the timestamp attribute, always included in the type of a tuple, and $v_1, \ldots, v_n$ are application-specific attributes. Streams are homogeneous: every tuple t of the same stream S has the same type.

Stream processing queries (or simply queries) are composed of *ingresses*, *operators*, and *egresses*. Ingresses forward tuples (e.g., events reported by sensors or other applications) to operators, the basic units manipulating tuples. Operators connected in a Directed Acyclic Graph (DAG) process and forward/produce tuples; eventually, tuples are fed to egresses, which deliver results to end-users or other applications. Multiple copies of the same operator can be deployed within the same DAG, each analyzing a portion of a given stream (e.g., tuples sharing the same key in *key-by* parallelism, as explained in the remainder).

As an ingress tuple t corresponds to an event, $t.\tau$ is the *event time* set by the ingress when the event took place. Event time is expressed in time units from a given epoch and progresses in SPE-specific δ increments (e.g., milliseconds [11]).

Operators are distinguished into *stateless* and *stateful*. FlatMap, Filter, and Map are stateless operators that do not maintain a state that evolves according to the tuples they process. Stateful operators produce results from a state, dependent on one or more tuples. This work targets stateful operators defined over delimited groups of tuples called *time-based windows* (or simply windows) which are commonly provided by SPEs [11,21,25]: *Aggregates* over windows.

An Aggregate $A(WA, WS, S_I, f_K, f_A, f_O, f_S)$ is defined by parameters:

Window Advance (WA), Size (WS): the epochs $[\ell\, WA, \ell\, WA + WS)$, with $\ell \in \mathbb{N}$, covered by A. Each epoch is referred to as a window *instance* γ. If $WA < WS$, consecutive *sliding* γs overlap and a tuple can fall into several γs. If $WA = WS$, each tuple falls in exactly one *tumbling* γ.

Stream S_I: the input stream fed to A.

Function f_K: which specifies the subset (possibly empty) of S_I tuples' attributes used to maintain dedicated γs for tuples that share the same *key*. Note that f_K affects the way in which A is parallelized, as discussed next.

Function $f_A(\gamma, t)$ to add to γ the contribution of t.
Function $f_O(\gamma)$ to compute the values of an output t_o from γ.
Function $f_S(\gamma, l)$ to advance γ to its new left boundary l, increasing l by WA.

A γ's left boundary (inclusive) is referred to as $\gamma.l$ (omitting γ if clear from the context). The right (exclusive) boundary is $\gamma.l + WS$. When an output tuple t_o is created from γ, $t_o.\tau$ is set by A to $\gamma.l + WS - \delta$ [6,11,25].

As aforementioned, SPEs parallelize the execution of an operator by deploying multiple copies of such an operator. To achieve this, SPEs let users define operators as *logical*, and later convert them into *physical instances*. Since stateless operators do not maintain a state that evolves based on the tuples they process, the data fed to multiple instances of the same logical operator can be shuffled or fed in a round-robin fashion. For logical stateful operators like A, SPEs rely on key-by routing/partitioning, splitting the data sent to such instances so that tuples sharing the same f_K value are correctly processed by the same instance.

2.2 Correctness Conditions

Users expect SPEs' executions to enforce A's semantics correctly:

Definition 1. *A's execution is correct if any subset of tuples from S_I sharing the same key and falling in the same γ is jointly processed by f_O exactly once and the resulting output tuple is forwarded to A downstream peers.*

For A, Definition 1 implies that all the tuples falling into γ should be added to γ by f_A and jointly processed by f_O exactly once. Correct execution can be achieved by consistently maintaining A's *watermarks* [17]:

Definition 2. *A's watermark W_A^ω at wall-clock time ω is the earliest event time a tuple t_i fed to A can have from ω on (i.e., $t_i.\tau \geq W_A^\omega, \forall t_i$ fed to A from ω on).*

Watermarks are commonly maintained assuming ingresses periodically output special watermark tuples with monotonically increasing timestamps [11,17]. They serve as notifications of how event-time advances from the perspective of ingresses, and operators use them to (1) make progress even in the absence of regular tuples and/or (2) reorder tuples from out-of-timestamp-order streams.

Upon receiving a watermark, A can store the watermark's time and update W_A^ω to the smallest value among those in the set comprised of the latest watermark from each input stream. Upon reception of a watermark that increases W_A^ω, A can output the results of all γs whose right boundary is not greater than W_A^ω (i.e., invoke f_O on any $\gamma | \gamma.l + WS \leq W_A^\omega$) since no more tuples will fall in such γs, and then forward W_A^ω to its downstream peers.

3 Problem Definition

This study aims at defining an Aggregate operator that lets users customize its behavior through its common parameters (see Sect. 2.1) allowing to control which

portion of its γs should be compressed, ranging from none to all, prioritizing the compression of γs updated least recently over those updated more recently.

Since the additional compression and decompression of γs can ease memory consumption but also incurs a computational cost besides the data handling and data analysis ones already defined by an Aggregate, this work assesses the memory/performance trade-offs based on the following metrics:

- Throughput: the number of tuples processed per unit of time,
- Latency: the delay in the production of an output tuple once the tuple triggering the production of such an output is fed to A,
- Number of compressions/decompressions (cumulative) over time,
- Memory usage: the total memory used by A's state, and
- Percentage of compressed/uncompressed γs (over the total number of γs).

This work assumes that each one of the input streams fed to an instance of A is sorted on its timestamp attributes or, alternatively, that it can be sorted by relying on watermarks before being fed to A [15]. If multiple streams are fed to A, then they are first merge-sorted based on their timestamp, for instance, like in [16]. Notice that, with such an assumption in place, the timestamp of each tuple fed to A is in fact an update of the A's watermark, since any later tuple fed to A will have an equal or greater timestamp, according to Definition 2. For A's downstream operators/sinks to also count with sorted streams, the algorithm should also produce A's output tuples in timestamp order [16].

When presenting the proposed algorithm, for ease of exposition, the SPE running the A is assumed to define two main methods: `process(`t`)`, invoked by the SPE to signal A that a new input tuple t can be processed by the latter, and `forward(`t`)`, invoked by A when a new output tuple t can be forwarded to downstream operators/sinks, after $t.\tau$ is set by the SPE according to the A's WA and WS parameters (see Sect. 2.1).

4 Proposed Algorithm and Aggregate Operator

This section overviews the algorithm proposed for an operator A_C that can compress part of the γs that have updated least recently, later decompressing/compressing them based on updates or results production.

Besides the parameters covered in Sect. 2.1, A_C defines an additional parameter D, to express the maximum time distance in event time that can elapse from the update of a γ to its compression. Note such a parameter allows tuning the fraction of γs that are compressed by A_C in a range that goes from $D = 0$, where all γs will be immediately compressed after their update, to $D = \infty$, where no γ will be compressed. Also, note that for $D \neq \infty$, any compressed γ must be decompressed before invoking f_A, f_O, or f_S on it.

Algorithm 1 shows A_C's variables and methods. A_C's variables include WA and WS (L 1), parameter D (L 2), two `Map` objects $uncomp_\gamma$ and $comp_\gamma$ maintaining uncompressed and compressed γs, respectively (L 3), the earliest left boundary e_l of any γ maintained by A_C (L 4), a `TreeMap` linking a given event

Algorithm 1: Aggregate A_C supporting compression, run by the SPE.

```
   Local variables:
 1 WA, WS // window advance and size
 2 D // max event-time diff. from last insertion to trig compression
 3 Map<k,γ> uncomp_γ, comp_γ // uncompressed/compressed γs
 4 e_l // earliest left boundary of any γ kept by A
 5 TreeMap<τ,Set<k>> τ_𝕂 // keys for each latest contrib. timestamp
 6 Map<k,τ> k_τ // latest contribution for key k
   Auxiliary Methods:
 7 getEarliestLeftBound(τ) // get γ's earliest left bound for τ
 8 getOrCreate(k, τ) // get (decompress if needed) or create γ
 9 compress(γ),decompress(γ) // compress/decompress γ
10 Method process(t) // process tuple t
11   while e_l + WS < t.τ do // Output and advance γs
12     for k ∈ uncomp_γ do // Output and advance uncompressed γs
13       forward(f_O(uncomp_γ[k]))
14       uncomp_γ[k] ← f_S(uncomp_γ[k], e_l + WA)
15       if |uncomp_γ[k]| == 0 then // remove if γ is empty
16         uncomp_γ.remove(k)
17     for k ∈ comp_γ do // Output and advance compressed γs
18       γ ←decompress(comp_γ[k]) // decompress γ
19       forward(f_O(γ))
20       γ ← f_S(γ, e_l + WA)
21       if |γ| == 0 then // remove if γ is empty
22         comp_γ.remove(k))
23       else // compress γ again
24         comp_γ[k] ← compress(γ)
25     e_l ← e_l + WA // advance earliest left boundary of γs kept by A
26   k ← f_K(t) // get t's key
27   l ← getEarliestLeftBound(t.τ) // get γ's earliest left bound for t
28   γ ← getOrCreate(k,l) // get (decompress if needed) or create γ
29   γ ← f_A(γ, t) // add the tuple
30   if k ∈ k_τ then // update k_τ and τ_𝕂 based on t
31     τ_𝕂[k_τ[k]].remove(k)
32   k_τ[k] ← t.τ
33   τ_𝕂[t.τ].add(k)
34   for τ ∈ τ_𝕂 do // compress γs not updated in the last D time units
35     if t.τ − τ ≥ D then
36       for k' ∈ τ_𝕂[τ] do
37         comp_γ[k'] ← compress(uncomp_γ[k'])
38         uncomp_γ.remove(k')
39         k_τ.remove(k')
40       τ_𝕂.remove(τ)
41     else
42       break
```

time τ with all keys for which τ represents the last event time at which the corresponding γ has been updated (L 5), and a `Map` storing, for each key, the latest event time at which the corresponding γ has been updated (L 6).

Notation: $M[k]$ refers to the entry k of `Map` or `TreeMap` M, M.`remove(`k`)` indicates key k is being removed from M, S.`add(`k`)` indicates the k is being added to set S, "`if (`$k \in M$`) {}`" checks if k is a key maintained by M, $|\gamma|$ refers to the number of tuples in γ. The keys of a `TreeMap` M are traversed in increasing order when running "`for (`$k \in M$`) {}`".

In Algorithm 1, the auxiliary method `getEarliestLeftBound(`τ`)` computes the left bound of the earliest γ to which a tuple with timestamp τ contributes to (L 7). Method `getOrCreate(`k, τ`)` retrieves the γ for key k and timestamp τ. If γ exists but is compressed, the method decompresses γ before returning it, while if γ does not exist, the method creates it (L 8). Methods `compress(`γ`)`/ `decompress(`γ`)` compress/decompress γ, respectively (L 9).

Upon reception of a new input tuple (method `process(`t`)`, L 10), A_C produces all the output tuples that can be produced based on $t.\tau$. As discussed in Sect. 3, $t.\tau$ represents in this case A_C's latest watermark (see Sect. 2.2). A_C keeps producing, in timestamp order, the results of the γs it maintains as long as their left boundary indicates such γs are expired (i.e., as long as $e_l + WS < t.\tau$, L 11), increasing e_l by WA at each iteration (L 25), and thus meeting the requirement of producing output tuples in timestamp order (see Sect. 3). Within each iteration, A_C begins by producing the results for uncompressed γs. For each γ, the result is retrieved and forwarded by A_C. Subsequently, γ is advanced and either maintained or discarded depending on whether it is empty or not once advanced (L 12-16). Results for compressed γs are produced similarly, with additional calls to methods `decompress`, before producing the result of a γ and advancing it, and `compress`, before storing back in $comp_\gamma$ a non-empty γ (L 17-24).

Once the output tuples (if any) that could be produced based on the incoming t are forwarded, A_C retrieves $k = f_K(t)$ (L 26), the left boundary of the earliest γ to which t contributes (L 27), the corresponding γ, decompressing or creating it if necessary (L 28), and proceeds adding t to γ (L 29). Then, it proceeds to update the information about the latest event time at which k has been updated. If k was previously updated it removes K from the set of keys for the corresponding event time update. Then, it proceeds to update $k_\tau[k]$ and $\tau_{\mathbb{K}}$ (L 30-33). Finally, A_C traverses all the keys of the uncompressed γs it maintains based on the latest event time at which each key updated its corresponding γ. The γs that refer to keys whose last update is far away from $t.\tau$ more than D time units are compressed and moved from $uncomp_\gamma$ to $comp_\gamma$, updating $k_\tau[k]$ and $\tau_{\mathbb{K}}$ accordingly. Note the traversal stops as soon as the distance between $t.\tau$ and the latest update of a given set of is less than D (L 34-42).

5 Evaluation

The evaluation begins by discussing hardware/software, data, and how experiments are conducted for A_C. Then, it discusses how the performance metrics introduced in Sect. 3 behave for various setups of the experiments' parameters.

Hardware/Software. Experiments run on an Intel Xeon E5-2637 v4 @ 3.50 GHz (4 cores, 8 threads) server with 64 GB of RAM with Ubuntu 18.04. The server is representative of a device that could be deployed at the edge end of the cloud-edge continuum in connection to the use-case presented next. Algorithm 1 has been implemented using the Liebre SPE [21] and the Snappy [1] compressor.

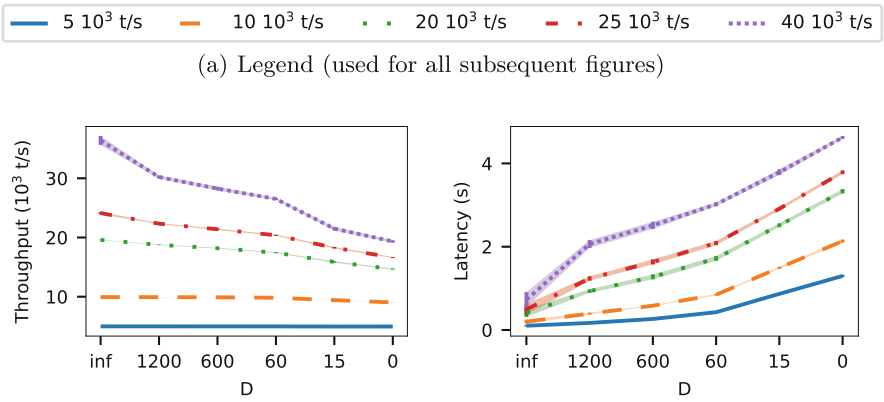
Data. For S_I (see Sect. 2.1), the data comes from the Linear Road benchmark [5], a popular benchmark in stream processing that simulates a real-time traffic monitoring system. Its data models individual vehicles with a given number of highways, generating a stream of reports of vehicles' position and speed, with new reports being generated every second. Consecutive reports from the same vehicle are 30 s apart (in event time). Each vehicle starts/ends producing reports within a given event time interval, lasting a few to tens of minutes. The simulated traffic covers 3 h (in event time) and results in $\sim$ 44 million tuples.

Experimental Setup. A_C's f_O (see Sect. 2.1) counts the number of stops each vehicle performs during its journey, where a stop is a sequence of at least one position report with zero speed in-between reports with non-zero speed (or happening at the very beginning/end of the journey), over a window with WA and WS set to 1 min and 3 h, respectively. A_C's f_A and f_S store and remove tuples in a γ's internal state, respectively. The experiments study A_C's performance for varying injection rates – $5{*}10^3$, $10{*}10^3$, $20{*}10^3$, $25{*}10^3$, and $40{*}10^3$ t/s – up to the rate sustainable for an A_C that does not compress its γs, and D values (see Sect. 4) – ∞, 20 m, 10 m, 1 m, 15 s and 0 s. Note that, for a fair comparison, the original Liebre Aggregate is used when $D = \infty$ (i.e., when no γ is to be compressed). Each experiment lasts 10 min. The data collected during the first (warm-up) and last (cool-down) minutes is excluded from the results. For each experiment, given the performance metrics introduced in Sect. 3, throughput and latency metrics refer to the average value observed during the experiment, while ratio, compression, decompression, and memory metrics refer to the value at the end of the experiment. All results are averaged over 10 repetitions. Shaded areas in the plots represent the 99% confidence interval. Since accurate runtime memory estimation in Java is costly, runtime measurements are based on precomputed measurements, obtained using the Jamm library[1], of tuples' sizes and per-tuple overheads (e.g., for maintaining a tuple in a `LinkedList`), later used as a multiplicative factor of the total number of tuples maintained by A_C.

Results. Figure 1(a) presents the legend for the different injection rates used in all the subsequent plots, while Figs. 1(b) and 1(c) present the throughput and latency metrics, respectively, for the various injection rates and D values. As shown, the throughput degrades for decreasing D values, and, the higher the injection rate, the more severe the degradation. Similar trends are observed for

[1] https://github.com/jbellis/jamm.

the latency, with an increasing overhead as D increases and higher overheads observed for higher injection rates.

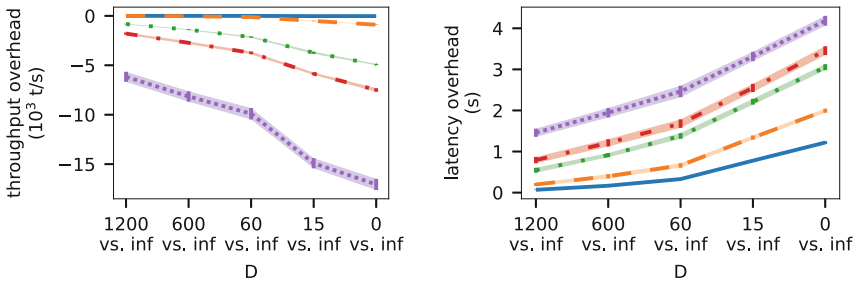


(a) Legend (used for all subsequent figures)

(b) Throughput for various D values

(c) Latency for various D values

Fig. 1. Legend for all figures(a), and throughput(b)/latency(c) performance figures.



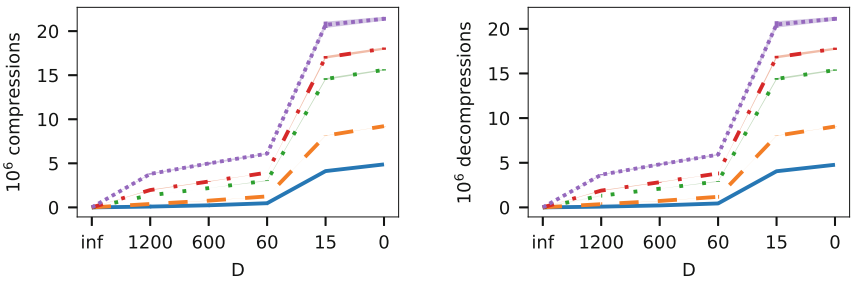
(a) Throughput overhead for various D values, compared with $D = \infty$

(b) Latency overhead for various D values, compared with $D = \infty$

Fig. 2. Throughput (b) and latency (c) overheads.

The actual overheads (in t/s for throughput and s for latency) observed when comparing the performance of an A_C that does not rely on compression (i.e., when $D = \infty$) and one that does (i.e., when $D \neq \infty$) are shown in Fig. 2(a) and Fig. 2(b), respectively. As shown, the degradation can grow to e.g., -40% throughput and $+9\times$ latency for a low D value (15 s) and high injection rate ($40{*}10^3$ t/s). To contextualize the overheads with respect to the gains in memory, though, one can observe from Fig. 3(a) and Fig. 3(b) that the values 15 s and 0 s

for D are the values at which the total number of compressions/decompressions spikes. If this is expected when $D = 0$ s, since each γ is immediately compressed once a tuple is added to it, it is also expected for $D = 15$ based on the data being processed: since 15 s is less than the time interleaving two reports from the same tuple, such a value implies each γ, once a tuple is added to it, is compressed before any subsequent tuple from the same vehicle is again added to it.



(a) Overall number of compressions (b) Overall number of decompressions

Fig. 3. Overall number of compression (a) and decompression (b) actions

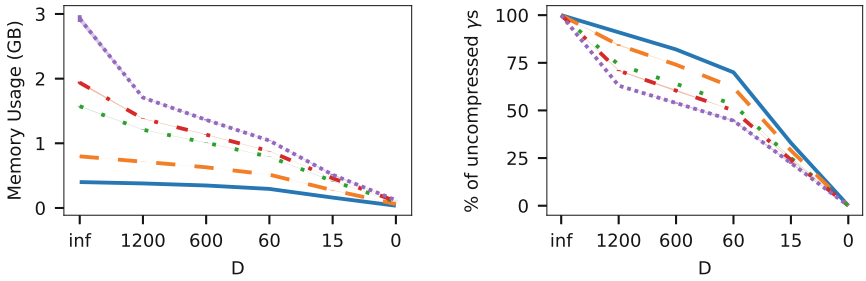
For a larger D value such as 60s, one can nonetheless observe that the benefits from the memory perspective are visible even for a less aggressive compression threshold, as shown in Fig. 4(a) and Fig. 4(b). For a medium injection rate, one can observe a reduction of approximately 2/3 in the overall memory (from 3 to 1 GB), with approximately 40% of the overall γs being compressed, with a throughput degradation of about 15%.

In general, the preliminary results indicate several configurations of the D parameter, for several of the studied rates, have overheads that lead to performance figures in the same order of magnitude as those without compression, while they allow reducing memory usage by at least one-third.

6 Related Work

Managing access to computational resources in streaming applications is a widely researched topic, as evidenced by the literature [3,7,9,19,22,23]. While several proposed techniques/tools focus on CPU resource control [9,23] memory management (in general) and compression (in connection to this work) have received less attention, unlike in databases where they have been discussed in major depth [26]. Examples in stream processing are e.g., discussed in [23], including the provisioning and scheduling of threads to queries and their operators.

Focusing on the existing solutions for streaming applications, memory control has been discussed in [12–14] but with a focus on specific hardware like FPGAs. The compression technique proposed in [14] is also for stream aggregation but for



(a) Total used memory (b) Ratio of uncompressed γs

Fig. 4. Overall memory usage (a) and ratio of uncompressed/compressed γs (b)

tuple-based windows, which differ from the windows considered in this work. The tuple-based windows are chosen for FPGAs because they allow the exact size of each window instance to be known in advance, enabling allocation and control of the required memory for that window instance. Expanding the discussion to generic hardware, other complementary approaches have been proposed in [10, 18]. However, these techniques differ from this work in that they apply lossy (with a controllable error bound) instead of lossless compression and apply compression to streams across operators rather than within an Aggregate operator. Similarly, [24] proposes a compression scheme for tuples maintained outside individual operators' states. While [24] evaluates compression for stream aggregation, the proposed solution only supports tumbling windows with $WA{=}WS$, unlike this work. Additional work that relates to stream aggregation is discussed in [4]. However, this contribution focuses on a data structure (the Compressed Buffered Tree) that can be leveraged by any operator rather than on the internal state of stream Aggregates. Additionally, the data structure supports continuous stream aggregation on a per-key basis but does not discuss window semantics.

7 Conclusions and Future Work

This work introduced a novel algorithm for tunable memory compression in stream aggregation. By defining a single additional parameter besides the common parameters defined for streaming aggregation over time-based windows, the algorithm allows controlling the amount of compressed window instances, prioritizing the compression of those not recently updated.

Initial results show the proposed algorithm can be beneficial and compress memory by at least one-third, with overheads in throughput/latency performance figures in the same order of magnitude as those without compression. This initial study can be expanded in several research directions: extended empirical studies, with more use-cases, setups (e.g., parallel/distributed executions) and heterogenous hardware, comparison with state-of-the-art baselines, and amounts of available memory, studying also the behavior of an Aggregate when memory

is exhausted or in high contention with e.g., a garbage collector; generalization of the proposed technique to accommodate different compression techniques, possibly extending to lossy compression too, and study of different compression levels for a given technique, and joint use of the proposed technique with AI-based agents adjusting parameter D based on e.g., Quality-of-Service requirements.

Acknowledgements. This work is 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, by Chalmers Un. AoA frameworks Energy and Production, proj. INDEED, and WP "Scalability, Big Data and AI", respectively, and by the Swedish Government Agency for Innovation Systems VINNOVA, proj. "Automotive Stream Processing and Distributed Analytics (AutoSPADA)" (DNR 2019-05884) in the funding program FFI: Strategic Vehicle Research and Innovation.

References

1. Snappy, a fast compressor/decompressor. https://github.com/google/snappy
2. Akidau, T., 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 **8**(12), 1792–1803 (2015)
3. Aldinucci, M., Danelutto, M., Kilpatrick, P., Torquati, M.: Fastflow: high-level and efficient streaming on multicore, chap. 13, pp. 261–280. Wiley (2017)
4. Amur, H., et al.: Memory-efficient groupby-aggregate using compressed buffer trees. In: Proceedings of the 4th Annual Symposium on Cloud Computing, pp. 1–16 (2013)
5. Arasu, A., et al.: Linear road: a stream data management benchmark. In: Proceedings of the Thirtieth International Conference on Very large data bases-Volume 30, pp. 480–491. VLDB Endowment (2004)
6. Apache beam. https://beam.apache.org/. Accessed 12 Nov 2020
7. Cardellini, V., Grassi, V., Presti, F.L., Nardelli, M.: On QoS-aware scheduling of data stream applications over fog computing infrastructures. In: 2015 IEEE Symposium on Computers and Communication (ISCC), pp. 271–276. IEEE (2015)
8. Cardellini, V., Nardelli, M., Luzi, D.: Elastic stateful stream processing in storm. In: 2016 International Conference on High Performance Computing Simulation (HPCS), pp. 583–590. IEEE (2016)
9. De Matteis, T., Mencagli, G.: Keep calm and react with foresight: strategies for low-latency and energy-efficient elastic data stream processing. ACM SIGPLAN Notices **51**(8), 1–12 (2016)
10. Duvignau, R., Gulisano, V., Papatriantafilou, M., Savic, V.: Streaming piecewise linear approximation for efficient data management in edge computing. In: Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing (2019)
11. Apache flink. https://flink.apache.org. Accessed 27 Jan 2023
12. Geethakumari, P.R., Gulisano, V., Svensson, B.J., Trancoso, P., Sourdis, I.: Single window stream aggregation using reconfigurable hardware. In: 2017 International Conference on Field Programmable Technology (ICFPT). IEEE (2017)
13. Geethakumari, P.R., Gulisano, V., Trancoso, P., Sourdis, I.: Time-SWAD: a dataflow engine for time-based single window stream aggregation. In: 2019 International Conference on Field-Programmable Technology (ICFPT), pp. 72–80. IEEE (2019)

14. Geethakumari, P.R., Sourdis, I.: Stream aggregation with compressed sliding-windows. ACM Trans. Reconfigurable Technol. Syst. **16**, 1–28 (2023)
15. Gulisano, V., Nikolakopoulos, Y., Cederman, D., Papatriantafilou, M., Tsigas, P.: Efficient data streaming multiway aggregation through concurrent algorithmic designs and new abstract data types. ACM Trans. Parallel Comput. **4**(2), 11:1–11:28 (2017). https://doi.org/10.1145/3131272
16. Gulisano, V., Nikolakopoulos, Y., Papatriantafilou, M., Tsigas, P.: ScaleJoin: a deterministic, disjoint-parallel and skew-resilient stream join. IEEE Trans. Big Data **7**(2), 299–312 (2021). https://doi.org/10.1109/TBDATA.2016.2624274
17. Gulisano, V., Palyvos-Giannas, D., Havers, B., Papatriantafilou, M.: 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)
18. Havers, B., Duvignau, R., Najdataei, H., Gulisano, V., Koppisetty, A.C., Papatriantafilou, M.: Driven: a framework for efficient data retrieval and clustering in vehicular networks. In: 2019 IEEE 35 th International Conference on Data Engineering (ICDE), pp. 1850–1861. IEEE (2019)
19. Hirzel, M., Soulé, R., Schneider, S., Gedik, B., Grimm, R.: A catalog of stream processing optimizations. ACM Comput. Surv. (CSUR) **46**(4), 1–34 (2014)
20. Kalyvianaki, E., Fiscato, M., Salonidis, T., Pietzuch, P.: Themis: fairness in federated stream processing under overload. In: Proceedings of the 2016 International Conference on Management of Data, pp. 541–553 (2016)
21. Liebre SPE. https://github.com/vincenzo-gulisano/liebre. Accessed 27 June 2022
22. Palyvos-Giannas, D., Gulisano, V., Papatriantafilou, M.: Haren: a framework for ad-hoc thread scheduling policies for data streaming applications. In: Proceedings of the 13th ACM International Conference on Distributed and Event-based Systems, pp. 19–30 (2019)
23. Palyvos-Giannas, D., Mencagli, G., Papatriantafilou, M., Gulisano, V.: Lachesis: a middleware for customizing OS scheduling of stream processing queries. In: Proceedings of the 22nd International Middleware Conference, pp. 365–378 (2021)
24. Pekhimenko, G., Guo, C., Jeon, M., Huang, P., Zhou, L.: TerseCades: efficient data compression in stream processing. In: 2018 USENIX Annual Technical Conference (USENIX ATC 18), pp. 307–320 (2018)
25. Apache storm. https://storm.apache.org. Accessed 1 Mar 2019
26. Zhang, H., Chen, G., Ooi, B.C., Tan, K.L., Zhang, M.: In-memory big data management and processing: a survey. IEEE Trans. Knowl. Data Eng. **27**(7), 1920–1948 (2015)