STRETCH: Virtual Shared-Nothing Parallelism for Scalable and Elastic Stream Processing

Vincenzo Gulisano, Hannaneh Najdataei, Yiannis Nikolakopoulos, Alessandro V. Papadopoulos, Marina Papatriantafilou, and Philippas Tsigas

Abstract—Stream processing applications extract value from raw data through Directed Acyclic Graphs of data analysis tasks. Shared-nothing (SN) parallelism is the de-facto standard to scale stream processing applications. Given an application, SN parallelism instantiates several copies of each analysis task, making each instance responsible for a dedicated portion of the overall analysis, and relies on dedicated queues to exchange data among connected instances. On the one hand, SN parallelism can scale the execution of applications both up and out since threads can run task instances within and across processes/nodes. On the other hand, its lack of sharing can cause unnecessary overheads and hinder the scaling up when threads operate on data that could be jointly accessed in shared memory. This trade-off motivated us in studying a way for stream processing applications to leverage shared memory and boost the scale up (before the scale out) while adhering to the widely-adopted and SN-based APIs for stream processing applications.
We introduce *STRETCH*, a framework that maximizes the scale up and offers instantaneous elastic reconfigurations (without state transfer) for stream processing applications. We propose the concept of Virtual Shared-Nothing (VSN) parallelism and elasticity and provide formal definitions and correctness proofs for the semantics of the analysis tasks supported by *STRETCH*, showing they extend the ones found in common Stream Processing Engines. We also provide a fully implemented prototype and show that *STRETCH*'s performance exceeds that of state-of-the-art frameworks such as Apache Flink and offers, to the best of our knowledge, unprecedented ultra-fast reconfigurations, taking less than 40 ms even when provisioning tens of new task instances.

Index Terms—Stream Processing, Shared-Nothing Parallelism, Shared-Memory, Elasticity, Scalability.

1 Introduction

Stream processing applications process data (*tuples*) coming from *unbounded streams* (e.g., tweets or trade records). Such applications are run by Stream Processing Engines (SPEs) such as Apache Flink [2] or Storm [3]. SPEs provide users with semantically-rich operators composable into Directed Acyclic Graphs (DAGs) and automate the process of deploying and executing efficiently user-defined DAGs.

SPEs' de-facto standard to parallelize the execution of stream processing applications builds on *Shared-Nothing* (SN) key-by parallelism. Simply put, the idea is to create multiple instances of each operator, each with a dedicated state and queues to exchange and route each tuple to exactly one of such instances. SN parallelism is also used in elasticity protocols, which aim at adjusting the parallelism degree of the DAGs run by SPEs, avoiding the overheads of over- or under-provisioned applications [4].

Trade-offs of shared-nothing parallelism.

While able to scale the execution of a DAG both up and out SN parallelism can incur unnecessary overheads. The first type of overhead is caused by operators' dedicated input/output queues. When the semantics of a parallel operator O require to route a tuple to multiple instances of O, this leads to data duplication. Consider an application, which we use as a running example, in which O computes the longest tweet on a per-hour, per-hashtag basis. Its analysis can be parallelized by having each parallel instance of O responsible for one observed hashtag. Tweets carrying multiple hashtags, though, might need to be shared with multiple instances. This overhead is exacerbated by the fact that SPEs might leave to users the task of correctly customizing the routing of tuples. The second type of overhead is caused by the operators' dedicated internal state. Since instances' states are not shared, state transfer is needed in elastic reconfigurations to adjust the workload distribution and/or parallelism degree of an operator [4]. This overhead is exacerbated when SPEs request users to implement serialization/deserialization methods for custom states [5].

While these overheads are unavoidable for operator instances running distributedly, they are not for instances that share memory within the same process, and could thus share tuples and states too. Following the principle that applications should be properly scaled up before being scaled out [6], we thus pose the following question: *How can we seamlessly leverage shared memory to boost parallel/elastic executions of common SPEs' operators, while avoiding data duplication and state transfer overheads?*

Contributions

We formally show that it is possible to define parallel and elastic SPE operators that, by *virtualizing* the common Application Programming Interfaces (APIs) based on SN paral-

- *A preliminary version of this paper appeared at the ACM International Conference on Distributed and Event-based Systems (DEBS'19) [1].*
- *V. Gulisano, H. Najdataei, M. Papatriantafilou and P. Tsigas are with Chalmers University of Technology, Göteborg; Y. Nikolakopoulos is with ZeroPoint Technologies and contributed to this work while he was with Chalmers University of Technology; A. V. Papadopoulos is with Mälardalen University, Västerås. E-mail: {vincenzo.gulisano,hannajd,ptrianta,tsigas}@chalmers.se; yiannis@zptcorp.com; alessandro.papadopoulos@mdh.se;*

lelism, can leverage shared memory to first scale streaming applications up while allowing to rely on SN parallelism to later scale them out. We thus introduce *STRETCH* and the concept of *Virtual Shared-Nothing* (VSN) parallelism/elasticity in stream processing. These are our contributions:

- we prove the potential need for data duplication in SN parallelism and propose a unified generalized model for SN parallelism encapsulating common SPEs' operators,
- we prove that VSN parallelism can correctly enforce the semantics of our generalized model while circumventing the overheads of data duplication and state transfer,
- we provide a fully implemented prototype, which builds on state-of-the-art data structures such as ScaleGate [7] and our extended *Elastic ScaleGate* implementation, as well as a thorough evaluation with several use-cases and both synthetic and real data.

Outline: § 2 covers preliminaries, § 3 formalizes our problem, § 4 discusses the data duplication overhead and introduces our generalized model, § 5-§ 7 cover *STRETCH*'s model and implementation, § 8 evaluates *STRETCH*, § 9 discusses related work, and § 10 wraps up our work. A table of abbreviations/symbols is found in Appendix A.

2 Preliminaries

2.1 Stream processing basics

In accordance with the DataFlow model [8], *streams* are unbounded sequences of *tuples*. Tuples have two *attributes*: the metadata and the payload φ. The metadata carries the timestamp τ and possibly further *sub-attributes*. We write $t.\tau$ to refer to the τ sub-attribute of tuple t's metadata. We reference φ's ℓ-th sub-attribute as $t.\varphi[\ell]$. A tuple's combined notation is $\langle \tau, \ldots, [\varphi[1], \varphi[2], \ldots] \rangle$.

Stream processing queries (or simply queries) are composed of *ingresses*, *operators*, and *egresses*. Ingresses forward *ingress tuples* (e.g., events reported by sensors or other applications). Each *ingress stream* can be fed to one or more operators, the basic units manipulating tuples. Operators, connected in a DAG, process input tuples and produce output tuples; eventually, *egress tuples* are fed to egresses, which deliver results to end-users or other applications.

As ingress tuples correspond to events, τ is the *event time* set by the ingress to when the event took place. Operators set τ of each output tuple according to their semantics, while φ is set by user-defined functions. Event time is expressed in time units from a given epoch, and progresses in SPE-specific discrete δ increments (e.g., milliseconds [2]).

The operators of a query are either *stateless* or *stateful*. Stateless operators process each tuple individually. The Map/Flatmap (M) operator, for instance, transforms each input tuple t_{in} into one or more output tuples t_{out} by setting $t_{out}.\tau$ to $t_{in}.\tau$ (we write $t_{out}.\tau \leftarrow t_{in}.\tau$) and using a user-defined function to create $t_{out}.\varphi$ from $t_{in}.\varphi$. Stateful operators run their analysis on delimited groups of tuples called *windows*, as explained next.

Stateful analysis over time-based windows

We focus on common stateful operators running their analysis over *time-based windows*, namely *Aggregates* and *Joins*. For conciseness, for operator O we use the notation

$$O(WA, WS, I, f_{SK}, WT, S, f_1, f_2, \ldots)$$

to refer to the parameters that such stateful operators share.

Parameters Window Advance (*WA*) and Size (*WS*) define the advance/size of O's windows, respectively, which cover periods $[\ell WA, \ell WA + WS)$, with $\ell \in \mathbb{Z}$. Consecutive periods overlap if $WA < WS$; the window is then called *sliding* and a tuple can fall into several window *instances*.

Parameter Inputs (*I*) is the number of O's input streams, one for each of O's upstream peers.

Single Key-by function f_{SK} extracts exactly one key from a tuple t, usually returning a subset of $t.\varphi$ [8]. O maintains distinct window instances for each of its I input streams and for groups of tuples that share the same *key*.

Parameter Window Type (*WT*) defines how window instances are internally maintained by O. If WT = single, a single instance is maintained per key and updated based both on tuples entering and leaving it. If WT = multi, multiple overlapping window instances are maintained per key and updated according to the incoming tuples. Hence, new window instances are continuously created while old ones are eventually discarded. As discussed in [9], WT = single is preferable when $WA \ll WS$. Figure 1 shows how two operators (that only differ in *WT*) maintain their w instances (each w contains the tuples falling in it). As shown, for each key O maintains a list of sets, each with I windows (one single set if WT = single, or one or more sets if WT = multi).

Parameter Schema (*S*) defines O's output tuples.

Functions $f_1, f_2, \ldots$ are operator-specific functions to update O's state and produce output tuples.

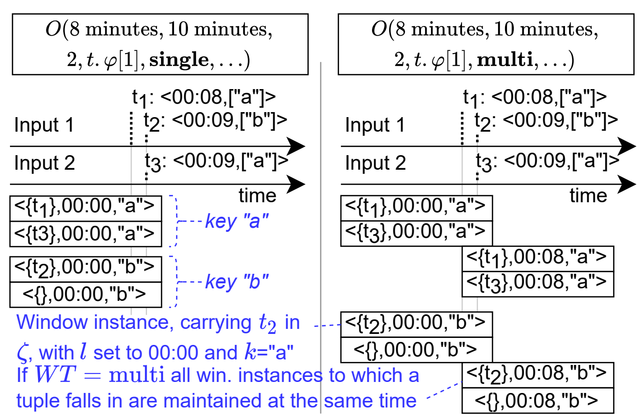


Fig. 1: Sample w instances maintained by two O operators with WT = single (left) and WT = multi (right).

In the following, we use $\langle \zeta, l, k \rangle$ for the combined notation of a window instance w, where ζ is w's internal state (e.g., the tuples falling in w), l is the event time of w's left boundary (inclusive), and k is w's key. The right boundary of w (exclusive) is computed as $w.l + WS$. As common in related works [2], [3], [10], when an output tuple t_{out} is created in connection to a window instance w, $t_{out}.\tau$ is set to w's right boundary. Since w's right boundary is exclusive:

Observation 1. *Any output tuple t_{out} produced from a window instance to which t_{in} falls in is such that $t_{out}.\tau > t_{in}.\tau$.*

The Aggregate $A(WA, WS, 1, f_{SK}, WT, S, f_A, f_R)$ defines f_A to aggregate the tuples falling in one window instance w into the φ attribute of the output tuple created for w, and f_R

to incrementally update $w.\zeta$. As we show in § 4, A can produce an output tuple both incrementally, updating its state for each new tuple it receives, or only upon the expiration of a window instance. The Join $J(WA, WS, 2, f_{SK}, WT, S, f_J)$ defines f_J to match pairs of tuples (one from each stream) that fall to window instances sharing the same boundaries and associated with the same key. Each matched pair can result in up to one output tuple.

2.2 Shared-nothing parallelism (model and notation)

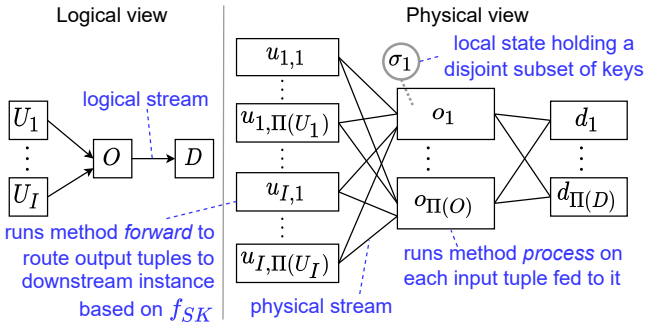


Fig. 2: Logical/physical views of O with SN parallelism.

SPEs let users define operators such as A/J as *logical*, and later convert them into *physical instances* (Figure 2). We use U_i and D to refer to a generic upstream ingress/operator and downstream operator/egress, respectively. $\Pi(X)$ denotes the parallelism degree of the operator, ingress, or egress X. Each logical stream connecting a pair of logical operators (or an ingress/egress) is converted into one or more physical streams. Each instance o_j of operator O is assigned a subset of the keys observed in their input streams (and thus windows), which it maintains in its local state σ_j.

With SN parallelism, each instance o_j has a dedicated queue for the tuples of each of its physical input or output streams. Moreover, o_j owns its dedicated σ_j, which is not concurrently accessed by other operator instances. From an implementation perspective, all the operations carried out by o_j are encapsulated in two methods: `process` and `forward`. Method `process` encapsulates o_j's analysis, and runs every time a tuple forwarded by an $u_{i,j}$ instance is available for processing, being $u_{i,j}$ the j-th instance of U_i. Method `forward` encapsulates the routing of tuples to downstream instances and runs every time one or more tuples produced by `process` should be sent to a D instance. To route tuples, method `forward` has access to a *mapping function* f_μ that maps output tuples' keys to D instances according to D's semantics. We say o_j is responsible for key k if, given the f_μ used by U_i's method `forward`, $f_\mu(k) = j$.

2.3 Correctness conditions

When deploying and running operators, users expect SPEs to enforce operators' semantics correctly. For operator O, correct execution within an SPE can be defined as follows.

Definition 1. O's instances execution is correct if, according to O's specifications, any subset of O's input tuples that could jointly contribute to an output tuple is indeed processed together and results in such an output tuple (if any).

For an Aggregate A, Definition 1 implies that all the input tuples falling into one specific window instance w should be jointly processed by f_A and/or used by f_R to update $w.\zeta$. For a Join J, it implies that any pair of tuples (one from the *left* stream L and one from the *right* stream R) $\langle t_L, t_R\rangle$ such that $t_L \in w_L, t_R \in w_R, w_L.l = w_R.l$, and $f_{SK}(t_L) = f_{SK}(t_R)$ is processed by f_J.

As discussed in [11] the correct execution for an instance o_j requires to consistently maintain its *watermark* $W^\omega_{o_j}$:

Definition 2. The watermark $W^\omega_{o_j}$ of operator instance o_j at a point in wall-clock time[1] ω is the earliest event time a tuple t^ℓ to be processed by o_j can have from time ω on (i.e., $t^\ell.\tau \geq W^\omega_{o_j}, \forall t^\ell$ processed from ω on).

We say a window instance is *expired* if its right boundary falls before the operator instance's watermark. We say a tuple is expired if it only contributes to expired window instances. Based on $W^\omega_{a_j}$, an instance a_i of A can safely invoke f_A for any w such that $w.l + WS \leq W^\omega_{a_j}$ since no more tuples that fall in w will be received or processed by f_R, and it is thus safe to shift (if WT = single) or discard (if WT = multi) w (cf. § 2.1). Appendix B builds on Figure 1 to provide an example visualizing expired and non-expired windows. Based on $W^\omega_{j_i}$, an instance j_i of J can safely shift/delete w if $w.l + WS \leq W^\omega_{j_i}$, since j_i will no longer invoke f_J on w's expired tuples. In related work, $W^\omega_{o_j}$ is updated based on one of the following ways.

Implicit watermarks

The first way to update watermarks assumes that each o_j's physical input stream is timestamp-sorted [4]. The tuples of such physical streams are merge-sorted in timestamp-order and fed to o_j once *ready*, as defined next based on [7]:

Definition 3. t^ℓ_i, the ℓ-th tuple from timestamp-sorted stream i, is *ready* to be processed if $t^\ell_i.\tau \leq \min_i\{\max_m(t^m_i.\tau)\}$, the minimum among the latest m-th tuple timestamps from each timestamp-sorted stream i.

In such a case, $W^\omega_{o_j}$ can be safely updated by o_j to $t.\tau$ for each incoming ready tuple t fed to o_j.

Explicit watermarks

The second way to update watermarks [2], [11] assumes ingresses/operators periodically propagate watermarks through the DAG as special tuples. This allows handling both out-of-timestamp-order streams as well as timestamp-sorted streams whose rate might drop to zero during periods of time (in the latter case this could affect the sorting performed when relying on implicit watermarks). Upon receiving a watermark, o_j stores the watermark's time, updates $W^\omega_{o_j}$ to the minimum of the latest watermarks received from each input stream, and propagates $W^\omega_{o_j}$.

Note that, while both options support correct execution, implicit watermarks also ensure streams are fed in total order to o_j. Hence, o_j can seamlessly support timestamp-order-sensitive analysis. Explicit watermarks are only safe for timestamp-order-insensitive analysis or require o_j to sort its input tuples before processing them [11], [12].

1. From here on, we only differentiate wall-clock time (or simply time) from event time if such distinction is not clear from the context.

2.4 The ScaleGate object

ScaleGate [13] (SG) is a shared data object, that in this work is extended and used to support *STRETCH*'s algorithmic implementation. It allows several *sources* to concurrently and efficiently merge timestamp-sorted streams into a timestamp-sorted stream of ready tuples (cf. Definition 3). Also, it allows several *readers* to consume all the ready tuples of the latter stream. Its lock-free, linearizable algorithmic implementation supports efficient deterministic processing [14], [15]. A fixed set of sources/readers can interact with an SG object using the API methods:

- `addTuple(tuple,i)`: which merges a `tuple` from the i-th source in the timestamp-sorted stream of ready tuples.
- `getNextReadyTuple(i)`: which provides to the i-th reader the next earliest ready tuple that has not been yet consumed by it. Note that each tuple, once ready, will be returned to all readers invoking the method.

To ease notation, we refer to methods `addTuple` and `getNextReadyTuple` as `add` and `get` in the remainder.

2.5 Elasticity

The computational cost of a stream processing application varies over time [4]. Hence, an execution in which the parallelism degree of operators is fixed can lead to imbalances in the work of such operators. When the overall work of an operator is unbalanced but can still be carried out by its instances, a *load balancing* reconfiguration is needed to change the work distribution (i.e., the mapping of each key and the instance responsible for it). If new instances are to be *provisioned*, the new work distribution re-assigns some keys to the newly allocated instances. Since over-provisioned systems can lead to high latency [15] and unnecessary costs [4], existing instances should also be *decommissioned* when fewer instances suffice for the overall workload. We use the term *reconfiguration* to refer to any of these actions (note provisioning and decommissioning imply load balancing).

3 Problem Definition and Approach

This work focuses on intra-process parallel and elastic execution of stateful stream processing analysis for time-based sliding windows (or simply windows), thus assuming $WA < WS$. For any given stateful operator O, we do not make any assumption on the frequency, periodicity, or timings with which tuples are fed to it. As such, we cannot infer the event time of the $\ell+1^{th}$ input tuple fed to an instance of O based on that of the ℓ^{th} tuple. We assume the information needed by an instance o_j to update its watermark $W^{\omega}_{o_j}$ is carried by tuples' metadata (cf. § 2.1), be it through $t.\tau$ and implicit watermarks, or by forwarding explicit watermarks (as special tuples or additional metadata of regular tuples). We assume ingresses/operator instances are continuously delivering tuples/watermarks and have completed their bootstrap phase (i.e., all have started delivering tuples/watermarks). We assume the threads in charge of the physical execution of O share memory (physical/logical) and can read/write shared objects stored in it.

Within this setup, we first show that SN parallelism might require data duplication to parallelize the execution of operators whose semantics are more general than those of A/J (e.g., those of our running example from § 1). We then prove that for such generalized semantics, implementations that rely on SN parallelism (with an arbitrary degree of data duplication) can be transformed to semantically equivalent implementations that rely on shared memory and do not incur the overheads of data duplication nor those of state transfer during elastic reconfigurations. Finally, we also provide an extensive discussion and evaluation (with several state-of-the-art baselines) based on a fully implemented prototype. For ease of presentation, we center most of our discussions around a single stateful operator. Nonetheless, *STRETCH* can support the execution of multiple stateful operators within one query, as we discuss in § 5 and § 7.

When it comes to elasticity, note that *STRETCH* does not aim at embedding a specific policy about when/how to balance load or provision/decommission instances (e.g., based on energy [16] or CPU consumption [4]), but rather defines a generic API for external modules. We show in § 8 the use of *STRETCH* in conjunction with two such modules.

4 A Generalized Stateful Operator

As introduced in § 1, data duplication is a potential drawback of SN parallelism. We prove herein that there exist operators whose semantics are richer than those of A and J (cf. § 2), and which might need data duplication for some of their tuples. To frame the type of stateful operator modeled by *STRETCH*, we then introduce a unified and *generalized operator* O^+, later used when arguing about the semantic equivalence between SN and VSN setups. With O^+ each input tuple t can be shared with an arbitrary number of instances, thus accounting for any data duplication level.

4.1 SN parallelism and data duplication

Before proving the data duplication need, we introduce the following lemma and definitions.

Lemma 1. *Let $\mathbb{W}$ be the set of non-expired window instances held by o_j for key k. The next tuple with key k fed to o_j can fall in a window instance w such that $w \in \mathbb{W}$.*

Proof. Being w^* one of the latest non-expired window instance in $\mathbb{W}$ (i.e., one with the highest left boundary l) and t^* the next tuple fed to o_j, if the statement does not hold, then either (i) $W^{\omega}_{o_j} \geq w^*.l + WS$, which contradicts that $\mathbb{W}$ contains non-expired window instances, or (ii) $t^*.\tau \geq w^*.l + WS$, which contradicts our assumption from § 3 about future tuples' timestamps being not known. □

Definition 4. $f_{MK}(t)$ is a Multi Key-by function that, differently from $f_{SK}(t)$, does not result in one key but rather in a set (possibly empty) of keys.

Definition 5. A^+ and J^+ are generalizations of A and J, respectively, for which f_{MK} replaces f_{SK}.

Given Lemma 1 and Definitions 4 and 5, we can introduce our theorem about data duplication.

Theorem 1. *If any arbitrary pair of tuples t^{ℓ}, t^{m} fed to A^+/J^+ can share a key, then A^+/J^+ might need data duplication for SN parallelism to correctly enforce A^+/J^+'s semantics.*

Proof. Let us begin with A^+ and let us assume t^ℓ is the last tuple forwarded to a_+, one of A^+'s parallel instances, and that $t^{\ell+1}$ is the next tuple that has to be forwarded to one of A^+'s instances. If t^ℓ and $t^{\ell+1}$ indeed share one key, and given Definition 1 and Lemma 1, then $t^{\ell+1}$ must be sent to a_+ to correctly enforce A^+'s semantics. The same holds for $t^{\ell+2}, t^{\ell+3}$, etc. To avoid any data duplication, all data should be sent to a_+ to account for any future tuple that could share a key with a previous tuple. In this case, though, A^+ would not run in parallel. The same reasoning holds for J^+ and for two tuples t_R, t_L (one from stream R, one from L) that should be compared when sharing a common key. □

To support the following discussion about our generalized stateful operator O^+, we give the following corollary.

Corollary 1. *A^+ and J^+ semantics can be enforced combining M, A and J operators with SN parallelism.*

This is so since by preceding A and J with M operators, M can be used to process each incoming tuple t^ℓ so that (1) as many copies as keys in t^ℓ are made of t^ℓ, (2) each such copy t'^ℓ carries one of t^ℓ's keys in $t'^\ell.\varphi$ as an extra attribute, and (3) such attribute is then returned by the f_{SK} of A or J.

To further build on our running example (cf. § 1), note that the operator computing the longest tweet per-hour and per-hashtag is an A^+ operator. In this case, f_{MK} would extract each tweet's hashtag as a key, and f_A would find the longest tweet of each such key. Corollary 1 shows how the programmer intervention is required to enforce A^+'s semantics, in this case expressing A^+ as an M/A combination (we refer the reader to Appendix C for an extended example).

4.2 *STRETCH*'s generalized model for stateful analysis

After discussing A^+/J^+ generalizations in Theorem 1, we now introduce a general operator O^+, and a model for SN parallelism that (1) encapsulates the semantics of A^+/J^+ (and thus that of A and J, cf. Corollary 1), and (2) allows for an arbitrary degree of data duplication.

The logical and physical representations of O^+ resemble those shown in Figure 2. O^+ has I upstream peers $U_1, \ldots, U_I$ (which could be an arbitrary mixture of operators and ingresses) and one downstream peer D (an operator or an egress). We refer to the j-th instance of U_i, O^+, and D as $u_{i,j}$, o_j^+, and d_j, respectively; $q_{a,b}$ is the queue between instances a and b. O^+ is defined as follows:

$$O^+(WA, WS, I, f_{MK}, WT, S, f_\mu, f_U, f_O, f_S)$$

Besides the aforementioned parameters *WA*, *WS*, I, f_{MK}, *WT*, S, and f_μ, O^+ defines an *update* (f_U), an *output* (f_O), and a *slide* function (f_S) to maintain its window instances: f_U is invoked, upon the reception of t, to update the state of the instances of keys associated with t and (optionally) to return a set of tuple payloads to be forwarded to D; f_O is invoked when a set of instances expires, to return a set of tuple payloads to be forwarded to D; f_S is invoked when instances slide, to return the updated states for a set of I window instances that have just advanced by *WA* time units.

As shown in Table 1, a default behavior is associated with each function. To draw a parallel with Object-Oriented Programming, O^+ acts as a *superclass* of operators like A^+ and J^+ (we formally prove this later in the section). The latter can be instantiated by specializing some of its functions f_U, f_O, and f_S. The user is expected to specialize at least one function, since the default implementations result in O^+ maintaining all tuples that fall in each window instance (via f_U and f_S) but without producing any output tuple.

TABLE 1: Functions O^+ uses to maintain window instances.

	Input	Output	Default behavior
f_U	$\{w_1, \ldots, w_I\}, t$	$\{\zeta_1, \ldots, \zeta_I, \varphi^1, \ldots, \varphi^\ell\}$	Store t in $w.\zeta$ of t's sender. Returns no φ.
f_O	$\{w_1, \ldots, w_I\}$	$\{\varphi^1, \ldots, \varphi^\ell\}$	Returns no φ.
f_S	$\{w_1, \ldots, w_I\}$	$\{\zeta_1, \ldots, \zeta_I\}$	Purge stale tuples.

Implementation of `forward` *and* `process` *methods*

We start with the method `forwardSN` (`forward` for SN setups) run by $u_{i,j}$ instances (Alg. 1). As soon as a tuple t is ready to be forwarded to O^+, `forwardSN` retrieves the set of keys associated with t (L5). For each key k, it then adds t to the queue of each downstream peer responsible for at least one of t's keys (L6-7).

We move now to the description of method `processSN` (`process` for SN setups, Alg. 2). Upon reception of t (L31), the method updates o_j^+'s watermark (L32) with the information contained in t's metadata (if any, cf. § 2). It then proceeds by outputting the result of all expired window instances and shifting/removing them (L33-35). We use the notation $\sigma_j[k][\ell]$ to refer to the ℓ-th set of I window instances (one for each of the I upstream operators) maintained in σ_j for key k. It starts with the earliest ones, having ρ as left boundary (L34), outputs their result invoking f_O (L13) and shifts them (when WT = single) invoking f_S (L14-16) and/or removes them (L17-18), if WT = multi or if all window instances have an empty state. When no window instances with left boundary ρ remain, the method continues checking the ones starting at $\rho + WA$ (L35). After taking care of expired window instances, the method proceeds identifying the set of window instances to which t falls in (depending on *WT*) and adjusting ρ if needed (L21-24). For each such window instance (L25), and for each of t's keys responsibility of o^+ (L26) the method creates the corresponding window instances, if not already defined, and updates their state invoking f_U (L27-30).

Upon invoking f_U/f_O, O^+ sets the event time of any resulting output tuple to the right boundary of the window

Algorithm 1: Method `forwardSN` (`forward` for SN setups) for a $u_{i,j}$ instance. Invoked when a non-empty set of tuples is to be forwarded to O^+.

```
   Instance-local variables:
1  f_MK, f_μ // From O+'s definition
2  Method forwardSN({t^1, ..., t^ℓ})
3      for t ∈ {t^1, ..., t^ℓ} do
4          ℙ ← {} // empty set of downstream peers
5          𝕂 ← f_MK(t) // get keys of t
6          for k ∈ 𝕂 do ℙ ← ℙ ∪ f_μ(k)
7          for p ∈ ℙ do q_{u_{i,j},o_p^+}.add(t)
```

Algorithm 2: Method `processSN` (`process` for SN setups) for an o_j^+ instance. Runs when t is returned by $q_{u_{i,j},o_j^+}$.

Instance-local variables (besides $WA, WS, I, f_{MK}, WT, S, f_\mu, f_U, f_O, f_S$):

1 σ_j // o_j^+'s state, initially empty
2 W // o_j^+'s watermark, initially 0
3 ρ // earliest $w.l$ of any $w \in \sigma_j$, initially 0

Auxiliary methods:

4 `updateW(t)` // update W based on t's metadata
5 σ_j`.remove(`k, ℓ`)` // remove the ℓ-th set of I window instances for key k
6 σ_j`.set(`$k, \ell, \{\zeta_1, \ldots, \zeta_I\}$`)` // set states of the ℓ-th set of I window instances for key k
7 σ_j`.shift(`$k, \ell, \{\zeta_1, \ldots, \zeta_I\}$`)` // for the ℓ-th set of I w inst. for key k, increase $w.l$ by WA and set states
8 σ_j`.check&Create(`k, l`)` // add set of I w instances for key k and $w.l = l$ if they are not already in σ_j
9 `earliestWinL(t)` // get earliest $w.l$ for any w in which t falls
10 `latestWinL(t)` // get latest $w.l$ for any w in which t falls
11 `prepareOutTuples(`$\{\varphi^1, \ldots, \varphi^\ell\}$`)` // create an output tuple and set its metadata for each of the ℓ payloads

12 **Method** `forwardAndShift(k)` // forward results (if any) and shift/remove the w instances
13 `forward(prepareOutTuples(`$f_O(\sigma_j[k][1])$`))`
14 **if** $WT = single$ **then**
15 $\{\zeta_1, \ldots, \zeta_I\} \leftarrow f_S(\sigma_j[k][1])$
16 **if** $\exists i \in \{1, \ldots, I\} | \zeta_i \neq \emptyset$ **then** σ_j`.shift(`$k, 1, \{\zeta_1, \ldots, \zeta_I\}$`)`
17 **else** σ_j`.remove(`$k, 1$`)`
18 **else** σ_j`.remove(`$k, 1$`)`

19 **Method** `handleInputTuple(t)`
20 **if** $\exists k | k \in f_{MK}(t) \wedge f_\mu(k) = j$ **then**
21 $\tau_1 \leftarrow$`earliestWinL(t)` // Find $w.l$ of window instance to update
22 **if** $WT = single$ **then** $\tau_2 \leftarrow$`earliestWinL(t)`
23 **else** $\tau_2 \leftarrow$`latestWinL(t)`
24 **if** $\tau_1 < \rho$ **then** $\rho \leftarrow \tau_1$
25 **for** $\ell \in \{0, \ldots, (\tau_2 - \tau_1)/WA\}$ **do** // Create/update window instance
26 **for** $k \in f_{MK}(t) | f_\mu(k) = j$ **do**
27 σ_j`.check&Create(`$k, \tau_1 + \ell * WA$`)`
28 $\{\zeta_1, \ldots, \zeta_I, \varphi^1, \ldots, \varphi^\ell\} \leftarrow f_U(\sigma_j[k][\ell])$
29 `forward(prepareOutTuples(`$\varphi^1, \ldots, \varphi^\ell$`))` // forward results (if any)
30 σ_j`.set(`$k, \ell, \{\zeta_1, \ldots, \zeta_I\}$`)`

31 **Method** `processSN(t)`
32 `updateW(t)` // update o_j^+ watermark
33 **while** $\rho + WS < W$ **do** // while o_j^+ has expired window instances
34 **while** $\exists k | \sigma_j[k][1].l = \rho$ **do** `forwardAndShift(`k`)` // and an expired window inst. w starts at ρ, handle w
35 $\rho \leftarrow \rho + WA$ // update ρ value
36 `handleInputTuple(`t`)`

instances on which f_U / f_O have been invoked, as done by A and J (see § 2), using method `prepareOutTuples`.

Formal Guarantees

After covering the implementation of O^+, we now prove O^+ encapsulates the semantics of A^+/J^+ (and thus A/J).

Theorem 2. *O^+ guarantees A^+ and J^+ semantics (cf. Def. 5).*

Proof. The method `forwardSN` of $u_{i,j}$ implements the same semantics of those of an additional map, placed after each $u_{i,j}$ instance, that create copies of each tuple according to Corollary 1. Moreover, with O^+'s, A can be instantiated by setting $I = 1$ and using f_A as f_O and/or f_R as f_S. Similarly, J can be instantiated by setting $I = 2$ and matching tuples via f_J either incrementally with f_U or upon window instance expiration with f_O. □

An additional property of O^+ is about how watermarks can be delivered by its instances, as discussed next.

Lemma 2. *For each o^+ instance, output tuples timestamps' represent valid implicit watermarks, since $t^m.\tau \geq t^\ell.\tau$ for any pair of consecutive output tuples t^ℓ and t^m.*

Proof. As shown in Alg. 2 L33-35, o^+ produces output tuples in timestamp order. As such, each output tuple's timestamp is also a valid watermark, since no tuple $t^{\ell+1}$ delivered after t^ℓ can have $t^{\ell+1}.\tau < t^\ell.\tau$. □

We refer to Appendix D for several complete operator examples, including one for an A^+ running the example from § 1 and one for ScaleJoin [13] (a J^+ that we later use in § 8), and to Appendix E for an additional example that connects to Theorem 2 and shows how an execution of an A^+ results in the same state updates observed when A^+'s semantics are implemented using M and A operators.

Despite not being within the scope of this work to create a complete taxonomy of all the stateful analysis semantics O^+ can cover, but rather to show O^+ is a generalization of common stateful operators such as A and J, note that, in the spirit of a description of O^+ semantics:

1) differently from A and J, O^+ can work with an arbitrary number of upstream peers each producing tuples with their own schema, and
2) O^+ can implement custom stateful operators (e.g., the ScaleJoin operator, presented in Appendix D and § 8).

5 VSN Parallelism and Elasticity

After introducing O^+, we show O^+ can leverage shared memory to avoid data duplication and state transfers during

TABLE 2: API of the *TB* shared data structure. Highlighted methods are only relevant for elastic setups.

API Method	Description
add(t, j)	invoked by source j to add tuple t.
get(j)	invoked by reader j to retrieve the next tuple conforming to the watermarks that are also delivered by the `get` method.
addReaders$(\mathbb{R}, j)$	invoked by reader j to add readers in $\mathbb{R}$ that are not already readers of *TB*. Once invoked, *TB* delivers to $\mathbb{R}$ tuples and watermarks starting from the ones that will be also returned to j once j invokes `get`. If more readers invoke `addReaders` concurrently, only one succeeds. Returns `true` only if it adds all new readers in $\mathbb{R}$.
removeReaders$(\mathbb{R})$	removes each of the readers in $\mathbb{R}$ (that are readers of *TB*) when invoked by a reader of *TB*. If more readers invoke `removeReaders` concurrently, only one succeeds. Returns `true` only if it removes all existing readers in $\mathbb{R}$.
addSources$(\mathbb{S})$	adds each of the sources in $\mathbb{S}$ that are not already sources of *TB* when invoked by a source of *TB*. If more sources invoke `addSource` concurrently, only one succeeds. Returns `true` only if it adds all new sources in $\mathbb{S}$.
removeSources$(\mathbb{S})$	removes each of the sources in $\mathbb{S}$ (that are sources of *TB*) when invoked by a source of *TB*. If more sources invoke `removeSources` concurrently, only one succeeds. Returns `true` only if it removes all existing sources in $\mathbb{S}$.

elastic reconfigurations while preserving its semantics. We first focus on how VSN parallelism can overcome the data duplication overhead. For ease of exposition, we begin with a static setup in which $\Pi(O^+)$ is fixed.

In *STRETCH*, we assume that each pair of instances $\langle u_{i,j}, o_j^+ \rangle$ has no dedicated queue (cf. § 2.2), but rather that all U_i and O^+ instances share a single *Tuple Buffer* (*TB*) object (cf. Figure 3) which behave according to the next definition.

Definition 6. *TB* is a data structure that allows a set of sources to concurrently add tuples to it, that delivers each tuple exactly once to each one of its readers, and that merges sources' watermarks into a single stream of non-decreasing watermarks, each delivered to all readers. It defines the methods presented in Table 2.

We rely on a generic *TB* data structure for generality. We discuss a specific data structure with such an API in § 6.

Implementation of the `forward` method

In this new setup, $u_{i,j}$ instances run the method `forwardVSN` (`forward` method for VSN setups) in Alg. 3. Each $u_{i,j}$ instance carries an id (L1) that represents its index as source of *TB* (i.e., 1 for $u_{1,1}$ and $\sum_i \Pi(U_i)$ for $u_{I,\Pi(U_I)}$) and passes such id when adding tuples to *TB* via the `add` method (L 3). In this case, since all tuples are visible to each o^+ instance, o^+ should process only the tuples that carry at least one key that is its responsibility. Noting that, nonetheless, this is already the case in method `handleInputTuple` (Alg. 2 L26), we make the following observation.

Algorithm 3: Method `forwardVSN` (`forward` for VSN setups) for a $u_{i,j}$ instance. Invoked when a non-empty set of tuples is to be forwarded to O^+.

```
   Instance-local variables:
1  id // instance's id
2  Function forwardVSN({t_1, ..., t_m})
3  |  for t ∈ {t_1, ..., t_m} do TB_in.add(t, id)
```

Observation 2. *When U_i and O^+ instances are connected through TB objects, and U_i instances use Alg. 3 to forward tuples, O^+ can run in parallel enforcing correctly O^+'s semantics without data duplication.*

Observation 2 focuses on the data duplication overhead for a single operator O^+. When considering a series of operators, note that tuples' data could be duplicated also when D is stateful and o^+ and d_j instances share tuples through dedicated queues. Hence, for generality, we also replace the queues of $\langle o_j^+, d_j \rangle$ pairs with a *TB*, as shown in Figure 3 (we name the objects TB_{in} and TB_{out} to differentiate them). Note our model assumes one D operator for simplicity; our results hold also if more D operators invoke `get` on TB_{out}.

From static to elastic setups

We now show that if o_j^+ instances share a global state σ – rather than per-instance σ_j states – this can enable state-transfer-free elasticity (cf. § 1). We thus move from a static setup in which $\Pi(O^+)$ and f_μ are fixed to one in which both can change over time. It should be noted that, as we formally prove later in the section, although σ can potentially be exposed to concurrent updates for distinct keys from the various o_j^+ instances, *STRETCH* ensures that each key is consistently updated by exactly one instance at a time.

A reconfiguration implies a change in f_μ to hold from a certain event time onward. We use the term *epoch* to refer to the event time period in between two watermarks during which the mapping of keys to instances does not change. Hence, being e the current epoch, $\mathbb{O}$ the set of O^+ instances, and f_μ the mapping in e, a reconfiguration implies the start of a new epoch e^* for which a new mapping f_μ^* is used for a (possibly different) set of operator instances $\mathbb{O}^*$. We focus on a single transition from epoch e to e^* since subsequent epoch switches happen with the same logic. To ease presentation, we define two temporary conditions:

- **Cond. 1:** For provisioning reconfigurations, the joining instances $\mathbb{O}^* \setminus \mathbb{O}$ are already instantiated, and start retrieving/processing tuples as soon as they are given access to TB_{in}. For decommissioning reconfigurations, the leaving instances $\mathbb{O} \setminus \mathbb{O}^*$ will stop processing/outputting tuples once they are disconnected from TB_{in} and TB_{out}.
- **Cond. 2:** All instances $o_j^+ \in \mathbb{O} \cup \mathbb{O}^*$ have access to a set of variables $\{e, e^*, \mathbb{O}, \mathbb{O}^*, f_\mu^*, \gamma\}$ that represent the current epoch id number (e), the next epoch id number (e^*), the set of instances of the current epoch ($\mathbb{O}$), the set of instances of the next epoch ($\mathbb{O}^*$), the mapping function of the next epoch (f_μ^*), and a shared event time (γ) greater than the current watermark of any instance o_j^+, used by all o_j^+ instances to trigger a reconfiguration as soon as their watermark is greater than or equal to γ. O^+ instances receive special *control* tuples, and set such parameters using a method named `prepareReconfig`.

Control tuples can be distinguished with a method named `isControl` and are not processed to update O^+'s state.

Figure 3 shows *STRETCH*'s setup. We cover the actual coordination of instances, the implementation of `isControl` and `prepareReconfig`, and satisfy Cond. 1/2 in § 7.

Elasticity and correctness

Before focusing on *STRETCH*'s shared state and elasticity protocol, we want to draw attention to one challenging connection between elasticity and correctness guarantees. A change in $\Pi(O^+)$ implies a change in the number of instances delivering tuples/watermarks to D. Independently of whether tuples/watermarks are delivered to D through individual queues or *TB* objects, a crucial question arises: *can a newly provisioned instance deliver tuples/watermarks to a d_l instance that conflict with $W^{\omega}_{d_l}$ (i.e., that carry a timestamp earlier than $W^{\omega}_{d_l}$)?* Without assumptions on when such operator instance will deliver tuples/watermarks or which values they will carry, this is indeed the case. Note this could violate correctness if D is stateful and t contributes to a window instance w that d_l has already treated as expired.

One way, discussed in the literature [17], to deal with this is to relax the correctness guarantees. An alternative way, which we follow in this work, is to prove that, for an O^+ operator, it is possible to guarantee a safe lower bound on the watermarks delivered by newly provisioned instances and thus consistently deliver tuples/watermarks to D (this is shown in detail in Lemma 3, later in this section).

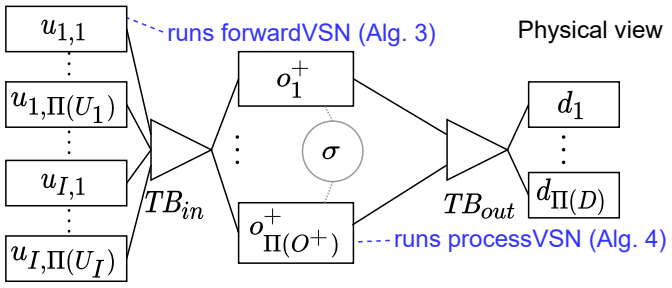


Fig. 3: *STRETCH*'s model for VSN parallelism/elasticity.

Implementation of the `process` method

Alg. 4 overviews the implementation of O^+ for VSN parallelism and elasticity. It resembles the structure of the one in Alg. 2 but has important differences.

The first difference between the algorithms is the additional L13-21 to handle the elastic reconfigurations. As shown, the algorithm first checks if the tuple is a control one (L13). If that is the case, it proceeds storing γ (the event time triggering the reconfiguration) and the future values of $e, \mathbb{O}, f_\mu$ in the respective variables $e^*, \mathbb{O}^*, f^*_\mu$. Notice that, as shown, a control tuple itself does not trigger a reconfiguration immediately. The reconfiguration is later triggered when a new incoming regular tuple increases the watermark and such increased watermark is greater than γ (L17). In this case, the algorithm enters a barrier and, once all instances reach such barrier, proceeds to apply the reconfiguration (L18-21) before processing t (L22 onward).

The second difference between the algorithms is that the code handling O^+'s state when producing output tuples (L22-24). In this case, each instance o_j^+ handles an expired window instance w only if $w.k$ is o_j^+'s responsibility (L23). Together with method `handleInputTuple`, which handles only the keys responsibility of o_j^+, this prevents concurrent modifications for the same key in σ.

Formal Guarantees

Given Alg. 4, we can now formally prove that *STRETCH* ensures that each key maintained in the shared state σ is consistently updated by exactly one instance at a time.

Theorem 3. *If $\mathbb{O} \cup \mathbb{O}^*$ instances run the `processVSN` method presented in Alg. 4, then elastic reconfigurations can be carried out while preserving O^+'s semantics (cf. § 4.2).*

Proof. To begin, note that when processing each tuple, all instances in $\mathbb{O}$ use the same f_μ (L13-21). Hence, each key k is only updated by the instance responsible for k.

We argue that each key k is also updated exclusively and consistently by one instance in the presence of reconfigurations. Each instance in $\mathbb{O}$ enters the `if` statement at L17 only after it receives a tuple t that increases its watermark to the first value greater than γ and then proceeds to wait for all other instances (L18). As specified in Definition 6, *TB* objects deliver the same watermarks to all instances, and each watermark observed by an instance has non-decreasing values. Before entering the barrier, all instances have already handled expired window instances whose right boundary fell before $\overline{W}$ using f_μ. Once leaving the barrier, all instances will consistently handle window instances whose right boundary falls after $\overline{W}$ using f^*_μ (both expired and non-expired ones). Newly provisioned instances (if any) are connected to TB_{in}/TB_{out}, or alternatively instances being decommissioned (if any) are disconnected from TB_{in}/TB_{out} by exactly one of the existing instances (the one that succeeds in adding the sources being provisioned or in removing the readers being decommissioned). Hence, if t carries a key k whose responsibility has shifted from o_i^+ to o_j^+, independently of whether o_j^+ is a newly deployed instance or not, o_j^+ will not only process t (in relation to key k) after o_i^+ is done processing all tuples preceding t, but also process t after having been connected to both TB_{in} and TB_{out}, if o_j^+ is a newly deployed instance. If o_i^+ is being decommissioned, o_i^+ will not process t (no key responsibility of o_i^+ is returned by f^*_μ) and o_i^+ will also not retrieve any tuple from TB_{in} nor will it output to TB_{out} once disconnected from the latter. □

After proving Alg. 4 can support parallel execution and elastic reconfigurations for O^+ while enforcing O^+'s semantics correctly, we now prove it also enables VSN parallelism/elasticity for any downstream peer. This is because, even in the presence of provisioning reconfigurations, O^+ can consistently deliver tuples/watermarks to TB_{out}, and the latter can act as the TB_{in} of D.

Lemma 3. *Being t the tuple that triggers a reconfiguration (entering the `if` statement at L17) $t.\tau$ is a safe lower bound for the watermark of newly provisioned instances.*

Proof. The method `addSources` (L19) is invoked successfully by exactly one o_j^+ instance only upon reception of a

Algorithm 4: Method `processVSN` (`process` for VSN setups) of a o_j^+ instance. Runs when t is returned by TB_{in}.

Instance-local variables (besides *WA*, *WS*, I, f_{MK}, *WT*, S, f_μ, f_U, f_O, f_S):

1 W // o_j^+'s watermark, initially 0
2 ρ // earliest $w.l$ of any $w \in \sigma$, initially 0
3 e, e^* // current/next epoch number
4 $\mathbb{O}, \mathbb{O}^*$ // set of current/next epoch instances
5 f_μ^* // next epoch f_μ
6 γ // Event time to trigger reconfiguration

Shared variables:

7 σ // shared state σ

Auxiliary functions: As defined in Alg. 2, but operating on σ rather than σ_j

8 `updateW(t)`, `σ.remove(k,ℓ)`, `σ.set(k,ℓ,{ζ1,...,ζI})`, `σ.shift(k,ℓ,{ζ1,...,ζI})`, `σ.check&create(k,l)`, `earliestWinL(t)`, `latestWinL(t)`, `prepareOutTuples({φ1,...,φℓ})`, `forwardAndShift(k)`, `handleInputTuple(t)`

9 `waitForInstances(𝕆)` // blocking call acting as a barrier
10 `isControl(t)` // check if t is a control tuple
11 `prepareReconfig(t)` // setup reconfig-related parameters

12 **Function** `processVSN(t)`
13 **if** `isControl(t)` **then** $\{e^*, \mathbb{O}^*, f_\mu^*, \gamma\} \leftarrow$`prepareReconfig(t)` // set up reconfiguration parameters. The reconfiguration is triggered as soon as W grows beyond γ
14 **else**
15 $\overline{W} \leftarrow W$
16 `updateW(t)` // update o_j^+ watermark
17 **if** $W > \overline{W} \wedge W > \gamma$ **then**
18 `waitForInstances(𝕆)`
19 **if** $|\mathbb{O}^*| > |\mathbb{O}| \wedge TB_{out}$.*addSources*$(\mathbb{O}^* \setminus \mathbb{O})$ **then** TB_{in}.`addReaders(𝕆* \ 𝕆, j)` // provision instances. The if clause ensures exactly one instance adds new instances first to TB_{out} and then to TB_{in}
20 **if** $|\mathbb{O}^*| < |\mathbb{O}| \wedge TB_{in}$.*removeReaders*$(\mathbb{O} \setminus \mathbb{O}^*)$ **then** TB_{out}.`removeSources(𝕆 \ 𝕆*)` // decommission insts. The if clause ensures exactly one instance removes instances first from TB_{in} and then from TB_{out}
21 $\{e, \mathbb{O}, f_\mu\} \leftarrow \{e^*, \mathbb{O}^*, f_\mu^*\}$
22 **while** $\rho + WS < W$ **do** // while o_j^+ has expired window inst. it is responsible for and starting at ρ
23 **while** $\exists k | \sigma[k][1].l = \rho \wedge f_\mu(k) = j$ **do** `forwardAndShift(k)`
24 $\rho \leftarrow \rho + WA$ // update ρ value
25 `handleInputTuple(t)`

tuple t that increases o_j^+'s watermark (from $\overline{W}$ to W, L15-18). All results that could have been produced before processing t have already been delivered to *TB* by all instances (Theorem 3) and have timestamp lower than or equal to $\overline{W}$ and thus lower than or equal to $t.\tau$ (since $t.\tau \geq \overline{W}$, cf. Definition 2), while all results that could depend on t or future tuples will have a timestamp greater than $t.\tau$ (cf. Observation 1). If according to Lemma 2, the timestamps of the tuples produced by o_j^+ can be used as watermarks by *TB*, then $t.\tau$ can be immediately delivered as a watermark for a newly provisioned instance o_j^+. □

6 Using ScaleGate as *TB* Objects

§ 5 covered *STRETCH*'s algorithms for VSN parallelism and elasticity. The latter relies on the *TB* data object, which exposes six methods, presented in Table 2. Here, we show how an enhanced SG object (cf. § 2.4) can offer all such methods and support a real implementation of *STRETCH*.

We begin observing that the original SG is already sufficient to provide methods `add` and `get`. More formally, under the assumption that each source delivers a timestamp-sorted stream of tuples, SG objects allow concurrent insertion and retrieval of tuples for arbitrary sets of sources and readers, delivering each tuple exactly once, and also delivering non-decreasing implicit watermarks (cf. § 2.3) through tuples' $t.\tau$ attribute. As discussed in § 4, each o_j^+ outputs window results in timestamp-order (Lemma 2). It is thus safe for each o_j^+ to deliver its output tuples to an SG data structure that can support data-duplication-free parallelism for downstream peers too, in a composable fashion. In this case, both O^+ as well as D (if the latter is a stateful operator) can support correct execution for both order-sensitive as well as order-insensitive functions.

SG objects do not define methods to dynamically change their sources and readers. They can be extended to provide the API methods highlighted in Table 2, nonetheless. We refer to such extended objects as *Elastic ScaleGate* (ESG) objects. In order to outline our ESG implementation, we first overview the internals of the `add` and `get` methods. SG builds a skip list where tuples are maintained in timestamp order, along with some auxiliary book-keeping structures.

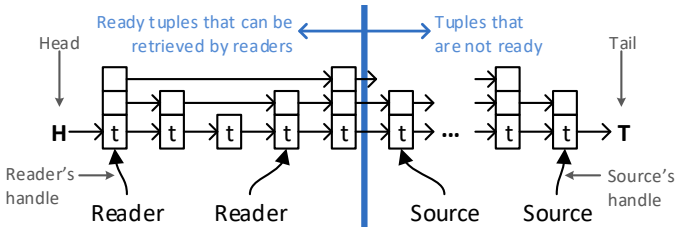


Fig. 4: ScaleGate's skip list, and readers'/sources' handles.

The book-keeping structures contain handles to the skip list, for sources and readers, to continue inserting or reading nodes (tuples), respectively. As shown in Figure 4, readers' handles traverse the list from head to tail, retrieving the next tuple only if the latter is not pointed by a source's handle (thus returning only ready tuples). At the same

time, sources' handles point to their last inserted tuples and facilitate the sorted insertion of subsequent tuples (also leveraging the skip list shortcuts). Since each source adds a timestamp-sorted stream, each insertion "falls" after its previous one. All the tuples before the earliest tuple pointed by a source (i.e., with earlier timestamps), are ready.

Adding new readers: Each reader has access to one of ESG's nodes through its own handle. Each new reader simply needs a handle to the node pointed by the j-th reader (the one invoking the `addReaders` method) to retrieve next the same tuple that will be delivered to the j-th reader (according to the API), and then traverse the rest of the list in timestamp order in subsequent `get` invocations.

Removing existing readers: Removing a set of existing readers only requires removing their thread-specific structures.

Adding new sources: According to Lemma 3, the timestamp of the tuple t triggering a reconfiguration is a safe lower bound for the watermark of newly provisioned instances in *STRETCH*. Since t is the last tuple observed by the instance invoking method `addSources`, and given that $t_o.\tau < W^{\omega}_{o_j^+} < t.\tau$, being t_o the last tuple produced by such o_j^+ instance, the book-keeping handles of new sources can be copying the handles of the source invoking successfully the `addSources` method. For the sake of the new thread, an initial *dummy* tuple following the one pointed by the source invoking successfully the `addSources` method is inserted, to initialize the functionality of its handles. Readers can move their pointer to the next tuple pointed by a source when such tuple is of type dummy but the tuple is not returned as ready to readers invoking `get`.

Removing existing sources: Removing a source consists mainly of adding, on behalf of the source, a special *flush* tuple in ESG, with a timestamp equal to the latest insertion of the source. Such a tuple will let the previously added tuples of the leaving source to be ready (according also to other sources' tuples). The removed source's associated book-keeping structures can then be removed. As for dummy tuples, readers can move their pointer to the next tuple pointed by a source when such tuple is of type flush, without returning it as ready via the `get` method.

Concurrent calls to the API methods: For concurrent calls of the same method that updates the set of threads (e.g. concurrent calls to `addReaders`), synchronization is in place (using a TestAndSet variable) so that only one of each type takes effect. Concurrent calls among competing such methods that modify the thread-specific book-keeping structures (e.g. `addReaders` and `removeReaders`) require extra synchronization to protect consistency; since these are low-contention operations, nonetheless, a simple lock can do. Since each reconfiguration results in sources/readers being added or removed but not both (Alg. 4 L19-20), and each reconfiguration can only start after the previous is completed (cf. § 7), we do not incur such extra synchronization overhead in our implementation. If regular operations (`add` and `get`) are concurrent with those that update the set of threads and their book-keeping structures, the latter can overwrite, causing the former to have no effect. Note that in *STRETCH* such invocations do not interfere.

7 Implementation - API and Architecture

We focus herein on how *STRETCH* meets Cond. 1 and Cond. 2 (cf. § 5). We begin overviewing the overall architecture of our implementation and discussing Cond. 1.

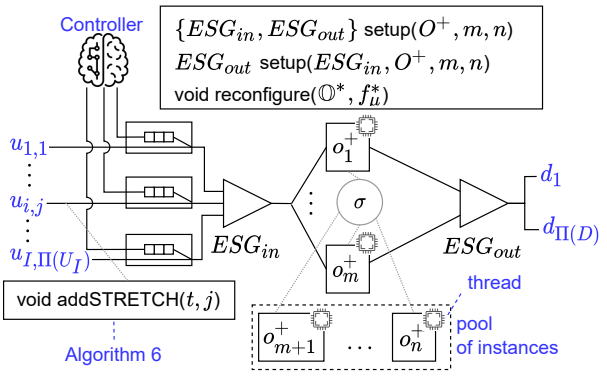


Fig. 5: API and architecture of *STRETCH*'s implementation.

As shown in Figure 5, *STRETCH*'s API defines two `setup` and one `reconfigure` method. Method $\{ESG_{in}, ESG_{out}\} = \texttt{setup}(O^+, m, n)$ takes as input the O^+ to instantiate, its initial parallelism degree m and its maximum parallelism degree n. Upon invocation of this method, *STRETCH* creates n o_j^+ that share state σ. Furthermore, it connects m of them to ESG_{in} and ESG_{out}. The remaining $n - m$ instances are kept in a pool. The pool is also used to keep instances that are later decommissioned.

Each o_j^+ instance is run by a thread. The latter is constantly trying to get a tuple to process through the `get` method of ESG_{in}. When no tuple is retrieved, either because no tuple is ready or because o_j^+ belongs to the pool and is thus not connected to ESG_{in}, exponential backoff prevents the thread from creating contention on ESG_{in}. Instances can promptly start retrieving tuples once provisioned and connected to ESG_{in}, thus invoking method `processVSN`, while decommissioned/pool instances will create negligible contention on ESG_{in}, and stop invoking the `processVSN`/`forwardVSN` methods, as per Cond. 1.

The `setup` method returns both ESG_{in} and ESG_{out}, for the latter to be fed by upstream and to feed downstream instances, respectively. As mentioned in § 3, we focus our discussions on a single stateful operator for simplicity, but *STRETCH* can be used to instantiate many (connected) operators within a query. Because of this, the $\{ESG_{out}\} = \texttt{setup}(ESG_{in}, O^+, m, n)$ variation can be used to create an O^+ operator and connect it to a previously created one through ESG_{in} (i.e., the ESG_{out} of such upstream peer).

Algorithm 5: Method `addSTRETCH`.

```
   Instance-local variables:
1  q[] // Queues holding reconfiguration messages
2  T[] // τ of the latest tuple added by source i
3  Function addSTRETCH(t, i)
4      T[i] ← t.τ
5      while q[i].size() > 0 do
6          x ← q[i].pop()
7          ESG_in.add(⟨T[i], control, [x.e, x.𝕆, x.f_μ]⟩, i)
8      ESG_in.add(t, i)
```

Algorithm 6: Method `prepareReconfig` used in Alg. 4, to set the instance-local variables needed during a reconfiguration.

```
1 Function prepareReconfig(t)
2     if t.φ[1] > e then // the reconfiguration id
        carried by t is greater than O+'s one
3         e* ← t.φ[1] // set reconfig. parameters
4         𝕆* ← t.φ[2]
5         f*_μ ← t.φ[3]
6         γ ← t.τ
```

Triggering elastic reconfigurations

As discussed in § 5, elastic reconfigurations depend on the parameters e^*, $\mathbb{O}^*$, f^*_μ, γ, which in Alg. 4 are handled by methods `isControl` and `prepareReconfig`, detail next.

In *STRETCH*, a reconfiguration is triggered by the external module sharing a new set of instances ids $\mathbb{O}^*$ and a mapping function f^*_μ. To deliver $\mathbb{O}^*$ and f^*_μ to O^+ instances, *STRETCH* encapsulates them in a special control tuple. Method `isControl` distinguishes regular from control tuples based on the attributes carried in their metadata.

It should be noted that, even if regular as well as control tuples can be delivered by $U_{i,j}$ instances and the controller, respectively, ESG_{in} still expects each of its sources to deliver tuples in timestamp order. To fulfill such a condition, *STRETCH* defines one dedicated control queue per upstream instance (as shown in Figure 5), and wraps the `add` method of ESG_{in} within the method `addSTRETCH`, shown in Alg. 5. With this method, *STRETCH* tracks the last timestamp τ forwarded by each upstream instance (Alg. 5 L4) and, by having the `reconfigure` method add the next $\mathbb{O}^*$, f^*_μ in each control queue, creates and forwards a control tuple carrying timestamp τ and $\mathbb{O}^*$, f^*_μ in its metadata. Control tuples can then be processed as shown in Alg. 6.

Formal Guarantees

Theorem 4. *Each reconfiguration that is applied based on Alg. 4, Alg. 5, and Alg. 6 takes place atomically and exactly once.*

Proof. As shown in Alg. 4, all o^+_j perform an elastic reconfiguration based on their local parameters e^*, $\mathbb{O}^*$, f^*_μ, and γ, with exactly one instance succeeding in connecting or disconnecting provisioned or decommissioned instances from TB_{in}/TB_{out}, respectively. These parameters are set by Alg. 6 based on a control tuple t, delivered by ESG_{in} to all o^+_j. Furthermore, if multiple such control tuples are delivered by ESG_{in}, all are delivered in the same order to o^+_j instances. Hence, all instances switch to the same e^* at the same point in time. If multiple e^* are delivered by multiple control tuples from ESG_{in}, the latest one is applied, and such latest one is the same for all o^+_j instances. □

8 Evaluation

We aim at comparing *STRETCH* with other state-of-the-art baselines. We place emphasis on stream joins since joins are among the most computationally heavy operators [18]. The choice of focusing on stream joins is also motivated by the existence of [13], a custom highly-specialized implementation of the VSN parallelism offered by *STRETCH* that, while supporting only a fraction of the stateful analysis that *STRETCH* can support, provides the best performance figures at which *STRETCH* can aspire. To account also for other state-of-the-art baselines, we also compare with Apache Flink (or simply Flink). First, we rely on an established baseline, word-count [19] and a variation counting distinct *pairs* of words, to study the effects of different data duplication levels on throughput and latency metrics. We then study the maximum performance *STRETCH* and Flink can offer for O^+ operators with $I = 2$ (i.e., including joins). Since both ScaleJoin and *STRETCH* support correct execution for order-sensitive functions, we assume that in SN setups input tuples are merged-sorted by both o^+_j and d_j instances. Our experiments seek answers to the questions found in Table 3: Q_1-Q_3 assume a static setup, while Q_4-Q_6 focus on elastic setups and also on real-world applications.

Experiments are run on a 2.10 GHz Intel(R) Xeon(R) E5-2695 CPU with 2 18-cores sockets, 72 logical threads with hyper-threading, and 64 GB memory. *STRETCH* is implemented in Java and tested with Java HotSpot(TM) 64-bit Server VM. For SN, we use Flink 1.6.0.

In the following, we present and discuss the results of the performance metrics of interest, averaged over five runs. More concretely, we use input rate – computed as the number of tuples/second (t/s) processed by an operator, throughput (for join operators) – computed as the number of comparisons/second (c/s) sustained by the operator, and latency – computed as the timestamp difference of each output tuple and the latest input tuple that contributes to it, while using *flow control* to handle backpressure. The implemented flow control mechanism is similar to that of Flink, in this case, putting a bound on ESG's size. For experiments concerning elasticity, we also report reconfiguration times – measured as the time difference between the moment the controller invokes method `reconfigure` (cf. § 7) to the moment the reconfiguration is completed, and the number of threads used throughout the experiment.

8.1 Throughput and latency in VSN (*STRETCH*) vs SN (Flink) for established baselines such as *wordcount* (Q_1)

In our first experiment, we use the `wordcount` benchmark [19], frequently used in applications like Sentiment Analysis ones [20]. To account for different amounts of duplication, we also run a `paircount` variation, that counts pairs of rather than individual words. In these experiments, *STRETCH*'s shared-memory allows each input tuple to be shared with all the parallel threads, having each one responsible for one word/pair, while Flink's shared-nothing approach requires each tuple to be transformed in multiple output tuples, according to Corollary 1. The definitions for all the used operators are found in Appendix D.

We process a dataset consisting of 4.3 million tweets, between the 1^{st} and 2^{nd} of October 2018. The input schema is defined as $\langle \tau, [\text{user}, \text{tweet}] \rangle$. With `wordcount`, each input tuple t results in as many output tuples as words carried in $t.\varphi[2]$. With `paircount`, each word in $t.\varphi[2]$ is paired and forwarded with its nearby words, up to a distance of 3, 10, and $+\infty$ for the Low (L), Medium (M), and High (H) duplication cases, respectively. `Wordcount` gives the least amount of duplication, `paircount` H the greatest.

Results are presented in Figure 6. Flink performance is shown as a shaded region since, according to Corollary 1,

TABLE 3: Questions addressed in the evaluation, together with information about the operators used in the experiments.

ID	Setup	Question	Operator	Section
Q_1	Static	How do VSN (*STRETCH*) and SN (Flink) compare for established baselines such as *wordcount*?	A^+ implementing `wordcount` and `paircount` (a variation that counts distinct pairs of words)	§ 8.1
Q_2	Static	What is the maximum throughput/minimum latency for VSN/SN setups in *STRETCH* and Flink?	Operator 6 (cf. Appendix D), processing tweets collected during October 2018.	§ 8.2
Q_3	Static	How does *STRETCH* compare with ad-hoc stateful operator implementations such as ScaleJoin (Operator 3)?	ScaleJoin (Operator 3, cf. Appendix D) with *WA*/*WS* set to δ and 5 minutes, resp., running the benchmark from [13].	§ 8.3
Q_4	Elastic	How long does it take for *STRETCH* to complete an elastic reconfiguration?	ScaleJoin (Operator 3, cf. Appendix D) with *WA*/*WS* set to δ and 5 minutes, resp.	§ 8.4
Q_5	Elastic	What is *STRETCH* performance under multiple reconfigurations?	ScaleJoin (Operator 3, cf. Appendix D) with *WA*/*WS* set to δ and 1 minute, resp.	§ 8.5
Q_6	Elastic	What is *STRETCH* performance in real-word applications?	ScaleJoin (Operator 3, cf. Appendix D) with *WA*/*WS* set to δ and 30 seconds, resp., analyzing financial trades	§ 8.6

a Map M is required to split each input tuple into distinct words/pairs, and such M can also be executed in parallel. The bottom and upper lines represent the minimum and maximum throughput/latency observed for any parallelism degree of M in $[1, 36]$. As shown, in the `wordcount` case (the one with the least amount of duplication) *STRETCH* and Flink are comparable. Despite its degradation (because of the contention on shared resources) after reaching its peak throughput, *STRETCH* is nonetheless able to achieve $+17\%$ throughput and -94% latency. *STRETCH*'s benefits become even more evident for the `paircount` cases, achieving $+137\%$, $+237\%$, and $+283\%$ throughput and -89%, -94%, and -94% latency for L, M and H, respectively. Note in this case the throughput is decreasing for an increasing duplication level (e.g., from counting words to counting pairs) since each input tuple (carrying a tweet) results in a higher number of pairs, thus resulting in a higher workload.

8.2 Maximum throughput and minimum latency in VSN (*STRETCH*) vs SN (Flink) for an O^+ with $I = 2$ (Q_2)

This experiment compares *STRETCH* performance using Flink as a baseline for SN operators that, like Joins, define two input streams. Since Flink does not offer parallel execution of general Joins (but only EquiJoins), we evaluate the performance of an operator that simply forwards each incoming tuple, and thus measures the maximum throughput/minimum latency observed when the performance bottleneck is given by data sharing and sorting (its definition is found in Appendix D). We use the same dataset from Q_1.

Figure 7 shows the operators' scalability. For an increasing $\Pi(O^+)$, *STRETCH*'s throughput decreases from approximately 120 000 to 100 000 t/s due to the synchronization overheads induced by the higher $\Pi(O^+)$. Flink starts from a lower throughput and decreases faster, from 40 000 to 2 000 t/s. *STRETCH* achieves from $3\times$ to $50\times$ better throughput. Flink's latency, regardless of $\Pi(O^+)$, is higher than 100 ms, while *STRETCH*'s is always less than 30 ms.

8.3 Throughput/latency of J^+ in *STRETCH* vs ad-hoc implementations (Q_3)

After studying Operator 6's maximum throughput/minimum latency, we focus now on J^+ (Operator 3). Since Flink

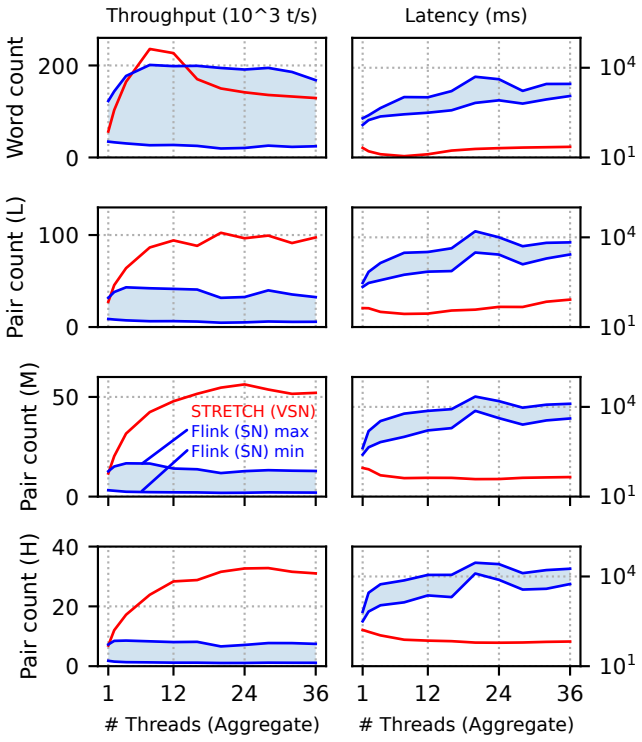


Fig. 6: *STRETCH* (VSN) and Flink (SN) Throughput/Latency for `wordcount` and `paircount`

supports only parallel equijoins, we compare *STRETCH* performance with that of the original ScaleJoin implementation [13] and with an additional optimized single-threaded implementation, which we refer to as 1T. The latter allows measuring the performance of an implementation that devotes as many CPU cycles as possible to data analysis rather than data communication when $\Pi(J^+) = 1$.

We follow the same benchmark used in [13], [21] to join two logical streams L and R. L tuples' schema is $\langle \tau, [x, y] \rangle$, where x is type of `int` and y is `float`. R tuples' schema is $\langle \tau, [a, b, c, d] \rangle$, where a, b, c, and d are of type `int`, `float`, `double` and `boolean`, respectively. *WA* and *WS* are set to δ (1 ms, as in Flink) and 5 minutes, respectively. For each pair

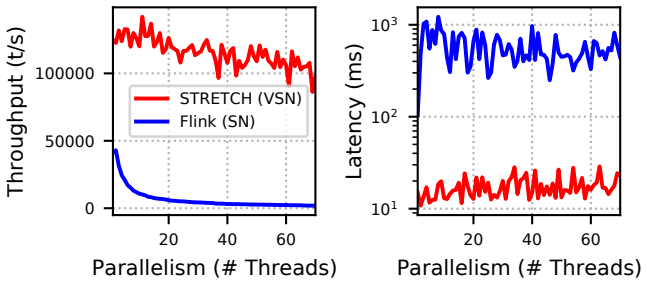


Fig. 7: Max Throughput/min latency of *STRETCH* (VSN) and Flink (SN) for a generic O^+ with $I = 2$ (Operator 6).

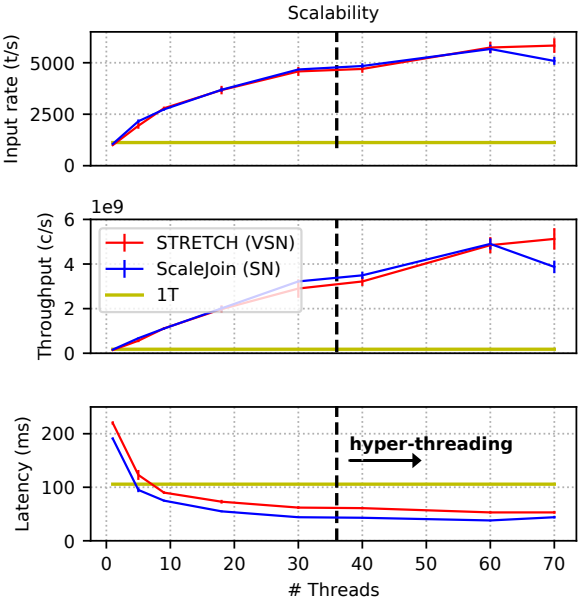


Fig. 8: The performance of *STRETCH* compared with those of ScaleJoin (SN) and 1T for J^+.

of tuples t_L and t_R, an output tuple is produced if:

$$t_R.\varphi[1] - 10 \leqslant t_L.\varphi[1] \leqslant t_R.\varphi[1] + 10 \bigwedge$$
$$t_R.\varphi[2] - 10 \leqslant t_L.\varphi[2] \leqslant t_R.\varphi[2] + 10$$

Attributes x, y, a and b are randomly selected from a uniform distribution with interval $[1, 10\,000]$ which results on average in an output tuple every 250 000 comparisons.

Figure 8 shows (i) the average of the maximum sustainable input rate and the standard deviation for increasing $\Pi(J^+)$, (ii) the corresponding throughput, in terms of number of comparisons, and (iii) the corresponding latency in logarithmic scale. The throughput of *STRETCH* and ScaleJoin when $\Pi(J^+) = 1$ is similar to 1T, but the latency for 1T is lower than the other two, because of *STRETCH*'s and ScaleJoin's data communication costs. As shown, despite being a general rather than a specialized operator, *STRETCH*'s throughput still grows linearly with $\Pi(J^+)$ and can match that of ScaleJoin (the latter shows some higher degradation caused by hyper-threading when $\Pi(J^+)$ exceeds the 36 physical threads), despite a small overhead (in the order of 10 ms) in latency.

8.4 *STRETCH* – elasticity overheads (Q_4)

We now move from static to elastic setups for the ScaleJoin benchmark (cf. § 8.3). With Q_4, we measure the overhead and duration of individual elastic reconfigurations in isolation. We trigger one elastic reconfiguration per experiment using a simple controller. The latter defines an upper, a target, and a lower CPU consumption threshold of 90%, 70%, and 45%, respectively. When the current processing load of active threads exceeds the upper threshold, the smallest amount of new threads needed to bring the average processing capacity below the target threshold is provisioned. When the current processing load of active threads is below the bottom threshold, the largest amount of underutilized threads that keep the average processing capacity below the target threshold is decommissioned.

To evaluate the elasticity of the framework, we increase (decrease) the load after filling the window and add (remove) threads while measuring the latency, throughput, and reconfiguration time. For the provisioning experiments, we start with an input rate set to 70% of the maximum rate sustainable by the corresponding number of J^+ threads. After 6 minutes, when the window is full and the system is stable, we increase the rate to 120% of the maximum sustainable rate and therefore trigger an epoch switch. For the decommissioning experiments, we start with 70% of the maximum sustainable rate and, after 6 minutes, decrease the rate to 30%. Table 4 summarizes the various provisioning/decommissioning experiments. For each number of starting threads, it shows the resulting number of threads after provisioning/decommissioning or a "-" when the corresponding number of starting threads does not allow for a provisioning/decommissioning action (e.g., it shows "-" in post-decommissioning for 1 starting thread since no thread can be decommissioned in such a case). The reconfiguration times for each action are shown on the left side of Figure 9. Each reconfiguration time is reported on the X axis based on the number of threads before the reconfiguration (e.g., the provisioning point, shaped as a circle, on 30 threads indicates it took approximately 12 ms to switch from 30 to 52 threads). To stress that each reconfiguration is triggered for a set of threads in which no thread has more spare resources than others (i.e., when the load is balanced), we report on the right-side of Figure 9 the coefficient of variation percentage of threads' workload (in this case one point for each number of starting threads before a provisioning or a decommissioning reconfiguration). All in all, all reconfiguration times are negligible, always lower than 40 ms (an higher reconfiguration time is observed when $\Pi(J^+)$ grows beyond the number of threads of a single socket), and there is at most 2% of load imbalance among threads.

TABLE 4: $\Pi(J^+)$ values for Figure 9

starting $\Pi(J^+)$	1	5	9	18	30	40	60	70
$\Pi(J^+)$ post-provisioning	2	9	16	31	52	69	-	-
$\Pi(J^+)$ post-decommissioning	-	2	3	7	12	17	25	30

Figure 10 shows the effect of increasing or decreasing the workload for *STRETCH* and ScaleJoin, and consecutively provisioning or decommissioning threads for J^+ in *STRETCH*, when $\Pi(J^+)$ is initially set to 18. As shown

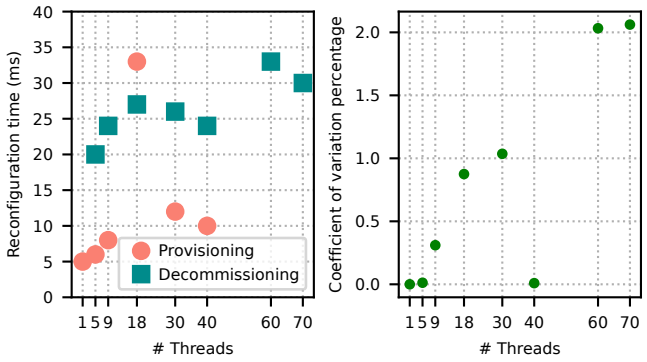


Fig. 9: Reconfiguration times and coefficients of variation for provisioning/decommissioning elastic reconfigurations.

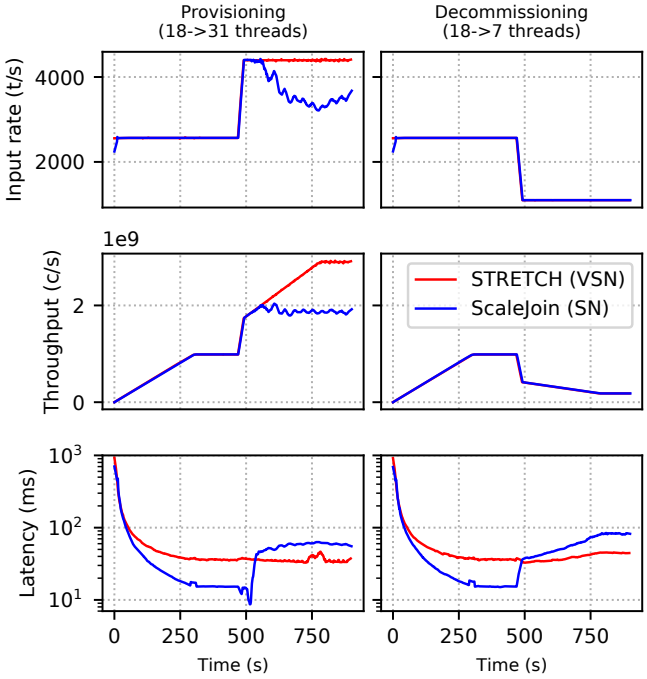


Fig. 10: Input rate, throughput, and latency behavior during one provisioning/decommissioning elastic reconfiguration.

for the provisioning, *STRETCH* sustains higher rates when adding new threads, resulting in higher throughput without affecting the latency. In the decommissioning procedure, when decreasing the workload, *STRETCH* achieves the same throughput as ScaleJoin but shows lower latency, as expected based on the model in [22].

8.5 *STRETCH* under multiple reconfigurations (Q_5)

We continue our evaluation with an experiment that aims at stress-testing *STRETCH*'s reconfigurations. First, we narrow the lower/upper thresholds to $[70\%, 80\%]$ of the capacity. Second, we modify the controller so that J^+ cost is not matched only with its current CPU consumption, but accounts also for the pending and predicted workload, according to the model in [22]. Third, we decrease *WS* to one minute. These changes imply more reconfigurations being frequently triggered (because of the narrower distance of lower/upper thresholds and because the controller proactively triggers reconfigurations on the predicted processing capacity in correspondence to rate changes) and higher sustainable input rates (because of the smaller *WS* and thus resulting comparisons per-tuple across windows).

Each experiment is 20 minutes long and consists of several sequential phases in which data tuples are injected with a constant rate, randomly chosen from the range $[500, 8\,000]$ t/s. The length of each phase is at least 100 and at most 300 seconds (i.e., long enough for J^+'s window instances to reach the size corresponding to the phase's rate). The transition between phases is by an abrupt change in the input rate to stress timely reconfiguration. Figure 11 shows the measurements of a representative execution out of the ones conducted, to keep the illustration uncluttered. Results of additional experiments are shown in Appendix F.

Figure 11(a) shows the input rate throughout the experiment. As shown in Figure 11(b), when the input rate increases, new processing threads are added accordingly. Likewise, in case of a decrease in the rate, unnecessary threads are removed without affecting throughput and latency. J^+'s throughput is shown in Figure 11(c). The highlighted region indicates the upper and lower bound of computational capacity at each moment for the corresponding number of processing threads. When the workload exceeds the upper bound or falls short of the lower bound, the controller proceeds in applying reconfigurations. Figure 11(d) shows the average latency per second. There are a few spikes in the latency which are associated with abrupt changes in the input rate. However, as shown by the box plot on the right side, the controller keeps the average latency around 20 ms. Also, as shown in the zoomed-in view, spikes are handled within 10 seconds.

After showing *STRETCH* results for this experiment, we want to further shed light on the benefits enabled by *STRETCH*'s ultra-fast reconfigurations. Figure 12 shows a zoomed-in view of Figure 11 at the time instants when the input rate increases at 290 seconds. We consider a transition duration starting at the moment the input rate changes until *WS* is exceeded (rectangular highlighted area in Figure 12). Out of the transition duration, we show a baseline by extracting the desired number of processing threads for the specific input rate. Within the transition duration, since *STRETCH* can accommodate reconfigurations in negligible time, the desired number of threads can change gradually, following how the new rate affects the amount of computations until the window instances are filled with tuples coming with the new rate.

8.6 *STRETCH* performance in a real-world setup (Q_6)

In this conclusive experiment, we evaluate *STRETCH* using real-world data. We use NYSE group reference data provided by the NYSE Market Data reporting authority (ftp://ftp.nyxdata.com/), containing six hours of trades registered in the NYSE and NASDAQ markets on July 30^{th}, 2018. One major characteristic of this data is the abrupt and very frequent changes in the incoming rate. This use-case examines correlations of the trades of the 10 biggest companies of the day. When looking into the characteristics of the data stream

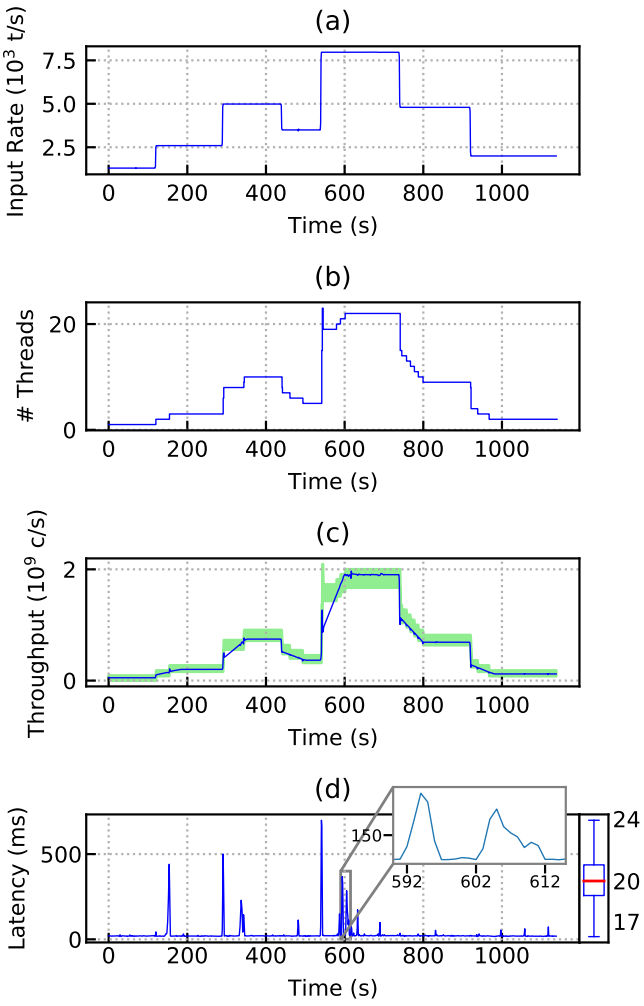


Fig. 11: Results of adjusting the number of processing threads with respect to the input rate for synthetic data.

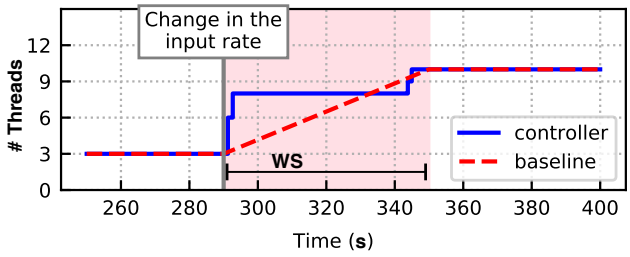


Fig. 12: Zoomed-in view of the reconfigurations around second 290 from Figure 11.

that these trades generate, we see its rate oscillates between 0 and 8 000 t/s. In this experiment, we run J^+ as a self-Join, setting *WS* to 30 seconds and feeding twice a stream with schema $\langle \tau, [\text{id}, \text{TradePrice}, \text{AveragePrice}] \rangle$, where id, of type `string`, is the unique id of the company, TradePrice, of type `int`, is the price of the trade, and AveragePrice, of type `int`, is the average trade price of the corresponding company for the previous day. The output schema S_O is $\langle \tau, [\text{l\_id}, \text{l\_price}, \text{r\_id}, \text{r\_price}] \rangle$, where l_id, l_price, r_id, and r_price are $t_L.\varphi[1]$, $t_L.\varphi[2]$, $t_R.\varphi[1]$ and $t_R.\varphi[2]$, respectively.

For the predicate function, we study the hedging (or negative correlation) of the stock prices among companies by computing the normalized distance of the stock price for each tuple with respect to its average price and then compare it with the other tuples. The normalized distance ND_t of tuple t is computed as $\frac{t.\varphi[2]-t.\varphi[3]}{t.\varphi[3]}$. Consequently, the join results in an output tuple if:

$$t_R.\varphi[1] \neq t_L.\varphi[1] \bigwedge -1.05 \leq \frac{ND_{t_R}}{ND_{t_L}}$$

As shown in Figure 13, the hedge predicate can be run by a small number of threads most of the time, which also emphasizes the need for adjusting processing threads at runtime to utilize the resources. Nonetheless, abrupt changes in the data rate require resource adjustments. The zoomed-in view of Figure 13 highlights the results for the time interval in which the highest peak of the input rate occurs. Figure 13(a) indicates the input rate with the highest peak $\sim$ 7 600 t/s. The corresponding throughput and latency are also shown in Figure 13(c) and Figure 13(d), respectively. As shown in Figure 13(d), *STRETCH* keeps the latency low (on average at 21 ms for the zoomed-in view, as shown by the box plot Figure 13(d), and 1 ms for the whole experiment) by frequently provisioning or decommissioning instances according to changes in the input rate.

9 Related Work

Scalable and elastic stream processing are discussed in several related works (e.g., [4], [23], [24]). For a systematic review, we refer to [25]. *STRETCH* does not focus on a specific stateful operator [26] nor on a specific policy to trigger reconfigurations [16]. Instead, it shows how shared-memory can be seamlessly leveraged to scale up stateful analysis while virtualizing the common APIs for SN parallelism (i.e., without altering how queries are usually composed and still being able to scale applications out via SN parallelism), something not previously discussed in the literature. In the following, we place particular focus on works about scale-up servers and intra-operator elasticity. Acknowledging that intra-operator elasticity is a way to boost performance in scale-up servers, we mostly discuss solutions that are not elastic when covering works dedicated to scale-up servers. For space reasons, we do not discuss elastic approaches not tailored to stream processing (e.g., [27]).

Focusing on scale-up servers, a first distinction can be made between works that aim at better leveraging the memory shared among CPU threads [26], [28], [29] and works that, orthogonally to *STRETCH*, focus on shared-memory in hybrid architectures, with a special focus on integrated [30], [31] and discrete [32] CPU-GPU ones.

For the former, differently from *STRETCH*'s intra-operator memory sharing, [28] focuses on the complementary aspect of inter-operator/intra-query memory sharing, to optimize operator placement for producer-consumer pairs e.g., based on NUMA distance. Complementary intra-query optimizations are also discussed in [29], by formalizing basic stream processing tasks and defining composition/transformation rules that can be used to jointly boost performance metrics such as throughput and latency. An approach closer to *STRETCH* is discussed in [26], with a

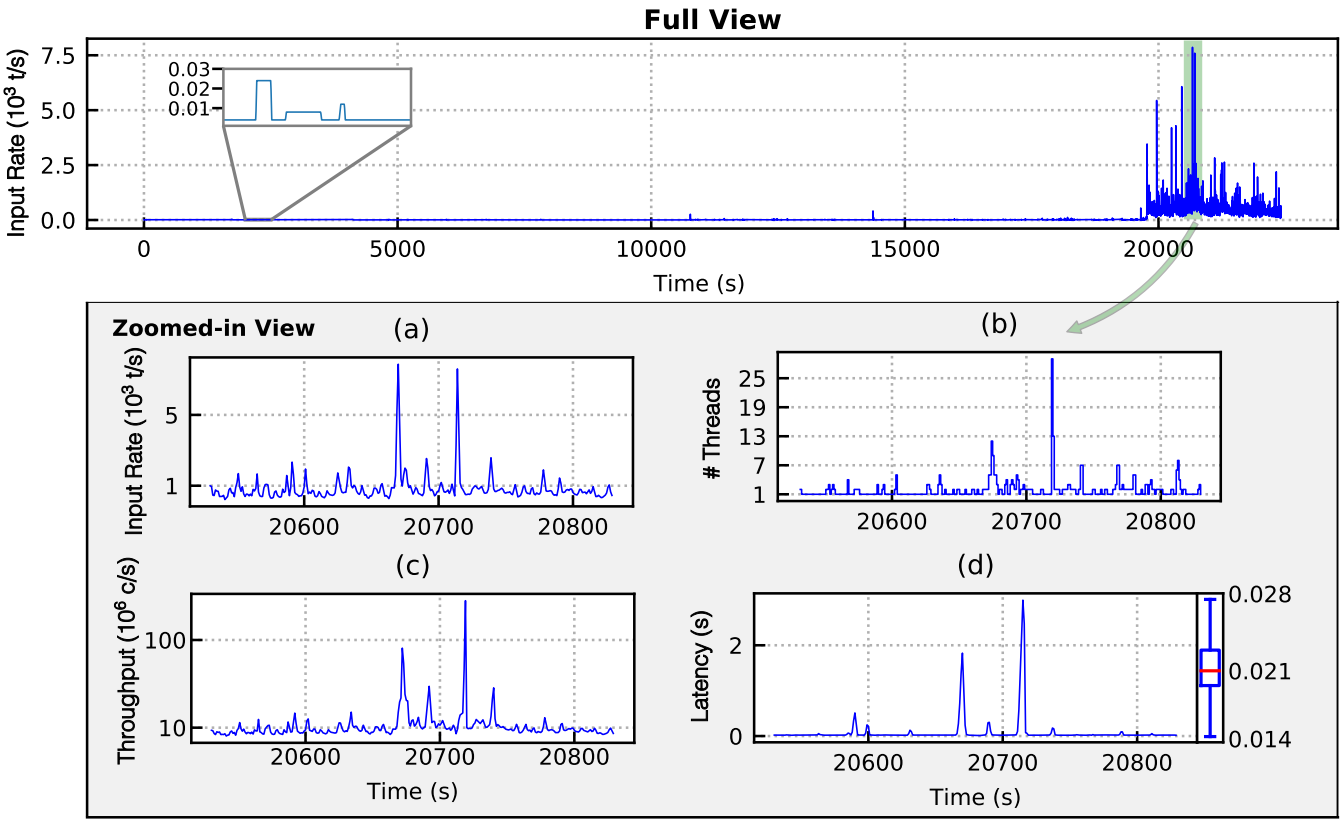


Fig. 13: Results of adjusting the number of processing threads while processing the NYSE market data. The zoomed-in view indicates the details of the experiment during the time when the input rate is the highest.

special focus on streaming aggregation. Such an approach allows trading a higher degree of customization for the richer semantics and elasticity offered by *STRETCH*.

Focusing now on elasticity for intra-operator parallelization, we discuss related work based on goals and properties such as the number of threads that can change per reconfiguration, the roles of state transfer and the triggering mechanism, the support for correctness, and the reaction time vs overhead of frequent reconfigurations.

Regarding changes in the number of threads, related works provide techniques for provisioning/decommissioning one thread at a time (e.g., [33], [34]) or more threads (e.g., [35]), as in our case. Differently from *STRETCH*, nonetheless, in [35] the execution of operators being reconfigured is halted to ensure all required state transfers are completed before the processing of tuples resumes.

In connection to state transfers, a common challenge is that of deciding how to balance the load of a parallel operator, which is usually a combinatorial problem related to packing (cf. [4], [35], [36], [37] and references therein). Some related work proposed techniques to make decisions that can reduce latency spikes [38], recreating small states by replaying past tuples rather than serializing/deserializing states [4], or by distributing the work to nodes through hashing, in ways that minimize the changes when rehashing [39]. Since *STRETCH* allows to re-balance work without any state transfer nor serialization/deserialization, it makes these issues orthogonal and existing techniques complementary.

Regarding correctness, we note that such an aspect has been formalized by e.g., [7], [13] and is also referred to as *determinism* [7], [13], *safety* [40] and *semantic transparency* [4]. Here we show sufficient conditions for correct execution, also under reconfigurations.

Focusing now on the issue of when to trigger reconfigurations, and the related trade-offs between overheads and time to react, related works can be distinguished into reactive or proactive approaches (cf. [4], [16], [17], [34], [41], [42], [43] and references therein). Earlier works proposed reactive strategies, e.g. with threshold-based rules based on CPU utilization [4], [44], [45], heuristic-based algorithms [46], and model-based approaches to enforce latency constraints [47]. To make timely decisions, proactive approaches have gained more attention over the years. Model-based proactive methods such as [16], for instance, use a limited future time horizon to choose reconfigurations for timely execution, with two different resource usage characteristics to achieve high throughput and low latency. Similar to [47], the technique proposed in [16] uses queuing theory to model stream operators as G/G/1 queues. More advanced queuing systems to model SPEs are discussed in [48] to increase the accuracy of the results. However, the technique proposed in [48] does not consider quality of service requirements and it is not optimized for low latency workload. All in all, various complementary triggering mechanisms can be combined with *STRETCH*. As discussed by [16] with respect to SASO properties (i.e., Stability, Accuracy, Settling time, and Overshoot), elasticity builds on top of contrasting objectives. *STRETCH* can imply better margins due to its ultra-fast reconfigurations.

10 Conclusions and Future Work

We introduced *STRETCH*, a model as well as an implemented prototype for VSN parallel and elastic execution of stateful stream processing. We started discussing how SN parallelism can potentially suffer data duplication and state transfer overheads, and how such overheads can be avoided when the instances of a parallel operator share memory. For *STRETCH* approach to be general while encapsulating the semantics of common stateful operators, we have introduced a generalized stateful operator O^+, and later discussed how VSN can be enabled for O^+'s instances by having the latter share their input tuples, output tuples, and state. Together with an algorithmic description, we have provided a fully implemented prototype, which builds on the state-of-the-art stream-processing-tailored data structure ScaleGate. Together with our theoretical contributions, we also provided an exhaustive evaluation, based both on synthetic and real data and comparing with various baselines.

To the best of our knowledge, this is the first work blending a formal approach with correctness proofs about parallel/elastic stateful stream processing with a model and prototype complementary to that of SN-based solutions. *STRETCH* can pave the road for hybrid approaches that extend from intra- to inter-node parallel and elastic solutions.

Acknowledgments

Work supported by the Swedish Foundation for Strategic Research (SSF) – project "FiC" grant GMT14-0032 – and the Swedish Research Council (Vetenskapsrådet) – projects "HARE" grant 2016-03800, and "PSI" grant 2020-05094.

References

[1] H. Najdataei, Y. Nikolakopoulos, M. Papatriantafilou, P. Tsigas, and V. Gulisano, "Stretch: Scalable and elastic deterministic streaming analysis with virtual shared-nothing parallelism," in *Proceedings of the 13th ACM International Conference on Distributed and Event-based Systems*, 2019, pp. 7–18.

[2] "Apache Flink," https://flink.apache.org, accessed:2019-3-1.

[3] "Apache Storm," http://storm.apache.org, accessed:2019-3-1.

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

[5] "Custom state serialization," https://ci.apache.org/projects/flink/flink-docs-release-1.13/docs/dev/datastream/fault-tolerance/custom_serialization/, accessed:2021-9-22.

[6] P. B. Gibbons, "Big data: Scale down, scale up, scale out," in *IEEE International Parallel and Distributed Processing Symposium*, 2015, pp. 3–3.

[7] V. Gulisano, Y. Nikolakopoulos, D. Cederman, M. Papatriantafilou, and P. Tsigas, "Efficient data streaming multiway aggregation through concurrent algorithmic designs and new abstract data types," *ACM Trans. Parallel Comput.*, vol. 4, no. 2, pp. 11:1–11:28, Oct. 2017.

[8] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Wittle, "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.

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

[10] "Apache Beam," https://beam.apache.org/, accessed:2020-11-12.

[11] 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.

[12] T. Akidau, E. Begoli, S. Chernyak, F. Hueske, K. Knight, K. Knowles, D. Mills, and D. Sotolongo, "Watermarks in stream processing systems: Semantics and comparative analysis of apache flink and google cloud dataflow," *Proceedings of the VLDB Endowment*, no. 3, 2020.

[13] V. Gulisano, Y. Nikolakopoulos, M. Papatriantafilou, and P. Tsigas, "ScaleJoin: a deterministic, disjoint-parallel and skew-resilient stream join," *IEEE Trans. Big Data*, vol. 7, no. 2, pp. 299–312, 2021.

[14] N. Zacheilas, V. Kalogeraki, Y. Nikolakopoulos, V. Gulisano, M. Papatriantafilou, and P. Tsigas, "Maximizing determinism in stream processing under latency constraints," in *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*, 2017, pp. 112–123.

[15] I. Walulya, D. Palyvos-Giannas, Y. Nikolakopoulos, V. Gulisano, M. Papatriantafilou, and P. Tsigas, "Viper: A module for communication-layer determinism and scaling in low-latency stream processing," *Future Generation Computer Systems*, vol. 88, pp. 297–308, 2018.

[16] T. De Matteis and G. Mencagli, "Proactive elasticity and energy awareness in data stream processing," *Journal of Systems and Software*, vol. 127, pp. 302–319, 2017.

[17] N. Zacheilas, V. Kalogeraki, N. Zygouras, N. Panagiotou, and D. Gunopulos, "Elastic complex event processing exploiting prediction," in *2015 IEEE International Conference on Big Data (Big Data)*. IEEE, 2015, pp. 213–222.

[18] R. Ananthanarayanan, V. Basker, S. Das, A. Gupta, H. Jiang, T. Qiu, A. Reznichenko, D. Ryabkov, M. Singh, and S. Venkataraman, "Photon: Fault-tolerant and scalable joining of continuous data streams," in *Proceedings of the 2013 ACM SIGMOD international conference on management of data*, 2013, pp. 577–588.

[19] H. Miao, H. Park, M. Jeon, G. Pekhimenko, K. S. McKinley, and F. X. Lin, "{StreamBox}: Modern stream processing on a multicore machine," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, 2017, pp. 617–629.

[20] Z. Qian, Y. He, C. Su, Z. Wu, H. Zhu, T. Zhang, L. Zhou, Y. Yu, and Z. Zhang, "Timestream: Reliable stream computation in the cloud," in *Proceedings of the 8th ACM European Conference on Computer Systems*, 2013, pp. 1–14.

[21] P. Roy, J. Teubner, and R. Gemulla, "Low-latency handshake join," *Proc. VLDB Endowment*, 2014.

[22] V. Gulisano, A. V. Papadopoulos, Y. Nikolakopoulos, M. Papatriantafilou, and P. Tsigas, "Performance modeling of stream joins," in *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*, 2017, pp. 191–202.

[23] R. Mayer, B. Koldehofe, and K. Rothermel, "Predictable low-latency event detection with parallel complex event processing," *IEEE Internet of Things Journal*, vol. 2, no. 4, pp. 274–286, 2015.

[24] L. Mai, K. Zeng, R. Potharaju, L. Xu, S. Suh, S. Venkataraman, P. Costa, T. Kim, S. Muthukrishnan, V. Kuppa *et al.*, "Chi: a scalable and programmable control plane for distributed stream processing systems," *Proc. VLDB Endowment*, vol. 11, no. 10, pp. 1303–1316, 2018.

[25] V. Gulisano, M. Papatriantafilou, and A. V. Papadopoulos, "Elasticity," in *Encyclopedia of Big Data Technologies*, S. Sakr and A. Y. Zomaya, Eds. Cham: Springer Int'l Conf. Publishing, 2018, pp. 1–7.

[26] G. Theodorakis, A. Koliousis, P. Pietzuch, and H. Pirk, "Lightsaber: Efficient window aggregation on multi-core processors," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 2505–2521.

[27] D. De Sensi, M. Torquati, and M. Danelutto, "A reconfiguration algorithm for power-aware parallel applications," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 13, no. 4, pp. 1–25, 2016.

[28] S. Zhang, J. He, A. C. Zhou, and B. He, "Briskstream: Scaling data stream processing on shared-memory multicore architectures," in *Proceedings of the 2019 International Conference on Management of Data*, 2019, pp. 705–722.

[29] G. Mencagli, M. Torquati, A. Cardaci, A. Fais, L. Rinaldi, and M. Danelutto, "Windflow: high-speed continuous stream processing with parallel building blocks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 11, pp. 2748–2763, 2021.

[30] F. Zhang, C. Zhang, L. Yang, S. Zhang, B. He, W. Lu, and X. Du, "Fine-grained multi-query stream processing on integrated architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 9, pp. 2303–2320, 2021.

[31] F. Zhang, L. Yang, S. Zhang, B. He, W. Lu, and X. Du, "Finestream:fine-grained window-based stream processing on cpu-gpu integrated architectures," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020, pp. 633–647.

[32] T. De Matteis, G. Mencagli, D. De Sensi, M. Torquati, and M. Danelutto, "Gasser: An auto-tunable system for general sliding-window streaming operators on gpus," *IEEE Access*, vol. 7, pp. 48 753–48 769, 2019.

[33] S. Schneider, H. Andrade, B. Gedik, A. Biem, and K.-L. Wu, "Elastic scaling of data parallel operators in stream processing," in *2009 IEEE International Symposium on Parallel & Distributed Processing*. IEEE, 2009, pp. 1–12.

[34] C. Hochreiner, M. Vögler, S. Schulte, and S. Dustdar, "Elastic stream processing for the internet of things," in *2016 IEEE 9th international conference on cloud computing (CLOUD)*. IEEE, 2016, pp. 100–107.

[35] T. Heinze, V. Pappalardo, Z. Jerzak, and C. Fetzer, "Auto-scaling techniques for elastic data stream processing," in *Data Engineering Workshops (ICDEW), 2014 IEEE 30th Int'l Conf. on*. IEEE, 2014, pp. 296–302.

[36] C. Balkesen, N. Tatbul, and M. T. Özsu, "Adaptive input admission and management for parallel stream processing," in *Proc. of the 7th ACM Int'l Conf. on Distributed event-based systems*. ACM, 2013, pp. 15–26.

[37] T. Heinze, Y. Ji, Y. Pan, F. J. Grueneberger, Z. Jerzak, and C. Fetzer, "Elastic complex event processing under varying query load," in *BD3@ VLDB*, 2013, pp. 25–30.

[38] T. Heinze, Z. Jerzak, G. Hackenbroich, and C. Fetzer, "Latency-aware elastic scaling for distributed data stream processing systems," in *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, ser. DEBS '14. New York, NY, USA: ACM, 2014, pp. 13–22.

[39] B. Gedik, "Partitioning functions for stateful data parallelism in stream processing," *The VLDB Journal*, vol. 23, no. 4, pp. 517–539, 2014.

[40] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm, "A catalog of stream processing optimizations," *ACM Computing Surveys (CSUR)*, vol. 46, no. 4, pp. 1–34, 2014.

[41] A. Martin, A. Brito, and C. Fetzer, "Scalable and elastic realtime click stream analysis using streammine3g," in *Proc. of the 8th ACM Int'l Conf. on Distributed Event-Based Systems*. ACM, 2014, pp. 198–205.

[42] V. Cardellini, M. Nardelli, and D. Luzi, "Elastic stateful stream processing in storm," in *High Performance Computing & Simulation (HPCS), 2016 Int'l Conf. on*. IEEE, 2016, pp. 583–590.

[43] A. G. Kumbhare, Y. Simmhan, M. Frincu, and V. K. Prasanna, "Reactive resource provisioning heuristics for dynamic dataflows on cloud infrastructure," *IEEE Transactions on Cloud Computing*, vol. 3, no. 2, pp. 105–118, 2015.

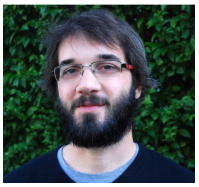
[44] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch, "Integrating scale out and fault tolerance in stream processing using operator state management," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 725–736.

[45] T. Heinze, V. Pappalardo, Z. Jerzak, and C. Fetzer, "Auto-scaling techniques for elastic data stream processing," in *2014 IEEE 30th International Conference on Data Engineering Workshops*. IEEE, 2014, pp. 296–302.

[46] B. Gedik, S. Schneider, M. Hirzel, and K.-L. Wu, "Elastic scaling for data stream processing," *IEEE Trans. Parallel and Distributed Systems*, vol. 25, no. 6, pp. 1447–1463, 2014.

[47] B. Lohrmann, P. Janacik, and O. Kao, "Elastic stream processing with latency guarantees," in *2015 IEEE 35th International Conference on Distributed Computing Systems*. IEEE, 2015, pp. 399–410.

[48] T. Cooper, P. Ezhilchelvan, and I. Mitrani, "A queuing model of a stream-processing server," in *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 2019, pp. 27–35.



Vincenzo Gulisano is an Associate Professor at Chalmers University of Technology. His research focuses on data processing and distributed / parallel / elastic and fault-tolerant data streaming. Dr. Vincenzo Gulisano holds a PhD in Computer Science from the Polytechnic University of Madrid, Spain.

Hannaneh Najdataei received her BSc degree in Software Engineering and her MSc degree in Artificial Intelligence from Shiraz University in Iran. She is currently a PhD student at Chalmers University of Technology. Her research focuses on parallel programming, data stream processing and cyber–physical systems.

Yiannis Nikolakopoulos received the B.Sc. in Mathematics from the National and Kapodistrian University of Athens, Greece and the Ph.D. degree in Computer Science and Engineering from Chalmers University of Technology, Gothenburg, Sweden. Currently, he is a system software engineer at ZeroPoint Technologies AB. His research interests include memory management, memory compression, concurrent data structures and stream processing on multicore and many-core systems.

Alessandro V. Papadopoulos (Senior Member, IEEE) received the B.Sc. and M.Sc. (summa cum laude) degrees in computer engineering and the Ph.D. degree (Hons.) in information technology from the Politecnico di Milano, Italy, in 2008, 2010, and 2013, respectively. He is an Associate Professor of Computer Science with Mälardalen University, Västerås, Sweden. His research interests include robotics, control theory, real-time systems, autonomic computing.

Marina Papatriantafilou is Associate Professor at Chalmers University of Technology. Earlier, she was with the Max-Planck Institute for Computer Science, Saarbruecken and CWI, Amsterdam. She received her PhD degree from the Computer Science and Informatics Dept., Patras University and is a member of Network of National Contacts ACM-WE NeNaC. Her research interests include: efficient parallel, distributed stream processing and applications in multiprocessor, multicore and distributed, cyber-physical systems; synchronization, consistency, fault-tolerance.

Philippas Tsigas received the B.Sc. degree in mathematics and the Ph.D. degree in computer engineering and informatics from the University of Patras, Greece. He was at the National Research Institute for Mathematics and Computer Science, Amsterdam, The Netherlands (CWI), and at the Max-Planck Institute for Computer Science, Saarbrucken, Germany, before. At present, he is a professor at Chalmers University of Technology, Sweden. His research interests include concurrent data structures and algorithmic libraries for multiprocessor and many-core systems, communication and synchronization in parallel systems, power aware computing, fault-tolerant computing, autonomic computing, scalable data streaming.

APPENDIX A
TABLE OF SYMBOLS (IN ALPHABETICAL ORDER)

Symbol	Description
$\langle \tau, \ldots, [\varphi[1], \varphi[2], \ldots] \rangle$	Combined notation for a tuple
$A(WA, WS, 1, f_{SK}, WT, S, f_A, f_R)$	Combined notation for Aggregate operator
$A^+(WA, WS, 1, f_{MK}, WT, S, f_A, f_R)$	Combined notation for Aggregate operator that relies on f_{MK} instead of f_{SK}
γ	Event time shared by O^+ instances at which a reconfiguration takes place
δ	Smallest increment of event time of an SPE
D	Generic downstream operator/egress of O/O^+
DAG	Directed Acyclic Graph
e	epoch during the execution of a certain operator
f_A	Aggregate function of A
f_J	Join function of J
f_μ	Function mapping keys to operators instances
f_{MK}	Multi Key-by function that maps each tuple t to an arbitrary number of key values
f_O	Function used by O^+ to produce the result from a window instance
f_R	Reduce function of A
f_S	Function used by O^+ to shift a window instance
f_{SK}	Single Key-by function that maps each tuple t to exactly one key value
f_U	Function used by O^+ to update a window instance upon reception of t
`forward/forwardSN/forwardVSN`	Methods used by an operator/ingress instance to route tuples downstream (for O, O^+ in SN, and O^+ in VSN setups, respectively)
$J(WA, WS, 2, f_{SK}, WT, S, f_J)$	Combined notation for Join operator
$J^+(WA, WS, 2, f_{MK}, WT, S, f_J)$	Combined notation for Aggregate operator that relies on f_{MK} instead of f_{SK}
M	Map/FlatMap operator
$O(WA, WS, I, f_{SK}, WT, S, f_1, f_2, \ldots)$	Combined notation for a generic stateful operator. Parameters *WA*, *WS*, I, f_{SK}, *WT*, and S are covered in the table. Parameters f_1, f_2 refer to operator-specific functions to update O's state and produce output tuples
o_j	j-th instance of O
$O^+(WA, WS, I, f_{MK}, f_\mu, WT, S, f_U, f_O, f_S)$	Combined notation for *STRETCH*'s generalized operator
o_j^+	j-th instance of O^+
$\mathbb{O}$	Set of parallel instances running for a certain during epoch e
$\Pi(X)$	Parallelism degree of ingress/operator/egress X
`process/processSN/processVSN`	Methods used by an operator instance to process each new input tuple (for O, O^+ in SN, and O^+ in VSN setups, respectively)
$q_{a,b}$	Queue connecting operator instances (or ingress/egress) a and b
S	Schema of the tuples of the referenced stream
σ_j	State of o_j/o_j^+ (when locally maintained at o_j/o_j^+
σ	State of o_j^+ (when shared by all O^+ instances
SN	Shared Nothing
SPE	Stream Processing Engine
$t.\tau$, $t.\varphi$	Timestamp and payload (ℓ-th sub-attribute is $t.\varphi[\ell]$) of tuple t
U_i	The i-th generic upstream operator/ingress of O/O^+
$u_{i,j}$	The j-th instance of U_i
VSN	Virtual Shared Nothing
$w = \langle \zeta, l, k \rangle$	Combined notation for a window instance w: state ζ, left inclusive boundary l, and key k.
$W^\omega_{o_j}$	Watermark of the j-th instance of operator O at clock wall time ω
WA	Window advance
WS	Window size
WT	Window Type maintained by a stateful operator (single or multi)
$TB/TB_{in}/TB_{out}$ (Tuple Buffer)	Data object *STRETCH* uses to connect U_i with O^+ and O^+ with D (cf. Table 2)

APPENDIX B
WATERMARK EXAMPLE

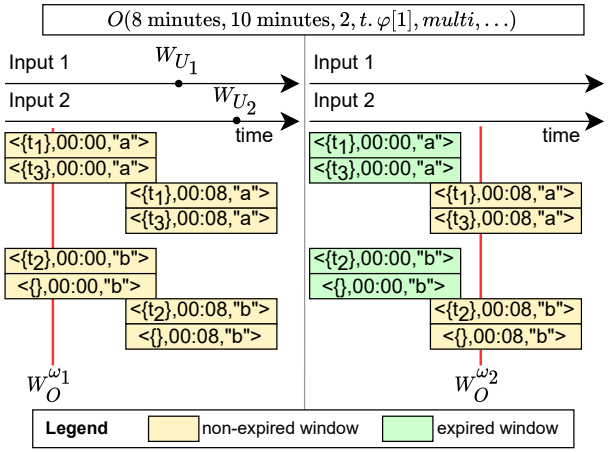


Fig. 14: Internal state of operator O before (left side of the figure) and after (right side of the figure) updating its watermark from $W_O^{\omega_1}$ to $W_O^{\omega_2}$.

Figure 14 shows how the internal state of Operator O from Figure 1 changes from wall-clock time ω_1 to ω_2, upon the processing of two tuples, one from each upstream peer, that carry new watermark values. As shown, none of the window instances maintained by O before updating its watermark (left side of Figure 14) is expired, since incoming tuples, with a timestamp greater than $W_O^{\omega_1}$, could still contribute to any of such window instances. Once O's watermark changes to $W_O^{\omega_2}$, no new input tuple can contribute to window instances starting at 00:00. Hence, such windows are expired and their corresponding output tuple can be created and forwarded to O's downstream peer(s).

APPENDIX C
EXAMPLE FOR COROLLARY 1

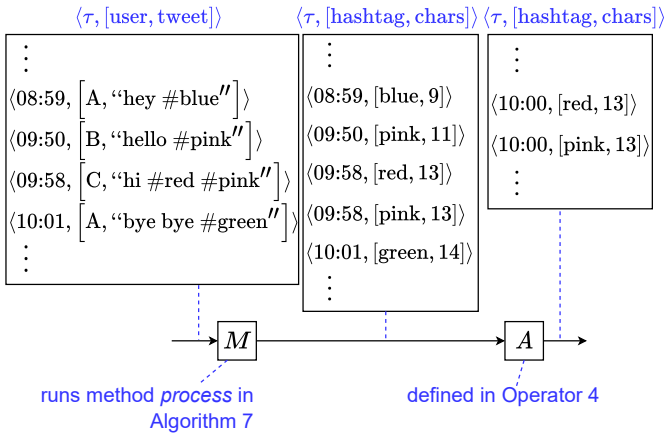


Fig. 15: Sample execution of a M and an A that, together, implement the semantics of the running example from § 1

Figure 15 illustrates how a M and an A operator can be jointly used to implement the semantics of an A^+, according to Corollary 1. For each stream, the figure shows an excerpt of the tuples observed as input/output of the two operators. M implements the `process` function presented in Alg. 7, creating and forwarding one output tuple for each hashtag found in an input tuple (Alg. 7 L2). For instance, forwarding tuples $\langle 09{:}58, [\text{red}, 13]\rangle$ and $\langle 09{:}58, [\text{pink}, 13]\rangle$ upon processing of $\langle 09{:}58, [\text{C}, \text{"hi \#red \#pink"}]\rangle$.

A is defined according to Operator 1. A aggregates tuples over a sliding window with *WA* and *WS* of 30 and 60 minutes, respectively. According to f_{SK}, tuples sharing the same hashtag are aggregated together, and routed to the same instance of A by f_μ (cf. § 2.2). In the example, output tuples $\langle 10{:}00, [\text{red}, 13]\rangle$ and $\langle 10{:}00, [\text{pink}, 13]\rangle$ are the results of aggregating the input tuples covering the event time $[09{:}00, 10{:}00)$, $\langle 09{:}50, [\text{pink}, 11]\rangle$ and $\langle 09{:}58, [\text{pink}, 13]\rangle$ for key *pink*, and $\langle 09{:}58, [\text{red}, 13]\rangle$ for key *red*.

Algorithm 7: Method `process` for the M used in Figure 15 to compute the longest tweet on a per-hashtag basis.

```
1 Function process(t)
2     for h ∈ hashtags(t.φ[2]) do
        forward(⟨t.τ, [h, length(t.φ[2])]⟩)
```

Operator 1: User-defined parameters for the A used in Figure 15 to compute the longest tweet on a per-hashtag basis.

$\text{A}(30m, 60m, 1, f_{SK}, \text{multi}, f_\mu, S_O, f_A, f_R)$, where:

```
1  Function f_SK(t)
2      return t.φ[1]
3  Function f_μ(k)
4      return hash(k) %Π(A)
5  Class ζ
6      long count
7  Function f_A(w, t) // Update count based on t.φ[2]
8      if t.φ[2] > w.ζ.count then
9          w.ζ.count ← t.φ[2]
10     return {w.ζ}
11 Function f_R(w, t)
12     return {w.k, w.ζ.count}
```

APPENDIX D
EXAMPLE OPERATORS

This appendix provides several complete examples. It begins by providing one example for the A^+ and one example for the J^+ operators introduced in § 4. The latter operator is one of the operators used to evaluate *STRETCH* in § 8. The remaining examples are for other operators that are also used in § 8.

A^+ – *compute longest tweet per hashtag*

This A^+ operator is the running example from § 1. We assume that the input schema is defined as $\langle \tau, [\text{user}, \text{tweet}]\rangle$, while the output schema S_O as $\langle \tau, [\text{hashtag}, \text{chars}]\rangle$, where *hashtag* and *chars* represent the number of characters *chars* of the longest tweet observed for each *hashtag*. As shown in Operator 2, f_{MK} returns one key for each hashtag contained in the *tweet* attribute of t (L3). Each hashtag is then matched with one instance by f_μ, hashing the key and returning its

Operator 2: User-defined parameters (cf. § 4) for an A^+ computing the longest tweet on a per-hashtag basis.

$A^+(WA, WS, 1, f_{MK}, \text{multi}, S_O, f_\mu, f_U, f_O, -)$, where:

```
1  Function f_MK(t)
2  |  K ← {}
3  |  for h ∈ hashtags(t.φ[2]) do K ← K ∪ h
4  |  return K
5  Function f_μ(k)
6  |  return hash(k) %Π(A+)
7  Class ζ
8  |  long count
9  Function f_U(w, t) // Update count based on t.φ[2]
10 |  if length(t.φ[2]) > w.ζ.count then
11 |  |  w.ζ.count←length(t.φ[2])
12 |  return {w.ζ}
13 Function f_O(w, t)
14 |  return {w.k, w.ζ.count}
```

value modulo the parallelism degree of A^+. The state associated with each window instance w is a counter. Function f_U is invoked to update such count, while f_O is invoked to forward the hashtag (k) and count of each expired window instance. Since WT = multi, each expired window instance is simply removed after being passed as parameter to f_O.

Operator 3: User-defined parameters (cf. § 4) for a J^+ implementing ScaleJoin [13].

$J^+(WA, WS, 2, f_{MK}, \text{single}, S_O, f_\mu, f_U, -, -)$, where:

```
1  Function f_MK(t)
2  |  return {1, ..., 1000}
3  Function f_μ(k)
4  |  return hash(k)%Π(J+)
5  Class ζ
6  |  long c // tuple counter
7  |  queue<t> T // previous tuples
8  Function f_U({w1, w2}, t)
9  |  T_O ← {} // create set for output tuples
10 |  w1.ζ.c ← w1.ζ.c + 1 // increase c of win inst.
11 |  w2.ζ.c ← w2.ζ.c + 1
12 |  if t from U1 then // set this/opposite win.
   |   inst.
13 |  |  this_w ← w1
14 |  |  opp_w ← w2
15 |  else
16 |  |  this_w ← w2
17 |  |  opp_w ← w1
18 |  while opp_w.ζ.T[0].τ + WS < t.τ do // purge
19 |  |  opp_w.ζ.T.dequeue()
20 |  for t' ∈ opp_w.ζ.T do // match
21 |  |  if t' and t match then T_O ← T_O ∪ {t'.φ, t.φ}
22 |  if c%1000 =this_w.k then // store t
23 |  |  this_w.ζ.T.enqueue(t)
24 |  return T_O // return results
```

Algorithm 8: Method `process` for the M used in § 8.1 in Flink's `wordcount` implementation.

```
1 Function process(t)
2 |  for w ∈split(t.φ[2]) do forward(⟨t.τ, [w]⟩)
```

Operator 4: User-defined parameters for the A used in § 8.1 in Flink's `wordcount`/`paircount` implementation.

$A(60s, 120s, 1, f_{SK}, \text{multi}, f_\mu, S_O, f_A, f_R)$, where:

```
1  Function f_SK(t) // implementation for wordcount
2  |  return t.φ[1]
3  Function f_SK(t) // implementation for paircount
4  |  return ⟨t.φ[1], t.φ[2]⟩
5  Function f_μ(k)
6  |  return hash(k) %Π(A)
7  Class ζ
8  |  long count
9  Function f_A(w, t)
10 |  w.ζ.count← w.ζ.count+1
11 |  return {w.ζ}
12 Function f_R(w, t)
13 |  return {w.k, w.ζ.count}
```

Algorithm 9: Method `process` for the M used in § 8.1 in Flink's `paircount` implementation.

```
   Instance-local variables:
1  B // Maximum dist. for 2 words to form a pair
2  Function process(t)
3  |  W ←split(t.φ[2])
4  |  for i ∈ 0, ..., |W| − 1 do
5  |  |  for j ∈ i + 1, ..., |W| − 1 do
6  |  |  |  if j − i ≤ B then forward(⟨t.τ, [W[i], W[j]]⟩)
```

J^+ – ScaleJoin

To provide an example of a J^+, we now show how O^+ can be used to implement ScaleJoin [13], which performs a Cartesian join of all tuples belonging to two windows.

To run in a parallel and skew-resilient fashion, it delivers each input tuple (from any of the two input streams) to all instances, and has each instance compare it with a share of previous tuples (from any of the two input streams). Each tuple is stored, in a round-robin fashion, by exactly one of the instances. We assume in this case that S_O is the concatenation of schemas from the two input streams.

As shown in Operator 3, this implementation maintains a fixed number of keys, larger than $\Pi(J^+)$ (1000 in the example). All keys are returned for each tuple by f_{MK} (L2). Hence, each instance will be given the chance of running f_U for their share of keys (L4). The state associated with each window instance consists of counter c and a `queue` of tuples (L6-7). Since $I = 2$, function f_U is invoked passing two window instances together with t. The counter of each pair of window instances (for all keys) is consistently increased by one for each window instance upon reception of the same tuple for all j_i^+, because ESG_{out} delivers all tuples in the same order to all instances. Afterward, t is used to purge all stale tuples from the window instance opposite to t's stream and subsequently matched with all remaining tuples in such window instance. Finally, t is stored, in a round-robin fashion based on c, in the window instance associated with exactly one key by one instance (L12-24).

Other evaluation operators

This section covers additional examples for other operators used in § 8. Algorithm 8 and Operator 4 refer to the `process` method of M and the implementation of A in Flink, respectively, for the `wordcount` experiment in § 8.2, while Algorithm 9 and Operator 4 are the ones used in Flink for M and A, respectively, for the `paircount` experiment in § 8.2. In this case, we re-use the same Operator and specialize function f_{SK} for compact notation. Operator 5 is the

A^+ used in *STRETCH* for the `wordcount` and `paircount` experiments in § 8.2. Also in this case, we re-use the same Operator and specialize f_{MK} for compact notation. Finally, Operator 6 is used to measure the maximum throughput/minimum latency when the performance bottleneck is given by data sharing/sorting (§ 8.2).

Operator 5: User-defined parameters for the A^+ used in § 8.1 for *STRETCH*'s `wordcount`/`paircount` implementation.

$A^+(WA, WS, 1, f_{MK}, \text{multi}, S_O, f_\mu, f_U, f_O, -)$, where:

```
1  Function f_MK(t) // implementation for wordcount
2  |  K ← {}
3  |  for w ∈ split(t.φ[2]) do K ← K ∪ w
4  |  return K
5  Function f_MK(t) // implementation for paircount.
   Parameter B (the max. dist. for 2 words to
   form a pair) is a local variable of f_MK.
6  |  K ← {}
7  |  W ← split(t.φ[2])
8  |  for i ∈ 0, ..., |W| − 1 do
9  |  |  for j ∈ i + 1, ..., |W| − 1 do
10 |  |  |  if j − i ≤ B then K ← K ∪ ⟨W[i], W[j]⟩
11 |  |  return K
12 Function f_μ(k)
13 |  return hash(k) %Π(A+)
14 Class ζ
15 |  long count
16 Function f_U(w, t)
17 |  w.ζ.count ← w.ζ.count+1
18 |  return {w.ζ}
19 Function f_O(w, t)
20 |  return {w.k, w.ζ.count}
```

Operator 6: O^+ for Q_2 and $\Pi(O^+) = n$.

$O^+(\delta, \delta, 2, f_{MK}, \text{single}, S_O, f_\mu, f_U, -, -)$, where:

```
1  Function f_MK(t)
2  |  return {1, ..., n}
3  Function f_μ(k)
4  |  return k
5  Function f_U({w1, w2}, t)
6  |  return {∅, ∅, t.φ} // return empty states for w1
   |     and w2 and t's payload
```

Appendix E
Example for Theorem 2

In here, we continue the example introduced in Appendix C, focusing on the execution path (in terms of invoked functions) that is triggered for the A^+ of Appendix D (which implements the same semantics of the M and A from Figure 15) upon the reception of tuple $t = \langle$09:58, [C, "hi #red #pink"]$\rangle$. We assume in this case that the *WA* and *WS* parameters for A^+ are set to 30 minutes and 1 hour, respectively. As we show next, the events triggered on O^+'s state upon reception of t are the same triggered for A in Figure 15 upon the reception of the two tuples produced by M from t, i.e. $\langle$09:58, [red, 13]$\rangle$ and $\langle$09:58, [pink, 13]$\rangle$.

For simplicity, we assume all the input tuples' timestamps from Figure 15 are valid watermarks for A^+ (as stated in § 3 such information can be carried in tuples' metadata). Hence, given the *WA*/*WS* parameters of A/A^+,

Execution Trace 1: Execution trace for the A^+ in Appendix D, upon invocation of `processSN`(t = $\langle$09:58, [C, "hi #red #pink"]$\rangle$)

Initial State:
W = 09:00
$\sigma = \{\langle 11, 09{:}00, \text{"pink"}\rangle, \langle 11, 09{:}30, \text{"pink"}\rangle\}$

```
1  updateW(t) // W ← 09:58
2  handleInputTuple(t)
3  earliestWinL(t) // τ1 ← 09:00
4  latestWinL(t) // τ2 ← 09:30
5  σj.check&Create("red", 09:00)
   // σ ← {⟨11, 09:00, "pink"⟩, ⟨0, 09:00, "red"⟩,
   // ⟨11, 09:30, "pink"⟩}
6  σj.set("red", 0, {13})
   // σ ← {⟨11, 09:00, "pink"⟩, ⟨13, 09:00, "red"⟩,
   // ⟨11, 09:30, "pink"⟩}
7  σj.check&Create("red", 09:30)
   // σ ← {⟨11, 09:00, "pink"⟩, ⟨13, 09:00, "red"⟩,
   // ⟨11, 09:30, "pink"⟩, ⟨0, 09:30, "red"⟩}
8  σj.set("red", 1, {13})
   // σ ← {⟨11, 09:00, "pink"⟩, ⟨13, 09:00, "red"⟩,
   // ⟨11, 09:30, "pink"⟩, ⟨13, 09:30, "red"⟩}
9  σj.check&Create("pink", 09:00)
10 σj.set("pink", 0, {13})
   // σ ← {⟨13, 09:00, "pink"⟩, ⟨13, 09:00, "red"⟩,
   // ⟨11, 09:30, "pink"⟩, ⟨13, 09:30, "red"⟩}
11 σj.check&Create("pink", 09:30)
12 σj.set("pink", 1, {13})
   // σ ← {⟨13, 09:00, "pink"⟩, ⟨13, 09:00, "red"⟩,
   // ⟨13, 09:30, "pink"⟩, ⟨13, 09:30, "red"⟩}
```

and since the tuple previously processed by A/A^+ was $\langle$09:50, [B, "hello #pink"]$\rangle$, only window instances such that $l \geq 09:00$ are maintained by A^+ upon reception of t.

The initial state and the execution path following the invocation of `processSN` for tuple t are shown in Listing 1. The first invoked method is `update`W, and W accordingly updated to 09:58 (L1). Since WT = multi, τ_1 and τ_2 are set to 09:00 and 09:30 within the execution of method `handleInputTuple` (L2-4). Subsequently, all window instances for key "red" and "pink", and for l = 09:00 and l = 09:30, are updated (L5-12). Since the window instances for key "pink" are already in place before the processing of t, only the counter maintained in their ζ is updated. As shown in L12, the final state maintained by A^+, contains the window instances that will then result in the production of tuples $\langle$10:00, [red, 13]$\rangle$ and $\langle$10:00, [pink, 13]$\rangle$ (Figure 15) once their timestamp is set to $l + WS$ (i.e., to 10:00).

Appendix F
Additional Experiments

This section contains additional experiments, similar to the one presented in § 8.5. These 20 minutes long experiments stress-test *STRETCH*'s reconfigurations with several phases in which the input rate is subject to abrupt changes.

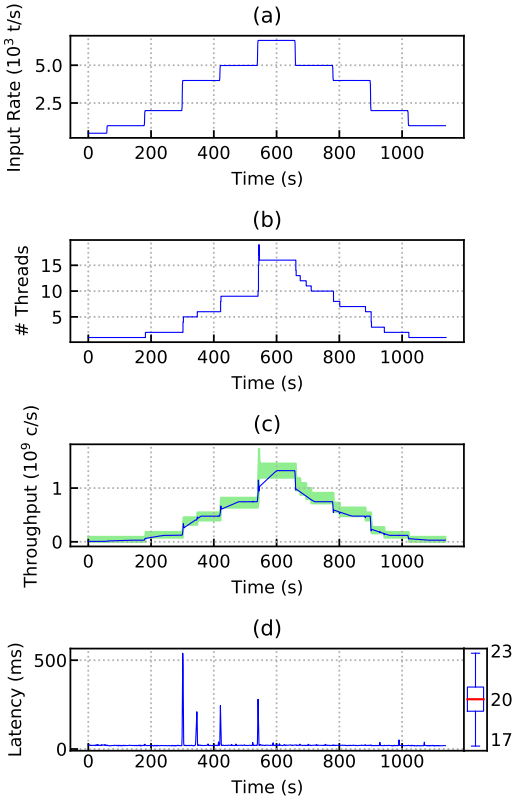


Fig. 16: Results of adjusting the number of processing threads with respect to the input rate for synthetic data.

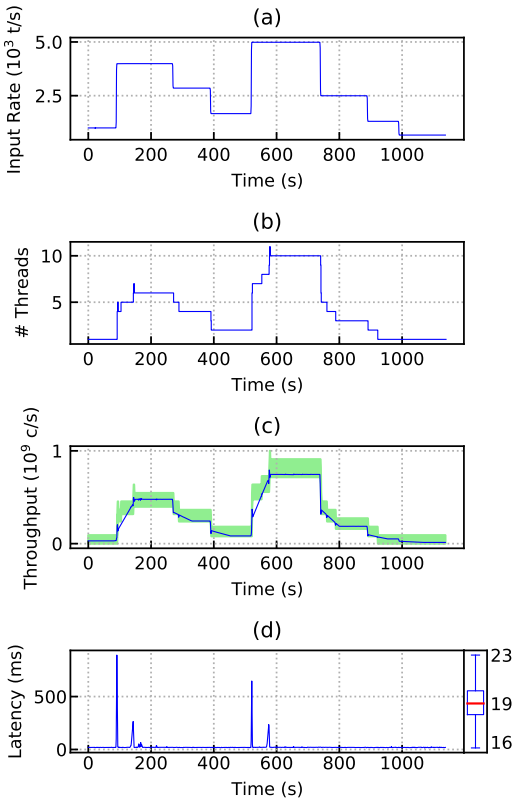


Fig. 17: Results of adjusting the number of processing threads with respect to the input rate for synthetic data.

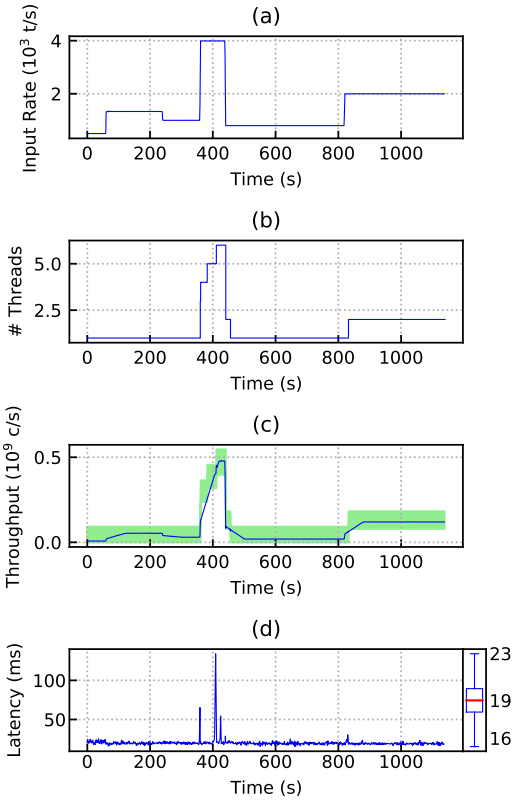


Fig. 18: Results of adjusting the number of processing threads with respect to the input rate for synthetic data.

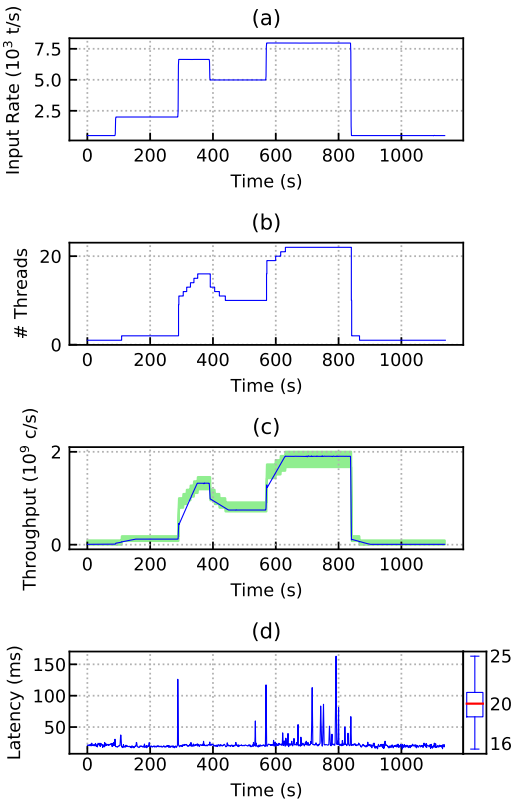


Fig. 19: Results of adjusting the number of processing threads with respect to the input rate for synthetic data.