STRETCH: Scalable and Elastic Deterministic Streaming Analysis with Virtual Shared-Nothing Parallelism

Hannaneh Najdataei, Yiannis Nikolakopoulos
Marina Papatriantafilou, Philippas Tsigas, Vincenzo Gulisano
Chalmers University of Technology
{hannajd,ioaniko,ptrianta,tsigas,vinmas}@chalmers.se

ABSTRACT

Despite the established scientific knowledge on efficient parallel and elastic data stream processing, it is challenging to combine generality and high level of abstraction (targeting ease of use) with fine-grained processing aspects (targeting efficiency) in stream processing frameworks. Towards this goal, we propose *STRETCH*, a framework that aims at guaranteeing (i) high efficiency in throughput and latency of stateful analysis and (ii) fast elastic reconfigurations (without requiring state transfer) for intra-node streaming applications. To achieve these, we introduce *virtual shared-nothing parallelization* and propose a scheme to implement it in *STRETCH*, enabling users to leverage parallelization techniques while also taking advantage of shared-memory synchronization, which has been proven to boost the scaling-up of streaming applications while supporting determinism. We provide a fully-implemented prototype and, together with a thorough evaluation, correctness proofs for its underlying claims supporting determinism and a model (also validated empirically) of virtual shared-nothing and pure shared-nothing scalability behavior. As we show, *STRETCH* can match the throughput and latency figures of the front of state-of-the-art solutions, while also achieving fast elastic reconfigurations (taking only a few milliseconds).

CCS CONCEPTS

• **Information systems → Stream management**; **Data streams**; **Online analytical processing engines**;

KEYWORDS

Data streaming, Shared-nothing parallelism, Elasticity, Scalability

ACM Reference Format:
Hannaneh Najdataei, Yiannis Nikolakopoulos Marina Papatriantafilou, Philippas Tsigas, Vincenzo Gulisano. 2019. STRETCH: Scalable and Elastic Deterministic Streaming Analysis with Virtual Shared-Nothing Parallelism. In *DEBS '19: The 13th ACM International Conference on Distributed and Event-based Systems (DEBS '19), June 24–28, 2019, Darmstadt, Germany*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3328905.3329509

DEBS '19, June 24–28, 2019, Darmstadt, Germany

ACM ISBN 978-1-4503-6794-3/19/06…$15.00
https://doi.org/10.1145/3328905.3329509

1 INTRODUCTION

Data streaming builds on efficient one-pass analysis of unbounded *streams* of *tuples*. It is widely adopted thanks to two decades of research results and thanks to open-source Stream Processing Engines (SPEs) [7, 10, 27, 28]. When such analysis is *stateful*, its resulting output tuples can depend on arbitrarily-long portions of the input.

In the literature, many solutions study how to deploy stateful analysis and efficiently leverage multi-core architectures by means of parallelism and elasticity (i.e., by multi-threaded executions in which resources as threads are adjusted over time) [4, 5, 9–12, 25, 29]. Such techniques focus on optimizing the parallelization of specific operators [9, 11, 12, 25, 29], managing the distributed state of parallel operators [4, 5], providing operator-oblivious parallelization and elasticity [5, 10] or guaranteeing deterministic execution [8, 12, 30], among other aspects.

Challenges. Existing SPEs provide good support for parallelization and elasticity of simple stateful analysis (e.g., continuous per-key summation), but leave to users how to use their APIs efficiently when programming complex stateful analysis. Consider, as a motivating example, a user trying to parallelize the analysis of an application that tries to find, on a per-hashtag basis, how many times an ordered sequence of words is found within or across consecutive tweets. Intuitively, the user could parallelize the application by assigning the analysis of tweets carrying different hashtags to distinct threads. `Key-by` partitioning, provided by Apache Flink [7] (or simply Flink) and Apache Storm [28], could not be directly leveraged, though, for tweets carrying two or more hashtags assigned to different threads. Users would need to either create single-hashtag copies of tweets if the latter carry multiple hashtags (and then use key-by partitioning) or define a custom tuple-to-thread assignment scheme. In both cases, data duplication would unnecessarily hamper performance if the threads share memory and could instead access the same copy of each tweet. To complicate the matter further, if the SPEs arbitrarily interleave tuples forwarded to a parallel thread from multiple streams, the user would also need to program how to deterministically merge such streams (e.g., sort them on a per-tuple basis [12] or with watermarks [1]), in order to prevent the arbitrary interleaving from affecting whether the ordered sequence of words is found or not. Finally, the user might also need to program how to serialize/deserialize the state of the analysis run in parallel for the SPE to trigger reconfigurations (e.g., provisioning or decommissioning of threads).

Addressing efficiently these connected challenges, that hold also for other stateful streaming operators such as joins, is not trivial for the average programmer. With our work, we aim at advancing the front of SPEs' automation and tools available to end users.

Contributions. These challenges have conflicting needs wrt "right amount of sharing" to (i) enhance independence among threads, as well as to (ii) enable efficient coordination for consistent redistribution of work when needed, while (iii) supporting determinism. The implied trade-offs relate to the efficiency in synchronization and state sharing among threads: on one hand, shared-nothing processing maximizes parallelism, but is costly in reconfigurations and in making copies of the data when needed; on the other hand, sharing processing state might introduce contention in maintaining parallelism, but facilitates the workload shifting among threads by not needing state transfer protocols. Based on this, our *motivating research question* is: *Can an intra-node streaming framework take the best of two worlds, shared-nothing and shared-memory parallelism, (i) allowing users to program parallel and elastic stateful operators, (ii) without partitioning but rather sharing input tuples with all threads and specify which ones the latter should process or ignore, while (iii) supporting deterministic execution and (iv) ensuring high efficiency in terms of throughput, latency and reconfiguration times?*

We propose *virtual shared-nothing* parallelisation and provide a framework leveraging it. The framework, called *STRETCH*, manages efficient processing for interconnected streaming operators, supporting determinism even with varying degree of parallelism. In more detail, the contributions include (i) a generalization of previous results (e.g., [12, 13]) in supporting efficient sharing and synchronization among parallel threads, building on ScaleGate, an established object for communication among operators in SPEs, which has been shown to facilitate efficient deterministic fine-grained stream processing; (ii) a novel, virtual shared-nothing state manager that provides to each thread exclusive access to a portion of an operator state while also allowing to efficiently change ownership of such portions at runtime for elastic reconfigurations; (iii) an extended API for ScaleGate, which we call *ESG* (elastic ScaleGate) and the algorithmic implementation for it, to allow dynamic number of threads accessing it, as well as a protocol describing the interactions between ScaleGate objects and the state manager for load balancing, thread provisioning and thread decommissioning; (iv) correctness proofs for the determinism guarantees of the methods and a model for the performance of virtual shared-nothing parallelism; and (v) an extensive evaluation, empirically validating our model, as well as showing that with the proposed methods, fast work redistribution is possible, with minimal overhead both in latency and throughput.

The paper is organized as follows: § 2 introduces preliminary concepts, § 3 presents our system model and objectives. We outline *STRETCH* in § 4 and provide algorithmic implementation details and correctness arguments in § 5, § 6 and § 7. We model the behavior of virtual shared-nothing parallelism comparing it also with shared-nothing parallelism in § 8 and evaluate the model and *STRETCH* in § 9. We discuss related work in § 10 and conclude in § 11.

2 PRELIMINARIES

Data streaming. A data streaming *continuous query* (or simply query in the remainder) is a directed acyclic graph (DAG) of *streams* (carrying information) and *operators* (manipulating such information). A *stream* is an unbounded sequence of tuples sharing the same schema composed by attributes $\langle ts, A_1, \ldots, A_n \rangle$. Attribute $t.ts$ represents the (*event*) time at which the tuple has been created. An *operator* is the minimum processing unit that defines at least one input stream (delivering the tuples to be processed) and one output stream (to forward the output tuples it produces).

Tuples sharing the same schema and being input to the same operator can come from multiple sources or operators. Hence we distinguish between *physical* and *logical* streams. The former represents one stream between a pair of operators while the second represents the set of streams defining the same schema and carrying the same type of information to the same operator. Following a common assumption in the streaming literature (e.g., [2, 10–12]), we assume that each physical stream contains timestamp-sorted tuples. If this is not the case, sorting tools such as [20] can be leveraged. We use the term stream without specifying whether it is logical or physical if it can be deduced by the context.

Streaming operators are distinguished into *stateless* and *stateful*. Such a stateful operator, which we also use later in the paper, is the *Join* [12]: it defines a left (L) and a right input stream (R) and produces output tuples combining the attributes of tuples $t_L \in L$ and $t_R \in R$, for each pair of tuples $\langle t_L, t_R \rangle$ satisfying a given predicate and being closer in time than a given window size WS (i.e., $|t_L.ts - t_R.ts| \leq \text{WS}$). We assume that the timestamp of an output tuple t_o produced by a stateful operator is equal to that of the latest processed tuple that triggers the output of t_o (additional timestamps related to the tuples contributing to t_o can be other attributes of t_o's schema).

Determinism. Deterministic execution of a *sequential* operator requires that each processing step depends on the notion of event time carried by the tuples themselves (attribute ts) and is affected neither by the latency incurred in transmitting tuples from an operator to another operator nor by the interleaving of tuples to an operator with multiple input streams. For a *parallel* operator, determinism is enforced when its results (for the same sequence of input tuples) are equivalent to those of its sequential counterpart [10–12]. As explained in [11, 12], a sufficient condition for determinism for both cases is to require merging the timestamp-sorted physical streams delivering tuples to a stream and processing such tuples in timestamp-order once they are *ready*, as defined in [11]: t_i^j, being the i-th tuple from timestamp-sorted stream j, is *ready* to be processed when $t_i^j.ts \leq merge_{ts}$, where $merge_{ts} = min_k\{max_1(t_1^k.ts)\}$ is the minimum among the latest l tuple timestamps, one from each timestamp-sorted stream k. *ScaleGate* [11, 12] is a shared data object, leveraged and extended in this work, that (i) efficiently supports concurrent deterministic merging of timestamp-sorted streams into a timestamp-sorted stream of ready tuples, while (ii) allowing a number of *reader* entities to consume all the ready tuples of the latter stream. Its lock-free algorithmic implementation was shown to facilitate efficient deterministic processing [11, 12, 30, 31].

Load Balancing and Elasticity. As discussed in [10], the computational cost of a streaming application varies over time, depending on the rate with which input tuples are fed to it and depending on the tuples' data distribution. Because of this, a parallel execution in which the distribution of work to threads is statically decided at deploy time can lead to imbalances in the work of threads. When

the overall work is unbalanced but could be carried out by the available threads as a whole, a *load balancing* adaptive reconfiguration is needed to change the work distribution. If more threads are to be provisioned, the new work distribution should also assign some work to the newly allocated threads. Notice that it is essential to adjust resources such as threads, since over-provisioned systems can lead to high latency [30] or unnecessary costs [10]. Because of this, *elastic reconfigurations* should also be triggered when the work can be done by fewer threads (independently of whether they are imbalanced), thus decommissioning some threads and changing the work distribution. We use the term *reconfiguration* to refer to any of these.

3 PROBLEM MODELING AND OBJECTIVES

State, buckets and streaming parallelization model. We target a general-purpose intra-node streaming tool, where users can implement parallel stateful analysis, without explicit handling of complexities inherent to determinism or fast reconfigurations. We adopt a known parallelism model of the literature [7, 8, 10, 28], that allows users to define and maintain the *state* of a streaming operator over a set of *buckets*. Each bucket, a fine-grained portion of the operator's state, can be accessed and updated (based on the tuples being processed) by *one* of the threads that run *instances* of the operator. The number of buckets of an operator is commonly chosen to be greater than that of the maximum threads that can be in charge of running its instances. The portion of state assigned to each thread is thus a partition of the operator's buckets. In the following, M refers to a buckets-to-threads mapping function, where $M[k]$ denotes the thread id to which bucket k is assigned to.

In relation to the twitter example (§ 1), the state of an operator running such analysis could consist of per-hashtag indexes (of the next word to find) and counters (of sequences matched so far). To maintain such state, the user could assign each hashtag to exactly one bucket (e.g., using a hash-function) and parallelize the analysis by letting each parallel thread update the indexes and counters of hashtags contained in the buckets assigned to it.

Notice that the state of an operator can depend on all the tuples observed so far, or on a *window* of them, as in the case of stream Joins (§ 2). In this sense, the problem does not impose any restriction on the length of such portion. As discussed in § 1, we aim at a parallelization approach in which users do not need to partition *input tuples* to threads (as in key-by partitioning). In turn, this does not limit the problem to a type of parallelism in which each parallel thread runs the same analysis on different portions of input data. The stream join we use as one example (§ 4.3) presents such a case, in which all parallel threads carry out some of the processing for all input tuples.

Execution epochs. Under the assumption that all threads are fed with the same sequence of tuples, a reconfiguration implies a change in M to hold true from a certain tuple onward. We use the term *epoch* to refer to the period spanning tuples in between two event times (i.e. between timestamps of a pair of tuples), during which the mapping of buckets to threads does not change. Hence, being E_i the current epoch, T_i^P the set of processing threads and M_i the mapping in E_i, a reconfiguration implies the beginning of a new epoch E_{i+1} for which a new mapping M_{i+1} is used for a (possibly different) set of processing threads T_{i+1}^P.

Problem statement. Our goal is a framework that can facilitate the programming of stateful analysis when efficient parallelism cannot be simply achieved by partitioning input tuples to threads (e.g., as in key-by partitioning). From an SPE perspective, and with the goal of combining the benefits of shared-nothing and shared-memory approaches, we aim at designing and implementing a framework for intra-node parallel and reconfigurable stateful analysis with the following objectives:

- O1 A programmable interface that does not require thread-safe programming of stateful analysis (i.e., as in shared-nothing parallelism), thus granting exclusive read and write access to a portion of the operator's state (i.e., a portion of the buckets maintaining it) to each parallel processing thread.
- O2 Support for deterministic sharing of all input tuples to all processing threads in the same order (without requiring the user to define any input tuple partitioning scheme) and deterministic merging for all threads fed by multiple streams.
- O3 Support for fast reconfigurations.

We do not place restrictions on the logic with which reconfiguration actions are taken. Instead, we assume the existence of an external orthogonal *policy* in charge of triggering the reconfigurations. One such policy, triggering reconfigurations based on the threads' CPU consumption is used in our evaluation (§ 9).

Notice that, in contrast to objective O3, which can be only evaluated empirically, O1 and O2 can be met if a set of *sufficient properties* is satisfied [10–12, 30], distinguished into *intra-epoch* and *inter-epoch* ones (i.e. to hold within each epoch, respectively when transitioning from E_i to E_{i+1}). Intra-epoch properties to be enforced are:

- P1 All threads observe all input tuples in the same order.
- P2 For a given mapping M_i, each thread has exclusive read/write access to the bucket(s) mapped to it.
- P3 The tuples received by any thread from multiple streams are merged into a sequence of tuples processed by the thread once ready.

Additional, inter-epoch properties to be enforced are:

- P4 If a bucket is mapped to a processing thread p_1 in epoch E_i (i.e., if such a bucket is potentially modified based on the tuples processed by p_1) and to processing thread p_2 in epoch E_{i+1}, then the first tuple belonging to E_{i+1} is processed by p_2 after the last tuple belonging to E_i is processed by p_1.
- P5 Each reconfiguration takes place atomically (either by being applied in its entirety or not being applied).
- P6 Tuples sharing the same timestamp belong to exactly one epoch.

4 OVERVIEW OF *STRETCH*

Figure 1 outlines *STRETCH*'s overall architecture, utilising as main components the State Manager (*SM*) and Elastic ScaleGate (*ESG*) objects. For ease of presentation of the *STRETCH*' and *SM*'s architecture and functionality, before outlining them in § 4.2, we outline the *ESG*'s API and functionality in § 4.1. In § 4.3, we provide an example implementation for a join operator, which we also use in the experimental evaluation.

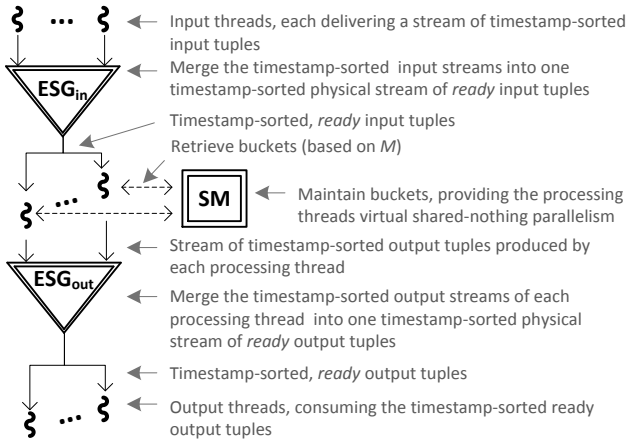


Figure 1: Overview of *STRETCH*.

4.1 The Elastic ScaleGate (*ESG*) data object

As mentioned in § 2, the Elastic ScaleGate, *ESG*, extends ScaleGate, which allows (i) a number of *source threads* to each insert in it a timestamp-sorted stream of tuples, and (ii) a number of *reader threads* to retrieve *ready* tuples from it in timestamp order, through the methods addTuple and getNextReadyTuple, respectively. They both encapsulate the necessary communication between *sources* and *readers*, to know whether a tuple is ready or not.

- addTuple(tuple,sID): allows a tuple from source sID to be merged by ScaleGate in the resulting timestamp-sorted stream of tuples.
- getNextReadyTuple(rID): provides to the calling reader rID the next earliest ready tuple that has not been yet consumed by rID. Note that each tuple, once it becomes ready, will be returned to all readers invoking the method.

For *ESG*, the API extension to enable changes in the sets of threads is listed below, outlining the additional methods and their behaviour, while their algorithmic implementation is described in § 7.

- announceReaders(List reader_IDs, rID): can be invoked by an existing *ESG* reader rID. The new readers can return ready tuples starting from the one that rID returned before calling this method.
- removeReaders(List reader_IDs): removes the denoted list of readers from *ESG*.
- announceSources(List source_IDs, min_ts): adds new source threads. To comply with the requirements for the identification of ready tuples, the method expects min_ts, the earliest timestamp the new sources can add, to be greater than the timestamp of the latest tuple retrieved by any reader when the method is invoked.
- removeSources(List source_IDs): removes from *ESG* the list of sources source_IDs. Any potential new tuple insertions by the latter will be ignored.

For all the above, among concurrent invocations and subsequent invocations with the same parameters, only one succeeds.

4.2 The *STRETCH* framework's architecture

Recalling from Figure 1, *STRETCH* uses a State Manager (*SM*) and two *ESG* objects, i.e. one for input tuples (ESG_{in}) and one for output tuples (ESG_{out}). Several threads interact with these components: *input threads* deliver input tuples to ESG_{in}, *processing threads* process input tuples and interact with *SM*, delivering results to ESG_{out}, while *output threads* retrieve the output tuples delivered to ESG_{out}. Despite not discussed for simplicity, *STRETCH* also defines a thread pool for provisioning and decommissioning. For ease of explanation, we focus our description on the parallel, elastic and deterministic execution of one stateful operator. The description extends for multiple operators, considering that the input (resp. output) threads of an operator are the processing threads of its upstream (resp. downsteam) peers.

To instantiate an operator, the *STRETCH user* initially provides:

$$\{M_0, T_0^I, T_0^P, T_0^O, \text{BucketImpl}, \text{filter}\}$$

M_0 defines the overall number of buckets and their initial mapping to the processing threads of epoch E_0, while T_0^I, T_0^P and T_0^O are the sets of input, processing and output threads for epoch E_0, respectively[1]. At runtime, each bucket is an instance of the BucketImpl class, defined by the user to implement the stateful analysis' logic. The class BucketImpl is expected to define a method process (to be invoked by T_0^P). The filter function, for each thread p and each tuple t the thread works on, aims at filtering out the buckets of p that need not be updated due to t. Shortly in this section we outline how filter can be used to speed-up the analysis. Once a stateful operator is instantiated, *STRETCH* defines the method:

$$\text{switchEpoch}(M)$$

which will change the mapping of buckets to threads (and the number of threads, if the reconfiguration is provisioning or decommissioning them). *STRETCH* relies on special tuples, named *epoch-switch* tuples to perform an epoch switch.

Each bucket is assigned to exactly one thread based on M. T^I (Alg. 1) are the source entities for ESG_{in}, T^P (Alg. 2) are the reader and the source entities for ESG_{in} and ESG_{out}, respectively, and T^O (Alg. 3) are the reader entities of ESG_{out}.

Algorithm 1: Input threads (T^I) - main loop

```
1 while executing do
2     retrieve / produce next tuple t
3     add t to ESG_in
```

T^I threads deliver each tuple (e.g., retrieved from the network or another operator) to ESG_{in} (Alg. 1, L 1-3). At the same time, each T^P thread p retrieves each next ready tuple t (Alg. 2, L 2) and checks whether t is an *epoch-switch* tuple or a regular one (Alg. 2, L 3). In the former case, p stores t to later trigger a reconfiguration (Alg. 2, L 4), at an appropriate time-point, to ensure determinism. In the latter case, the norm is to invoke process if needed; however, p first checks if t signifies the appropriate time-point to trigger reconfiguration, i.e., it checks whether there exists some previously stored *epoch-switch* tuple yet to be processed and if t's timestamp is greater than (i.e. not equal) that of the previous regular tuple (prev_ts). If so, p requests a new epoch, synchronizing with both *ESG*s and the *SM* (we provide details on synchronization in § 6) and retrieves the buckets mapped to it in such new epoch (Alg. 2, L 8- 10).

[1]We will skip index i for T^I, T^P, T^O, M in contexts not focusing on a specific epoch.

Algorithm 2: Processing threads (T^P) - main loop

```
1  while executing do
2      retrieve t from ESG_in
3      if t is an epoch-switch tuple then
4          store t in list pendingEpochSwitchTuples
5      else
6          if t is the first regular tuple ever processed then
7              get buckets assigned to this thread
8          else if pendingEpochSwitchTuples is non-empty and
             t.ts > prev_ts then
9              trigger the epoch switching protocol (Alg 7)
10             get buckets assigned to this thread in new epoch
11         for BucketImpl b returned by filter(buckets,t) do
12             Tuple[] outs = b.process(t)
13             add outs to ESG_out
14         prev_ts = t.ts
```

Algorithm 3: Output threads (T^O) - main loop

```
1  while executing do
2      retrieve next tuple t from ESG_out
3      process / forward t
```

Eventually, p proceeds traversing the buckets returned by the filter function, invoking the process function on them and delivering any output tuple to ESG_{out} (Alg. 2, L 11- 13). In parallel, T^O retrieve and process (or forward) from ESG_{out} the tuples delivered by the T^P threads (Alg. 3, L 1-3).

The role of function filter: To provide a hint about its usefulness, consider the analysis example in § 1 and one processing thread p. On one hand, not all tuples need to be processed by p, only the ones carrying at least one hashtag assigned to p. Moreover, not all buckets of p need to be updated when p processes a tuple t, only those that are about t's hashtags assigned to p. At the same time, any tuple carrying a timestamp falling outside the current window can cause a window-shift and could thus trigger the production of output tuples by p in a timely fashion. A sufficient (but not optimal) brute-force strategy to ensure that the buckets assigned to p are traversed when one of the above cases is given, is for p to traverse *all* its buckets for *each* input tuple. Instead of that, for efficiency, through the filter function, p can be instructed to traverse its buckets only when a tuple t carries hashtags assigned to p and when t falls outside the current window.

4.3 Example: *STRETCH*-implemented Join

ScaleJoin is a stream join that perform deterministic and efficient parallel stream processing. As described in detail in [12], in its parallelization approach, each of the n processing threads is responsible for running approx $1/n$ of the overall comparisons incurred by each input tuple. This is achieved by having all processing threads process each tuple but exactly one maintaining it in its local state, thus being responsible for the comparisons of future tuples with it. In *STRETCH*, this strategy can be implemented by having exactly one bucket (and thus exactly one thread) responsible for storing each tuple. Alg. 4 presents how the BucketImpl class can implement ScaleJoin's semantics. Whenever process is invoked, the tuple is used to purge the opposite window, check the predicate against the tuples of the opposite window and eventually adds itself to its window if the counter modulo the number of buckets (B) is equal to the thread id. Since each input tuple needs to be compared with all the tuples stored in any bucket, the filter function (Alg. 5) returns the entire set of buckets of the processing thread. Alg. 4, once run by *STRETCH*, guarantees the join semantics, since:

(1) The method process of each bucket, is invoked for all the tuples taken from ESG_{in} in the exact order in which such tuples are retrieved and is never invoked concurrently (for a given bucket) by two or more threads (Theorem 5.1).
(2) By (1) we have that all buckets update the counter consistently.
(3) By (1) and (2) we have that each tuple is stored in exactly one bucket.
(4) By (1) and (3) we have that each stored tuple is compared with all the tuples needed, according to the join semantics.

Algorithm 4: BucketImpl class for ScaleJoin [12]

```
1  Tuple[] W_R,W_L (portions of the global state in the bucket)
2  counter = 0 (replicated part of the global state, in each bucket)
3  Function Tuple[] process(Tuple t)
4      increase counter
5      purge t's opposite stream window
6      for each t' in t's opposite stream window do
7          if predicate holds for t and t' then
8              add ⟨t, t'⟩ to results
9      if this bucket index % B == counter then
10         store t in t's stream window
11     return results
```

Algorithm 5: filter function for ScaleJoin [12]

```
1  Function BucketImpl[] filter(BucketImpl[] b,Tuple t)
2      return b
```

5 INTRA-EPOCH PROCESSING

We detail here how the data structures and threads presented in § 4 interact within each epoch, i.e., for a given mapping M. We first introduce the API of SM and then show how, by the processing defined for the processing threads, *STRETCH* satisfies the intra-epoch properties listed in § 3, namely that all processing threads T_P (i) observe all input tuples in the same order (P1), each with exclusive read/write access to its buckets (P2) and (ii) produce streams of output tuples that are deterministically merged into a logical sequence of output tuples (P3). The description is later extended in § 6 for inter-epoch processing. *SM*'s API methods are:

- setup(M_0,BucketImpl): initializes the *SM* associated to the stateful operator. Based on M_0, *SM* knows how many buckets should be maintained and the thread id to which each one is assigned to initially.

- getBuckets(Thread id): used by the processing threads to retrieve the buckets assigned to each one of them for the current epoch.
- requestNewEpoch(M): announces to SM the intention of starting a new epoch using the mapping M.

Alg. 6 presents the SM implementation for methods `setup` and `getBuckets`. The method requestNewEpoch(M) is presented in detail in § 6. Method setup is invoked when the stateful operator is instantiated and relies on M_0 for the mapping of buckets to threads during epoch E_0. Internally, the SM creates as many instances of the given `BucketImpl` class as buckets defined in M_0. Method `getBuckets` is then invoked by T_0^P threads to retrieve the buckets assigned to them based on M_0. As shown in Alg. 6, the method returns a set of (pointers to) buckets. Hence, the state of the stateful operator as a whole is not composed by disjoint partitions maintained locally at each thread, but rather defined in a single array of buckets, whose elements are assigned to T_0^P threads. Assuming shared memory, this enables virtual shared-nothing parallelism since each T_0^P thread has exclusive access to its buckets but, at the same time, buckets can be re-assigned to threads without state transfer.

Algorithm 6: *SM* implementation

```
1  BucketImpl[] buckets
2  Function setup(M0,BucketImpl)
3      store M0
4      for i = 1 . . . size(M0) do
5          store new BucketImpl instance in buckets[i]
6  Function BucketImpl[] getBuckets(Thread id)
7      BucketImpl[] threadBuckets
8      for i = 1 . . . size(M) do
9          if M[i]==id then
10             add a pointer to buckets[i] to threadBuckets
11     return threadBuckets
```

5.1 Enforcing properties P1-P3 in E_0

At this point, we argue that *STRETCH* satisfies properties P1-P3 during the first epoch E_0 (i.e. from the moment a certain stateful operator is deployed, to its first reconfiguration or for the entire execution of a parallel operator with a static mapping of buckets to threads). It should be noticed that, since no reconfiguration is defined before E_0, the behavior of the two *ESG*s is equivalent to that of the base ScaleGate. This argumentation is later extended to any epoch E_i by induction, after showing that properties P4-P6 are met when transitioning across epochs.

Theorem 5.1. *STRETCH satisfies properties P1-P3 in* E_0.

Proof. (Sketch) Property P1 is satisfied by leveraging the ESG_{in} and Alg. 2, since the former delivers all input tuples (once ready) in the same order to all T_0^P threads while the latter does not discard any input tuple. Property P2 is enforced by Alg. 2 and the SM's implementation (Alg. 6) since the former retrieves the buckets exactly once (upon processing of the first tuple) while the latter returns each bucket to one and only one thread in T_0^P. Property P3 is satisfied because each thread in T_0^P delivers a non-decreasing timestamp-sorted stream of output tuples and the merging of such streams is carried out deterministically by ESG_{out}. □

6 INTER-EPOCH PROCESSING

Here we describe how *STRETCH* transitions from one epoch to another, while guaranteeing properties P4-P6 (§ 3). We first give a high level description of the protocol and later provide more detail and we prove that properties P4-P6 are met while switching from epoch E_i to epoch E_{i+1}, thus extending Theorem 5.1 to any epoch E_i.

At this point recall that input tuples can be of type *regular* or *epoch-switch*. The former refers to regular tuples, the latter refers to special control tuples used by *STRETCH* when switching epochs. In a nutshell, when a special *epoch-switch* tuple t^* is received in E_i by the T_i^P threads, the epoch switch protocol is triggered as soon as the first *regular* tuple $\bar{t}$ with a timestamp greater than the latest timestamp observed before t^* is received (as also shown in Alg. 2 L 8-10). As we further elaborate in the following, this implies that property P6 holds. Independently of the nature of the switch from epoch E_i to epoch E_{i+1} (i.e., decommissioning, load balancing or provisioning), all T_i^P threads invoke the method requestNewEpoch(Mapping M) of SM and block there until the method returns. When threads are provisioned, one of the current T_i^P threads activates the necessary new threads from the thread pool and connects them with ESG_{out} and ESG_{in}. When threads are decommissioned, the latter are disconnected from ESG_{in} and ESG_{out} and are returned to the thread pool.

6.1 Switching epochs

As outlined in § 4, *STRETCH* provides the method switchEpoch(M) to express the intention of switching the current epoch E_i to the epoch E_{i+1}, in which mapping M is enforced. When this function is invoked, an *epoch-switch* tuple carrying M is inserted in ESG_{in} by each T_i^I thread. For each T_i^I thread, the timestamp of such *epoch-switch* tuple is set to that of the latest tuple added to ESG_{in} by the T_i^I thread. This, combined with the definition of ready tuples, implies that at least one of these *epoch-switch* tuples is immediately ready for T_i^P threads process.

As shown in Alg.2, all *epoch-switch* tuples retrieved by a thread in T_i^P are initially stored in its local list pendingEpochSwitchTuples (L 4). At any execution point, there could be many *epoch-switch* tuples stored by T_i^P threads, either referring to the same switchEpoch invocation (remember all T_i^I threads forward one such tuple) or to different invocations of the switchEpoch method (if the latter is invoked when pending epoch transitions are still to be completed). Each T_i^P thread checks whether one or more *epoch-switch* tuples are in pendingEpochSwitchTuples, only when an incoming regular tuple $\bar{t}$ with a timestamp greater than (i.e. not equal to) the previous one is retrieved from ESG_{in} (§ 4). Since all T_i^P threads retrieve all tuples in the same order from ESG_{in}, all T_i^P threads have the same set of *epoch-switch* tuples with timestamp equal to or smaller than $\bar{t}.ts$ in their pendingEpochSwitchTuples lists at the time $\bar{t}$ is processed. Because of this, if one or more *epoch-switch* tuples exist in pendingEpochSwitchTuples upon processing of $\bar{t}$, the most recent *epoch-switch* tuple t^* referring to E_j ($j \geq i$) is processed (if any), while the rest of them are discarded.

Alg. 7 presents the steps followed by the T_i^P threads to switch epoch. First, the most recent unprocessed epoch-switch tuple t^* is

Algorithm 7: Switching epoch protocol (for a thread in T_i^P upon retrieving of $\bar{t}$)

```
1 retrieve most recent epoch-switch tuple t* to be processed from
    pendingEpochSwitchTuples and discard the rest
2 if t* refers to E_j where j ≥ i then
3     SM.requestNewEpoch(t*.M) // blocking call
4     if t*.M requests provisioning of threads then
5         ESG_out.announceSources(new threads ids, t̄.ts)
6         ESG_in.announceReaders(new threads ids, this thread id)
7     else if t*.M requests decommissioning of threads then
8         ESG_in.removeReaders(removed_ids)
9         ESG_out.removeSources(removed_ids)
```

Algorithm 8: Method requestNewEpoch(M) (SM)

```
1 Method requestNewEpoch(Mapping M)
2     block until the union of processing threads for this epoch invokes
        this method
3     use M as mapping from now on
```

retrieved and the rest of epoch-switch tuples are discarded from the pendingEpochSwitchTuples. Then, if t^* belongs to E_j where $j \geq i$, the blocking method requestNewEpoch of SM (Alg. 8) is invoked by all T_i^P threads passing the new mapping $t^*.M$. This method will change the mapping used by SM to $t^*.M$ as soon as all the T_i^P threads have invoked the method. If the set of processing threads defined by $t^*.M$ is larger than that of the current epoch (i.e., if threads are to be provisioned), then these are announced by all T_i^P threads as sources to ESG_{out} and readers to ESG_{in}. Alternatively, if the set of T_i^P threads defined by $t^*.M$ is smaller than that of the current epoch (i.e., if threads are to be decommissioned), methods removeReaders and removeSources are invoked for ESG_{in} and ESG_{out}, respectively.

6.2 Satisfying properties P4-P6 from E_i to E_{i+1}

Theorem 6.1. *STRETCH satisfies properties P4-P6 when switching from E_i to E_{i+1}.*

Proof. (Sketch) In order to prove P4 is satisfied, it should be noted that, based on Algs. 2, 7 and 8, the following invariants hold:

(1) $\exists$ unique regular tuple $\bar{t}$, that is seen in the same relative position in the stream by all threads of E_i and that distinguishes epochs (i.e. mapping of buckets to threads); $\bar{t}$ is the first tuple of the new epoch. Specifically:
   (a) $\forall t' | t'.ts < \bar{t}.ts, t' \in$ old epoch
   (b) $\forall t'' | t''.ts \geq \bar{t}.ts, t'' \in$ new epoch
(2) Let $out(t)$ denote the set of output tuples triggered by a tuple t. Then:
   (a) $\forall t' | t'.ts < \bar{t}.ts$, $out(\bar{t})$ is read by threads T^O after $out(t')$
   (b) $\forall t'' | t''.ts \geq \bar{t}.ts$, $out(t'')$ is read by threads T^O after $out(\bar{t})$.

Moreover, such $\bar{t}$ is chosen to be the first regular tuple with a timestamp greater than (i.e. not equal to) the previous one (observe that $\bar{t}$, as all tuples in ESG, is seen by all T_i^P). This, together with the assumption on output tuples timestamps and the fact that ESG preserves ordering, implies P6. Finally, note that, while multiple epoch-switch tuples can be stored at the same time, two threads in T_i^P cannot be more than one epoch away because of the blocking method requestNewEpoch; this implies P5. □

6.3 Satisfying properties P1-P3 in $E_i, \forall i > 0$

Theorem 6.2. *$\forall i > 0$, STRETCH satisfies properties P1-P3 in E_i.*

Proof. (Sketch) All the T^P threads of epoch E_i process all tuples in E_i. This is because of one of the following cases: (i) If threads have been provisioned for E_i, the new readers can return ready tuples starting from the latest ready tuple gotten by the calling reader that succeeded to execute announceReaders, which is $\bar{t}$ (Alg. 7 precondition (caption), and L 5-6). Hence the new readers will retrieve their assigned buckets before processing $\bar{t}$ (Alg. 2 L 7). The new threads are also already registered as sources to ESG_{out} (Alg. 7 L 5-6), so if the processing of $\bar{t}$ or any subsequent tuple triggers any output tuple, the latter will be deterministically delivered by ESG_{out} once ready. (ii) Alternatively, if threads have been decommissioned, threads only existing in E_{i-1} are no longer readers of ESG_{in} or sources of ESG_{out} (once one of the calls by any existing threads to methods removeReaders and removeSources has completed) and have terminated. Hence, properties P1-P3 hold in any arbitrary E_i as they do in E_0. □

7 ALGORITHMIC IMPLEMENTATION OF *ESG*

ESG, similarly to ScaleGate, builds a list where tuples are maintained in timestamp order, along with some auxiliary book-keeping structures. The protocol for adding and accessing tuples is customized for the needs of data streaming operator pipelines. Recall that each source thread adds tuples in timestamp order, while each reader traverses the sorted stream of tuples, so that each tuple t will be returned to each invoking reader, once t becomes ready. The algorithmic implementation of all the methods is outlined below. The addTuple and getNextReadyTuple methods are similar to ScaleGate's [11, 12], while the methods to modify the set of threads accessing the object are new.

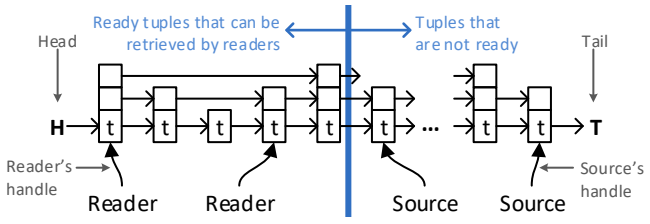


Figure 2: ScaleGate's and *ESG*'s skip list, and readers' / sources' handles.

addTuple, getNextReadyTuple: The algorithmic implementation constructs a skip list, with auxiliary book-keeping structures —essentially acting as thread-specific data for the sources and readers— and fine-grained synchronization to avoid global locking. 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 Fig. 2, 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 next insertion "falls" after its previous one (i.e., closer to the tail). All the tuples before (i.e., with earlier timestamps) the earliest tuple pointed by the source are ready.

announceReaders(List reader_IDs, rID), removeReaders(List reader_IDs): As mentioned above, a reader has access to one of *ESG*'s nodes through its own handle. A new reader p to the *ESG* simply needs a handle to a node that is ready, so that p can safely traverse the rest of the list in timestamp order in subsequent getNextReadyTuple invocations. Since announceReaders is called by an existing reader, the caller's handle to the most recently read node of such reader is used, so that all the new readers have a handle to the *ESG*. Removing a reader is as simple as removing the thread-specific structures of that reader.

announceSources(List source_IDs, min_ts): A new source to be registered in *ESG* needs its own related book-keeping structures, i.e. its own handles, which essentially can be copying the handles of an existing source. for the sake of the new thread, an initial *dummy* tuple with timestamp min_ts is inserted, to initialize the functionality of its handles. Dummy tuples are not returned as ready to readers invoking getNextReadyTuple, but enable other tuples with smaller timestamps to be characterized as ready and be returned to readers. At this point it is useful to recall that the announceSources operation is called for ESG_{out} after the processing thread has returned from the blocking method requestNewEpoch (Alg. 8) with min_ts = $\bar{t}.ts$, thus ensuring that the source calling announceSources on ESG_{out} is still pointing to a tuple with a timestamp smaller than min_ts, thus guaranteeing the method is always invoked with appropriate timestamp for the respective parameter (cf. also Theorem 6.2). Adding more than one source at a single time is delegated to a single thread that will add a block of new book-keeping structures and dummy tuples (each pointing to the respective new sources).

removeSources(List sources_ids): Removing a source consists mainly of adding, on behalf of the source, a specially marked *flush* tuple in *ESG*, with timestamp equal to the latest insertion of the source. The effect of such tuple is that it will essentially push the previously added tuples of the leaving source to be ready and the removed source's associated book-keeping structures can be removed. As with *dummy* tuples, flush tuples are not returned as ready by getNextReadyTuple. For the removal to be safe without loosing tuples, it should be invoked when it is known that the source does not have pending insertions of ordinary tuples. In *STRETCH* this is ensured by the invocation of the method in Alg. 8.

Concurrent calls of the same method that updates the set of threads (e.g. concurrent calls to announceReaders), or similar calls in the same epoch may happen; synchronization is in place (using a TestAndSet variable) so that only one of each type takes effect. Concurrent calls among competing such methods (e.g. announceReaders and removeReaders) are not supposed to happen, as they both need to modify the thread-specific book-keeping structures (indeed such invocations are not done in *STRETCH*). If an *ESG* implementation wants to allow that, it will need to enforce synchronization to protect consistency; since these are low-contention operations, a simple lock can do. If regular operations (to add and get tuples) are concurrent with those that update the set of threads and the respective book-keeping structures, the latter can overwrite, causing the former to have no effect. Note that their use in *STRETCH* imply that such invocations do not interfere.

8 MODELLING *STRETCH*'S PERFORMANCE

Before evaluating *STRETCH*, we model in this section the expected scalability behavior of its virtual shared-nothing parallelism (V_{SN}) and that of pure shared-nothing parallelism (P_{SN}).

Let us consider a setup in which three threads t^I, t^P and t^O are defined for T^I, T^P and T^O, respectively. Moreover, let us assume that d^I, d^P and d^O are the per-tuple expected processing times for t^I, t^P and t^O, respectively. If we aim at scaling the analysis of these threads by provisioning more threads to T^P, then t^P is the bottleneck of the pipeline. That is, $d^P = \max(d^I, d^P, d^O)$. Otherwise, the threads to be provisioned should be dedicated to T^I if $d^I = \max(d^I, d^P, d^O)$, or T^O if $d^O = \max(d^I, d^P, d^O)$. If we opt for P_{SN}, we need to define a mechanism for t^I to route tuples to T^P's threads. The routing overhead will depend on the semantics of the stateful operator (e.g., if key-by partitioning can be leveraged or if tuples should be broadcast to T^P's threads). A mechanism to merge-sort deterministically tuples at t^O is also required. When n threads are defined for T^P, we refer to $d^R_{n,P_{SN}}$, $d^P_{n,P_{SN}}$ and $d^M_{n,P_{SN}}$ as the time spent by t^I to route tuples, the new per-thread processing time of T^P's threads and the time spent by t^O to merge tuples deterministically. If, alternatively, we opt for *STRETCH*'s V_{SN}, we are not required to route tuples, since all of them can be directly accessed by T^P's threads, but we need to account for the synchronization and congestion overheads incurred by ESG_{in} and ESG_{out} to share tuples and support determinism. In this case, when n threads are defined for T^P, we refer to $d^C_{n,V_{SN}}$, $d^P_{n,V_{SN}}$ and $d^M_{n,V_{SN}}$ as the synchronization and congestion overheads incurred by t^I, the new per-thread processing time of T^P's threads and the time spent by t^O to merge tuples deterministically.

As discussed in [11, 12], we expect $d^M_{n,V_{SN}} < d^M_{n,P_{SN}}$ since the merging-sorting costs can be distributed to all T^P threads in the V_{SN} case. Because of such costs, and because of the possible extra-costs incurred by T^P's threads to chose which tuples to process, nonetheless, we also expect $d^P_{n,V_{SN}} \geq d^P_{n,P_{SN}}$. Finally, we also expect $d^C_{n,V_{SN}} < d^R_{n,P_{SN}}$ for efficient implementations of the shared *ESG* data structures [11]. Based on this, we thus expect P_{SN} to scale to the highest n so that:

$$d^P_{n,P_{SN}} = \max(d^I + d^R_{n,P_{SN}}, d^P_{n,P_{SN}}, d^O + d^M_{n,P_{SN}})$$

Similarly, we can expect V_{SN} to scale to the highest n so that:

$$d^P_{n,V_{SN}} = \max(d^I + d^C_{n,V_{SN}}, d^P_{n,V_{SN}}, d^O + d^M_{n,V_{SN}})$$

Hence, V_{SN} allows for better scalability than P_{SN} for all n so that:

$$\begin{cases} d^I + d^C_{n,V_{SN}} < d^P_{n,V_{SN}} < d^I + d^R_{n,P_{SN}} \\ d^O + d^M_{n,V_{SN}} < d^P_{n,V_{SN}} < d^O + d^M_{n,P_{SN}} \end{cases}$$

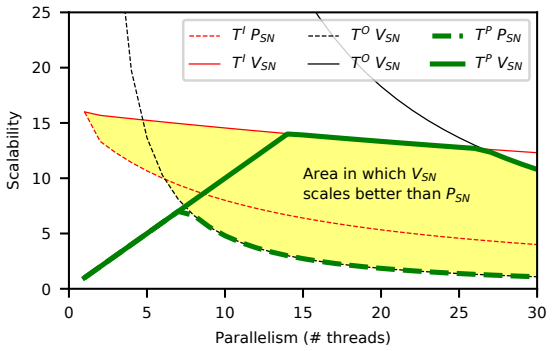


Figure 3: Scalability estimation based on the model

d^P	d^O	$d^P_{n,P_{SN}}$	$d^P_{n,V_{SN}}$	$d^R_{n,P_{SN}}$	$d^C_{n,V_{SN}}$	$d^M_{n,P_{SN}}$	$d^M_{n,V_{SN}}$
$16d^I$	d^I	$\frac{d^P}{n}$	$\frac{d^P}{n}$	$0.1d^I n$	$0.01d^I n$	$0.1d^I n \log(n)$	$0.01d^I n \log(n)$

Table 1: Model's variables sample costs for Figure 3

Figure 3 shows how T^I's, T^P's and T^O's threads scale, based on this model, when variables are set as specified in Table 1. In this case, the $n \cdot \log n$ factor in $d^M_{n,P_{SN}}$ and $d^M_{n,V_{SN}}$ models the merge-sorting costs, while $d^R_{n,P_{SN}}$ has an n factor to model the cost of broadcast communication (later evaluated in § 9).

For simplicity, the figure shows T^I's and T^O's max rates' scalability without accounting for bounded communication queues among threads nor mechanisms such as flow control. Also, since we assume in this model the existence of a single thread in T^I, we do not explicitly account for P_{SN} for the additional costs needed to merge tuples at T^P's threads if the latter are forwarded by multiple physical streams.

9 EVALUATION

We first empirically validate the model of § 8, including virtual shared-nothing parallelism V_{SN} (in *STRETCH*) and shared-nothing parallelism P_{SN} (in Flink), for both synthetic and real-world data, from Twitter. Then, we evaluate *STRETCH*'s performance for the ScaleJoin usecase (§ 4.3) and compare it with that of the original implementation [12], focusing on intra-epoch throughput and latency for the maximum sustainable rate and studying the scalability for increasing number of threads. Lastly, we evaluate *STRETCH*'s elasticity by provisioning/decommissioning threads, measuring the reconfiguration time and its effect on throughput and latency.

Evaluation setup. Experiments run on a 2.10 GHz Intel(R) Xeon(R) E5-2695 CPU with 2 sockets (each with 18 cores), 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 with default garbage collection settings. For P_{SN}, we use Flink 1.6.0.

All experiments show the average end-to-end latency and throughput over five distinct runs. The former is the timestamp difference of each output tuple and the latest input tuple that contributes to it. To compute the latter, we let input streams inject tuples at full speed while using *flow control* to handle backpressure. The implemented flow control mechanism is similar to that of Flink, where the rate of a stream is adjusted by an intermediate bounded queue.

9.1 V_{SN} vs P_{SN} scalability - synthetic dataset

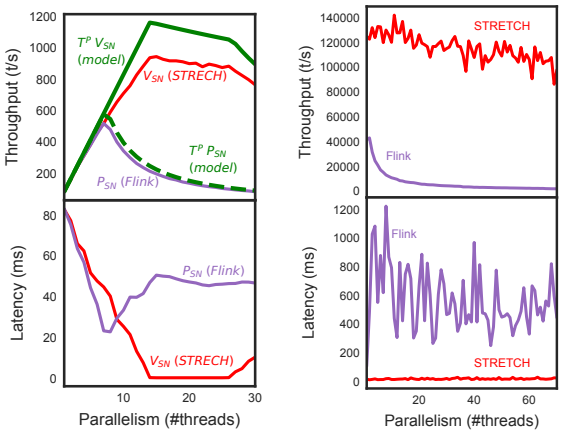
To evaluate our model (§ 8), we used two synthetic queries in Flink. In our implementation, d^I is approximately 0.7 msec (i.e. it takes 0.7 msec for t^I to create and forward an input tuple). Following the costs presented in § 8, the values of all parameters are then calculated for different number of threads for T^P. Flink dedicates one thread per source, per parallel instance of an operator, and per sink. To match the number of threads used by Flink to that of *STRETCH*, we define one source for each T^I thread, we set the operator parallelism to the number of T^P threads, and define one sink for each T^O thread. We use Flink's broadcast primitive to share tuples between the sources and the parallel operator instances.

Figure 4a shows the evalution results. To compare the observed behavior with that of the model, the figure also includes the expected scalability based on the model (i.e. as in Fig. 3). As shown, the throughput figure of P_{SN} matches the corresponding one from the model. The behaviour of V_{SN} also matches that of the model (despite the larger deviation between the observed and expected behavior due to the higher overheads in the system). For both V_{SN} and P_{SN}, the rate of T^P threads grows to match that of T^I while the latency decreases due to the smaller delay for tuples to be processed. V_{SN} achieves higher throughput and lower latency than P_{SN}, as also expected based on the model. Once T^O's threads become the bottleneck of the pipeline, further increasing the number of threads for T^P results in a growing latency. In this case, the buffers used to handle the backpressure by Flink (not discussed in our model) stabilize the latency by controlling the input rate.

9.2 V_{SN} vs P_{SN} scalability - Twitter dataset

In this case, we study the maximum throughput for a deterministic operator with two logical streams. T^P threads pass tuples downstream without performing any processing. By removing the processing cost, the maximum throughput is then given either by the speed of T^I's threads or the sorting costs of T^P or T^O threads. We create two similar queries in Flink (as P_{SN}) and *STRETCH* (as V_{SN}), and process a dataset consisting of 4.3 million tweets, between the 1^{st} and 2^{nd} of October 2018. Also in this case, our Flink implementation is defined so that the number of threads used to forward and process tuples matches *STRETCH*'s one. We use two threads for T^I to read and forward input tuples. In *STRETCH*, since T^P and T^O threads receive tuples from *ESG*s in timestamp order, there is no need for sorting. However, in Flink, T^P and T^O threads are responsible for sorting the tuples to support determinism.

Figure 4b shows the operators' scalability. For an increasing number of T^P threads, *STRETCH* faces a slight decrease in throughput due to the synchronization overheads induced by the higher number of threads. P_{SN} in Flink starts from a lower throughput and decreases faster for an increasing number of threads. Moreover, the average latency in P_{SN} in Flink, regardless of the number of T^P threads, is higher than 200 msec, while V_{SN}'s in *STRETCH*'s is always less than 20 msec.



(a) Synthetic workload **(b) Twitter**

Figure 4: Throughput and latency of V_{SN} and P_{SN}.

9.3 ScaleJoin usecase

We continue our evaluation focusing on a specific operator with two logical streams: ScaleJoin [12]. The performance of this parallel operator can be hampered in existing SPEs because of the need for broadcasting tuples to all T^P threads. Because of this, and since Flink only supports parallel EquiJoin, we compare our results in this case with those achieved by the ScaleJoin implementation.

We follow the same benchmark used in [12, 25] to join two logical streams of R and S tuples. R tuples carry attributes $\langle ts, x, y\rangle$, where x is type of `int` and y is `float` while S tuples carry attributes $\langle ts, a, b, c, d\rangle$ where a, b, c, and d are types of `int`, `float`, `double` and `boolean`, respectively. For each pair of tuples t_R and t_S, an output tuple with schema $\langle ts, x, y, a, b, c, d\rangle$ is produced if:

$$t_S.a - 10 \leqslant t_R.x \leqslant t_S.a + 10 \; and \; t_S.b - 10 \leqslant t_R.y \leqslant t_S.b + 10$$

Attributes x, y, a and b are randomly selected from a uniform distribution with interval $[1, ..., 10000]$ which results on average in an output tuple every 250000 comparisons.

Intra-epoch performance. To assess the intra-epoch performance of *STRETCH*, we first check its scalability regarding the number of T^P threads. Also, we show the performance of a single thread, processing the tuples sequentially. In the single thread implementation, we use one thread per T^I, T^P, and T^O sets. There is one bounded buffer between the threads of T^I and T^P and one buffer between the ones of T^P and T^O, to control the rate. Hence, the single thread implementation is similar to the ScaleJoin and *STRETCH* with one thread for T^P, where the ScaleGates are substituted with bounded buffers. Figure 5 shows (i) the maximum sustainable input rate averaged over 5 different runs (and the standard deviation) for increasing number of T^P threads, (ii) the corresponding throughput, in terms of number of comparisons and (iii) the corresponding latency in logarithmic scale, for a time-based window of size 5 min.

As shown in the intra-epoch column of the figure, the throughput of ScaleJoin and *STRETCH* with one thread for T^P is similar to the single thread but the latency for the single thread is lower than the other two, which is the cost of using ScaleGate. However, as expected (and as discussed in detailed in [12]), by increasing the number of T^P threads, the throughput for ScaleJoin and *STRETCH* grows linearly. Although hyper-threading (after 36 physical threads) causes a degradation, by keeping increasing the number of T^P threads, both *STRETCH* and ScaleJoin are still capable of scaling. The latency for the highest throughput achieved by different numbers of threads for T^P shows that the *STRETCH* achieves latency in the same order as that of ScaleJoin.

Inter-epoch performance. As discussed in § 3, *STRETCH* provides a general API that allows any external policy to take decisions about when to provision or decommission a certain number of threads (based on the statistics provided by the *STRETCH* framework itself or other external information). In these experiments, similarly as done in [10], we trigger the provisioning or decommissioning of threads based on the processing capacity of the threads. More concretely, we define an upper, a target and a bottom processing capacity threshold. 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 smallest amount of underutilized threads needed to bring the average processing capacity below the target threshold is decommissioned. In our experiments, the upper, target and bottom thresholds are set to 90%, 70%, and 45% of the threads maximum throughput, respectively.

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 input rate 70% of the maximum rate that can be sustained by the corresponding number of T^P threads for time-based window with window size 5 minutes. After 6 minutes, when the window is full and the system is stable, we increase the rate to 120% of the maximum sustainable rate which requires provisioning of new threads for T^P in order to keep up with the input rate, and therefore trigger an epoch switch. For the decommissioning experiments, we start with 70% of the maximum sustainable rate and then after 6 minutes, similarly as the previous experiment, decrease the rate to 30%. In this case, the system needs decommissioning a few number of active T^P threads in order to utilize resource usages. Table 2 presents the number of threads that need to be provisioned or decommissioned depending on the number of T^P threads running in the system.

Figure 5, columns labeled "provisiong" and "decommissioning", show the effect of increasing or decreasing the workload for ScaleJoin and *STRETCH*, and consecutively provisioning or decommissioning threads for T^P for the 18 threads in *STRETCH*. As shown for the provisioning, by increasing the workload, *STRETCH* can sustain higher rates when adding new threads, resulting in higher

starting $\lvert T^P \rvert$	5	9	18	30	40
$\lvert T^P \rvert$ after Provisioning	9	16	31	52	69
$\lvert T^P \rvert$ after Decommissioning	2	3	7	12	17

Table 2: Provisioned / decommissioned threads depending on the number of T^P threads.

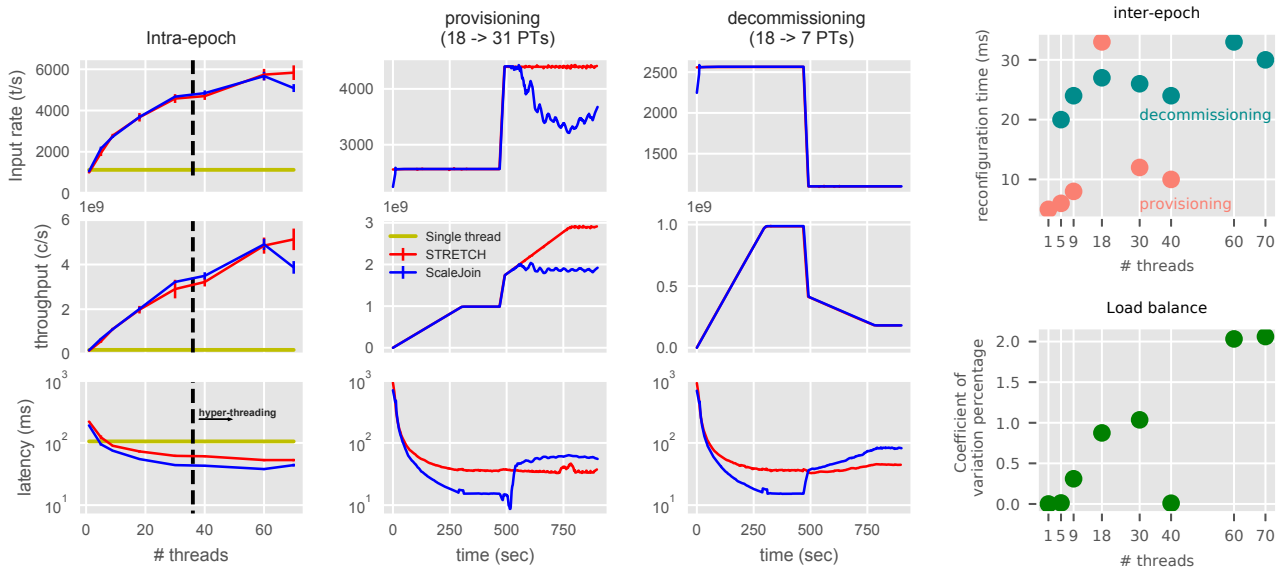


Figure 5: The performance of *STRETCH* framework with ScaleJoin for Join operator using time-based window with window size 5 minutes

throughput without affecting the latency. In the decommissioning procedure, when decreasing the workload, not only *STRETCH* achieves the same throughput as ScaleJoin, as expected, but it also shows slightly lower latency.

Moreover, we measure the reconfiguration time, which starts from the moment *STRETCH* receives a reconfiguration command till it successfully finishes executing it. As shown at the rightmost column in Fig. 5, the reconfiguration time is always less than 40 msec, which indicates there is no significant degradation during the epoch switch. Furthermore, at the same column, we show the load imbalance, in terms of coefficient of variation percentages. As it can be observed, in most cases there is at most 1% difference, while in all cases the difference is at most 2%.

Summary of the evaluation results

The empirical study (i) validates the model in § 8 (Fig. 4 and 3), (ii) demonstrates very fast reconfiguration possibilities, of just a few msec (Fig. 5 inter-epoch parts), enabled by *STRETCH*, while it also (iii) shows the low-overhead induced for in achieving these (Fig. 5 intra-epoch column), while preserving determinism in *STRETCH*.

10 RELATEDWORK

Several scalable and elastic parallel approaches have been discussed in the literature, e.g. [10, 22, 24]. For a systematic review, we refer the reader to [14]. This work does not focus on a particular strategy or a specific operator, but rather provides a general framework with the goal of promoting virtual shared-nothing parallelism (also showing it can embrace existing parallelism schemes), which, to the best of our knowledge, has not been studied before. The following is a summary of the state of the art in elasticity for intra-operator parallelization, including relation to our results. We organize the discussion in terms of key goals and properties, i.e. number of threads that can change per reconfiguration, the roles of state transfer and of the triggering mechanism, the focus on determinism and the reaction time vs overhead of frequent reconfigurations.

Regarding changes in number of threads, the literature provides methods for provisioning and decommissioning one thread at a time (e.g., [26], [19]) or more threads (e.g., [17]), as in our case. Differently from us, nonetheless, [17] does not actively target determinism.

Regarding state-transfers, one issue, orthogonal to our work, is related to load balancing, a combinatorial problem, related to packing (cf. [3, 10, 16, 17] and references therein). Another issue is that of the cost of transferring, since the overheads of state serialization and deserialization can degrade the SPE's performance. This can be alleviated by techniques aiming at reducing latency spikes, such as the ones in [15], or at recreating small states at the downstream thread by sending to the latter previously sent tuples (rather than transferring the upstream's thread's state), or by distributing the work to nodes through hashing, in ways that minimize the changes when rehashing [8]. Our work enables possibilities for efficiency due to sharing and advances the front of scaling-up by not requiring any state transfer, thus making these issues orthogonal and existing methods complementary to ours.

Targeting determinism needs appropriate synchronization. Determinism has been formalized in the context of parallel SPEs and in the context of algorithms for parallel data streaming operators, for instance by [11, 12] and is also referred to as *safety* by [18] and *semantic transparency* by [10]. Here we show sufficient conditions for determinism, also under reconfigurations.

Another key issue is when to trigger reconfigurations, as it is related to trade-offs between overheads and time to react when reconfigurations are needed. In the literature there exist proactive and reactive approaches, load-based approaches and application-performance- related approaches (cf. [6, 10, 19, 21, 23, 32] and references therein). Various triggering mechanisms can be combined

with *STRETCH*. As discussed by [6] with respect to SASO properties (i.e., Stability, Accuracy, Settling time and Overshoot), elasticity mechanics build on top of contrasting objectives. In addressing this issue, the work presented here implies better margins, due to the extra efficient reconfiguration proposed when scaling-up is the target.

11 CONCLUSIONS AND FUTURE WORK

We presented *STRETCH*, a framework that provides virtual shared-nothing parallelism and supports determinism while easing the programming of scalable, elastic, high-throughput and low-latency stateful streaming analysis. The ease of use is not only because *STRETCH* does not require coding of state transfer protocols but also due to the specified set of conditions that, once fulfilled, we have shown they imply deterministic execution. *STRETCH* builds on several results of the recent years for efficient intra-node stream processing and fuses them in a holistic framework, able to embrace many parallelization strategies. We provide empirical evidence that *STRETCH*'s virtual shared-nothing parallelism performs as modeled offering improvements over pure shared-nothing parallelism. We also show that *STRETCH* outperforms state-of-the-art tools such as ScaleJoin and components in SPEs such as Flink, also providing fast reconfigurations, in the realm of few msec.

Future directions include the study of possibilities to combine *STRETCH*'s intra-node elastic approach with complementary inter-node elasticity, focusing on joint schemes for homogeneous distributed cloud-based systems, or heterogeneous fog-based ones.

ACKNOWLEDGMENTS

We thank our shepherd, Paris Carbone, and the anonymous reviewers for their constructive comments and suggestions. The work was supported by the SSF proj. "FiC" nr. GMT14-0032, by the Chalmers Energy AoA framework proj. INDEED and STAMINA and by the Swedish Research Council proj. "HARE" nr. 2016-03800.

REFERENCES

[1] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Wittle. 2015. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proc. Endowment* 8, 12 (2015), 1792–1803.

[2] Magdalena Balazinska, Hari Balakrishnan, Samuel R Madden, and Michael Stonebraker. 2008. Fault-tolerance in the Borealis distributed stream processing system. *ACM Trans. on Database Systems (TODS)* 33, 1 (2008), 3.

[3] Cagri Balkesen, Nesime Tatbul, and M Tamer Özsu. 2013. Adaptive input admission and management for parallel stream processing. In *Proc. of the 7th ACM Int'l Conf. on Distributed event-based systems*. ACM, 15–26.

[4] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. 2017. State management in Apache Flink®: consistent stateful distributed stream processing. *Proc. VLDB Endowment* 10, 12 (2017), 1718–1729.

[5] Valeria Cardellini, Matteo Nardelli, and Dario Luzi. 2016. Elastic stateful stream processing in storm. In *High Performance Computing & Simulation (HPCS), 2016 Int'l Conf. on*. IEEE, 583–590.

[6] Tiziano De Matteis and Gabriele Mencagli. 2017. Proactive elasticity and energy awareness in data stream processing. *Journal of Systems and Software* 127 (2017), 302–319.

[7] flink [n. d.]. Apache Flink. https://flink.apache.org. ([n. d.]). Accessed:2019-3-1.

[8] Buğra Gedik. 2014. Partitioning Functions for Stateful Data Parallelism in Stream Processing. *The VLDB Journal* 23, 4 (Aug. 2014), 517–539. https://doi.org/10.1007/s00778-013-0335-9

[9] Buğra Gedik, Rajesh R Bordawekar, and S Yu Philip. 2009. CellJoin: a parallel stream join operator for the cell processor. *The VLDB journal* (2009).

[10] Vincenzo Gulisano. 2012. *StreamCloud: An Elastic Parallel-Distributed Stream Processing Engine.* Ph.D. Dissertation. Universidad Politécnica de Madrid.

[11] Vincenzo Gulisano, Yiannis Nikolakopoulos, Daniel Cederman, Marina Papatriantafilou, and Philippas Tsigas. 2017. Efficient Data Streaming Multiway Aggregation Through Concurrent Algorithmic Designs and New Abstract Data Types. *ACM Trans. Parallel Comput.* 4, 2, Article 11 (Oct. 2017), 28 pages. https://doi.org/10.1145/3131272

[12] Vincenzo Gulisano, Yiannis Nikolakopoulos, Marina Papatriantafilou, and Philippas Tsigas. 2016. ScaleJoin: a Deterministic, Disjoint-Parallel and Skew-Resilient Stream Join. *IEEE Trans. Big Data* PP, 99 (2016), 1–1. https://doi.org/10.1109/TBDATA.2016.2624274

[13] Vincenzo Gulisano, Yiannis Nikolakopoulos, Ivan Walulya, Marina Papatriantafilou, and Philippas Tsigas. 2015. Deterministic Real-time Analytics of Geospatial Data Streams Through ScaleGate Objects. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems (DEBS '15)*. ACM, New York, NY, USA, 316–317. https://doi.org/10.1145/2675743.2776758

[14] Vincenzo Gulisano, Marina Papatriantafilou, and Alessandro Vittorio Papadopoulos. 2018. Elasticity. In *Encyclopedia of Big Data Technologies*, Sherif Sakr and Albert Y. Zomaya (Eds.). Springer Int'l Conf. Publishing, Cham, 1–7. https://doi.org/10.1007/978-3-319-63962-8_191-1

[15] Thomas Heinze, Zbigniew Jerzak, Gregor Hackenbroich, and Christof Fetzer. 2014. Latency-aware Elastic Scaling for Distributed Data Stream Processing Systems. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems (DEBS '14)*. ACM, New York, NY, USA, 13–22. https://doi.org/10.1145/2611286.2611294

[16] Thomas Heinze, Yuanzhen Ji, Yinying Pan, Franz Josef Grueneberger, Zbigniew Jerzak, and Christof Fetzer. 2013. Elastic Complex Event Processing under Varying Query Load.. In *BD3@ VLDB*. 25–30.

[17] Thomas Heinze, Valerio Pappalardo, Zbigniew Jerzak, and Christof Fetzer. 2014. Auto-scaling techniques for elastic data stream processing. In *Data Engineering Workshops (ICDEW), 2014 IEEE 30th Int'l Conf. on*. IEEE, 296–302.

[18] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. 2014. A Catalog of Stream Processing Optimizations. *ACM Comput. Surv.* 46, 4, Article 46 (March 2014), 34 pages. https://doi.org/10.1145/2528412

[19] C. Hochreiner, M. Vögler, S. Schulte, and S. Dustdar. 2016. Elastic Stream Processing for the Internet of Things. In *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*. 100–107. https://doi.org/10.1109/CLOUD.2016.0023

[20] Yuanzhen Ji, Hongjin Zhou, Zbigniew Jerzak, Anisoara Nica, Gregor Hackenbroich, and Christof Fetzer. 2015. Quality-Driven Continuous Query Execution over Out-of-Order Data Streams. In *Proc. of the 2015 ACM SIGMOD Int'l Conf. on Management of Data*. ACM, 889–894.

[21] A. G. Kumbhare, Y. Simmhan, M. Frincu, and V. K. Prasanna. 2015. Reactive Resource Provisioning Heuristics for Dynamic Dataflows on Cloud Infrastructure. *IEEE Transactions on Cloud Computing* 3, 2 (April 2015), 105–118. https://doi.org/10.1109/TCC.2015.2394316

[22] Luo Mai, Kai Zeng, Rahul Potharaju, Le Xu, Steve Suh, Shivaram Venkataraman, Paolo Costa, Terry Kim, Saravanan Muthukrishnan, Vamsi Kuppa, et al. 2018. Chi: a scalable and programmable control plane for distributed stream processing systems. *Proc. VLDB Endowment* 11, 10 (2018), 1303–1316.

[23] André Martin, Andrey Brito, and Christof Fetzer. 2014. Scalable and elastic realtime click stream analysis using streammine3g. In *Proc. of the 8th ACM Int'l Conf. on Distributed Event-Based Systems*. ACM, 198–205.

[24] Ruben Mayer, Boris Koldehofe, and Kurt Rothermel. 2015. Predictable low-latency event detection with parallel complex event processing. *IEEE Internet of Things Journal* 2, 4 (2015), 274–286. https://doi.org/10.1109/JIOT.2015.2397316

[25] Pratanu Roy, Jens Teubner, and Rainer Gemulla. 2014. Low-Latency Handshake Join. *Proc. VLDB Endowment* (2014).

[26] S. Schneider, H. Andrade, B. Gedik, A. Biem, and K. Wu. 2009. Elastic scaling of data parallel operators in stream processing. In *2009 IEEE International Symposium on Parallel Distributed Processing*. 1–12. https://doi.org/10.1109/IPDPS.2009.5161036

[27] spark [n. d.]. Apache Spark. https://spark.apache.org. ([n. d.]). Accessed:2019-3-1.

[28] storm [n. d.]. Apache Storm. http://storm.apache.org. ([n. d.]). Accessed:2019-3-1.

[29] Jens Teubner and Rene Mueller. 2011. How soccer players would do stream joins. In *Proc. of the 2011 ACM SIGMOD Int'l Conf. on Management of data*.

[30] Ivan Walulya, Dimitris Palyvos-Giannas, Yiannis Nikolakopoulos, Vincenzo Gulisano, Marina Papatriantafilou, and Philippas Tsigas. 2018. Viper: A module for communication-layer determinism and scaling in low-latency stream processing. *Future Generation Computer Systems* 88 (2018), 297–308.

[31] Nikos Zacheilas, Vana Kalogeraki, Yiannis Nikolakopoulos, Vincenzo Gulisano, Marina Papatriantafilou, and Philippas Tsigas. 2017. Maximizing Determinism in Stream Processing Under Latency Constraints. In *Proceedings of the 11th ACM Int'l Conf. on Distributed and Event-based Systems (DEBS '17)*. ACM, 112–123. https://doi.org/10.1145/3093742.3093921

[32] N. Zacheilas, V. Kalogeraki, N. Zygouras, N. Panagiotou, and D. Gunopulos. 2015. Elastic complex event processing exploiting prediction. In *2015 IEEE International Conference on Big Data (Big Data)*. 213–222. https://doi.org/10.1109/BigData.2015.7363758