TENSILE: High-Throughput Reconfiguration of Streaming Pipelines

Satyam Jay
IIT, Delhi, India
anz238224@iitd.ac.in

Geetansh Juneja
IIT, Delhi, India
cs5200649@cse.iitd.ac.in

Abhilash Jindal
IIT Delhi, India
ajindal@cse.iitd.ac.in

Vatsal Jingar
IIT Delhi, India
cs5200449@iitd.ac.in

Kumar Madhukar
IIT Delhi, India
madhukar@cse.iitd.ac.in

Abstract

Streaming pipelines process unbounded inputs with low latency. These pipelines need to be reconfigured to react to changes, such as in input rate and hardware availability. Existing reconfiguration protocols block channels between pipeline operators, leading to loss of throughput. Due to the low pipeline throughput, data received during reconfiguration builds backlog, and experiences high latency because of queueing delays. After reconfiguration, this backlog of queued data needs to be processed first, before new data can be processed. Therefore, new data also experiences high latency. In this paper, we present a high-throughput reconfiguration protocol that skips blocking channels and consequently starts providing available throughput immediately. We have implemented our proposed protocol in a research prototype Tensile. Our experiments show that since Tensile starts providing high throughput immediately after reconfiguration is triggered, it reduces P99 latencies by up to 8.8x compared to existing reconfiguration protocols.

1 Introduction

Modern stream processing engines need to handle dynamic and unpredictable workloads. For example, consider a simple pipeline with two mapper instances followed by a join instance shown in Figure 1(a). To maintain low latency during a surge in the input rate, we might need to horizontally scale join. This involves (1) assigning some keys to a new join instance, (2) migrating the states for these keys from the old instance to the new one, and (3) updating the routing table of mappers so that the tuples of migrated keys start going to the new join instance.

Figure 1(b,c) show the ideal latency and throughput respectively of such a horizontal scaling (using an orange line). Say at ◯, the input load increases, shown by an increase in the dashed black line in Figure 1(c). The pipeline starts using idle resources and running at a higher throughput, shown as an increase in the orange line in Figure 1(c). But this throughput is insufficient for the applied load (orange line is lower than the dashed black line). Due to this, a backlog starts building and hence the latency starts increasing highlighted by Ⓑ in Figure 1(b). To improve the latency, the

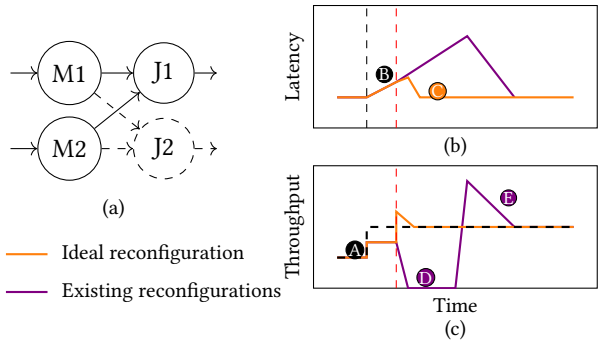


Figure 1. Latency and throughput during horizontal scaling. Existing protocols fall short of an ideal reconfiguration as their throughput drops during reconfiguration.

pipeline monitoring controller (automated or manual) starts scaling bottleneck operators at the red vertical dashed line in Figure 1(b,c). An ideal reconfiguration protocol scales up immediately to its increased capacity (the orange line immediately goes up in Figure 1(c)) and starts clearing the backlog (the orange line is higher than the load in Figure 1(c)). Once the backlog is cleared, the latency is quickly recovered at Ⓒ.

However, existing reconfiguration protocols do not provide high throughput during such reconfigurations. In fact the throughput of the system often deteriorates during the reconfiguration further exacerbating backlog build up and worsening latency.

Broadly, current reconfiguration protocols fall into two categories: stop-and-resume and on-the-fly reconfiguration with channel blocking. Apache Flink [8] exemplifies the former. It performs reconfigurations by suspending the pipeline at a consistent checkpoint and restarting it with the new configuration. During reconfiguration, input processing is entirely paused, resulting in zero throughput (purple line goes to zero at ◯ in Figure 1(c)). It takes a long time to finish a reconfiguration due to state reshuffling between operator instances [7]. After the pipeline resumes, it has to clear the large built-up backlog (shown by a large peak in throughput at Ⓔ in Figure 1(c)). Finally, it recovers the low latency.

On-the-fly reconfiguration protocols aim to avoid these complete pauses but still suffer from significant throughput degradation mainly due to delay caused by state transfer. For

instance, Chi [19] and others [10, 20] perform reconfiguration in two phases: a marker alignment phase, where control markers propagate through the pipeline to synchronize operators and update routing logic, followed by a state migration phase, where state is transferred between operator instances. During the state migration phase, new operator instances remain idle until the transfer is complete. While correctness is preserved—ensuring that events before and after reconfiguration are processed by the correct configurations—the throughput can drop significantly especially if state transfer time is large. This, in turn, leads to upstream backlog and increased latency.

Megaphone [15] introduces a fluid migration model that divides reconfiguration into smaller sub-reconfigurations, each transferring a subset of the state. While this avoids a single large throughput drop, it still causes multiple smaller throughput dips. Because of these throughput dips, backlog can still build up leading to slow reconfigurations and high latency. Several other on-the-fly reconfiguration protocols have been proposed which we discuss in §5.

The common drawback of existing protocols is that the new operator instances block their processing, waiting to receive the state, and therefore the pipeline operates at a lower throughput. This leads to upstream backlog and high latency. These limitations underscore the need for high-throughput reconfiguration protocols which are comparable with an ideal protocol where the pipeline starts running at a high throughput instantaneously (as was shown by the instantaneous rise of orange line in Figure 1(c)).

In this paper, we propose a novel high-throughput reconfiguration protocol that does not suspend processing during the state migration phase. Instead, the new operator instance starts building a local state from scratch. Later when it receives the state from the old instance, it just *merges* this state with its local state. This design prevents backlog buildup and mitigates the cascading effects of throughput drops. We show that our protocol enables pipelines to maintain low latency and high throughput even with large state transfers.

One of the main challenge is to show that with merging of states, our reconfiguration protocol is correct. In other words, the protocol shall provide identical outputs as a stop-and-resume reconfiguration, given identical inputs. For correctness, we identify sufficient conditions for state merging and show that common pipeline operators satisfy these conditions. Under these conditions, we formally prove that our protocol is correct. The proof uses a key observation that operators have evolved to do out-of-order processing [18] *i.e.,* if the operators are given the same input in any order, they still produce the same output (in some order).

Overall, this paper makes the following contributions:

- We highlight the limitations of existing reconfiguration protocols, demonstrating that their throughput drops during reconfiguration.
- We introduce a novel high-throughput reconfiguration protocol that, unlike existing reconfiguration protocols, does not block channels (§2).
- We formally define a state merging condition under which our proposed protocol is correct. We prove correctness and show that common operators satisfy the merging condition (§3).
- Our experiments show that Tensile provides significantly higher throughput during reconfiguration, and therefore reduces P99 latencies by up to 8.8x compared to existing reconfiguration protocols (§4).

2 High-Throughput Reconfiguration Protocol

To provide high throughput during reconfigurations, we relax the strict coordination and channel blocking imposed by existing protocols. Our protocol guarantees exactly-once processing and reconfiguration correctness (see §3).

The protocol proceeds in two distinct phases: (i) a *Non-Blocking Alignment Phase* and (ii) a *State Migration/Merging Phase*. We first give an overview of these phases and then illustrate the protocol with an example.

(i) Non-Blocking Alignment Phase: This phase is inspired by the reconfiguration marker-based alignment in existing on-the-fly reconfiguration protocols [19], but differs fundamentally in that it avoids blocking input channels upon receiving the markers. In other words, all operator instances—including both existing and new instances—continue processing incoming data and producing output. Since no alignment-based stalling occurs, this design avoids throughput degradation and backlog accumulation.

(ii) State Migration/Merging Phase: When an old operator instance, that is being scaled up, receives reconfiguration markers from every input channel, it initiates state migration. In this phase, it migrates states for selected keys to the new operator instances. Since we did not block the operators, the new operator instance would have built up its own local state in the meantime. The new operator instance *merges* the received state with its local state and subsequently outputs any pending tuples that result from this merging. When states for all keys are merged, reconfiguration is complete.

We now illustrate the protocol using an example pipeline in Figure 2 which joins two input streams—Left (L) and Right (R)—on a common key. For illustration simplicity, we assume that there are just two keys A and B and their input tuples are named $A_0, A_1, \cdots$ and $B_0, B_1, \cdots$ respectively. The pipeline has two mappers M_1 and M_2. For simplicity, M_1 produces input tuples for the L stream and M_2 for the R stream. Join instances maintain L and R tables for the tuples they have seen thus far. When they receive a new tuple, they join it with the appropriate table to produce the joined output, which we represent as A_1A_2 for example.

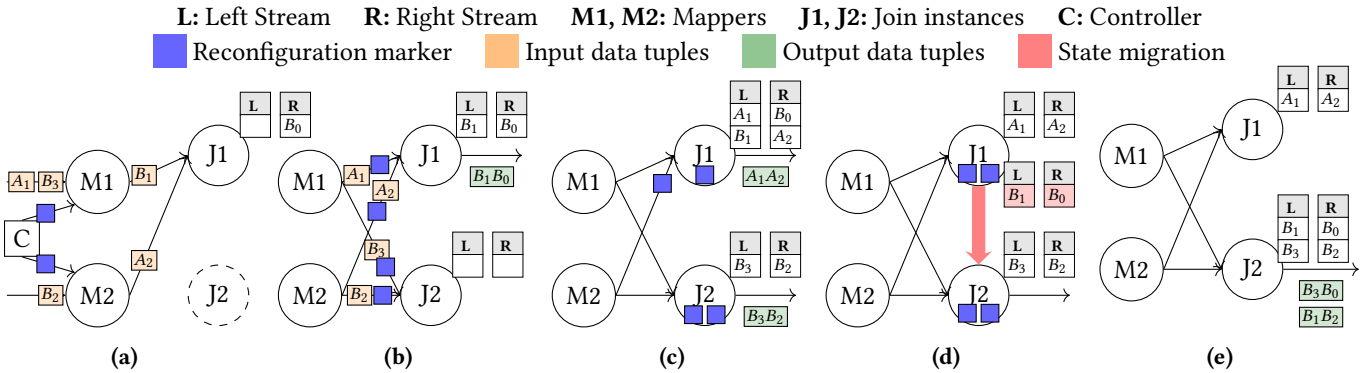


Figure 2. Illustration of our high-throughput reconfiguration protocol. (a) Controller C initiates upscaling by sending reconfiguration markers to M_1 and M_2. (b) M_1 and M_2 update their routing tables and forward reconfiguration markers. (c) Join instance J_1 continues processing input tuples *without* waiting for alignment of markers. More importantly, new join instance J_2 continues processing input tuples *without* waiting for migrated states. (d) After marker alignment, J_1 sends state to J_2. (e) J_2 merges received state with its local state and outputs pending tuples.

Initially, there is one join instance J_1. During reconfiguration, another join instance J_2 is added, and the join state corresponding to key B is migrated from J_1 to J_2. As shown in Figure 2(a), initially say J_1 has just one tuple B_0 in its R table and no tuples in its L table. Further, say tuples A_2 and B_1 are in-flight, shown on M_2–J_1 and M_1–J_1 edges respectively.

Step 1: In Figure 2(a), the controller initiates reconfiguration by spawning the new join instance J_2 and sending *reconfiguration markers* (blue squares) to mappers M_1 and M_2. These markers carry reconfiguration information: key B will migrate to J_2.

Step 2: In Figure 2(b), upon receiving the marker, both mappers update their routing tables and propagate the marker to both downstream join instances J_1 and J_2. From this point on, tuples with key B (*e.g.*, B_2 and B_3) are sent to J_2, while tuples with key A (*e.g.*, A_1) continue to be sent to J_1. From the point of view of M_1 and M_2, reconfiguration is complete.

Step 3: When a join instance receives a marker, it enters the *non-blocking alignment phase*. In Figure 2(c), even though J_1 has not received the marker from M_2, it continues processing A_1 from M_1 and outputs A_1A_2 (shown in a green box). More importantly, J_2 also does not block waiting to receive the B key state from J_1. Instead, J_2 immediately starts processing incoming tuples and outputs B_3B_2. This non-blocking behavior is the primary reason why the proposed protocol provides high throughput during reconfiguration.

Step 4: When J_1 receives reconfiguration markers on all its input channels (Figure 2(d)), the *non-blocking alignment phase* is complete. J_1 then starts the *state migration/merging phase*. The state associated with key B from L and R tables is migrated from J_1 to J_2 (denoted by the red arrow). From the point of view of J_1, reconfiguration is complete.

Step 5: Upon receiving the migrated state in Figure 2(e), J_2 starts the *state migration/merging phase*. J_2 merges the received state with its local state and emits pending joined tuples. As shown in Figure 3, for continuous joins, if the migrated state is (L_1, R_1) and the local state is (L_2, R_2), the final join state becomes $L = L_1 \cup L_2$ and $R = R_1 \cup R_2$. The corresponding pending join results are: $(L_1 \bowtie R_2) \cup (L_2 \bowtie R_1)$. In this example, two joined tuples B_3B_0 and B_1B_2 are produced by J_2. At this point, reconfiguration is complete. Both J_1 and J_2 continue normal operation on disjoint keys.

This example illustrates that the protocol avoids blocking input channels during both the alignment and migration phases, ensuring high throughput. Processing of tuples is only briefly blocked during state merging by the new instance; merging is CPU-bound and is therefore typically much faster than the network-bound state migration.

It can also be seen that the protocol provides exactly-once processing. For example, for the overall L inputs of A_1, B_1, B_3 and R inputs of A_2, B_0, B_2, the protocol correctly outputs (green boxes left to right in Figure 2) B_1B_0; A_1A_2; B_3B_2; B_3B_0 and B_1B_2. Moreover, the outputs are produced on the correct output channels: (1) A outputs are always produced by J_1; (2) B outputs for inputs before the reconfiguration are produced by J_1 (B_1B_0); (3) other B outputs are produced by J_2.

2.1 Continuous Joins

In §3, we will formally argue that the protocol always produces outputs that are consistent with a stop-and-resume reconfiguration. But first, we revisit the reconfiguration in Figure 2. Since keys are independent, it suffices to examine a single key that is being migrated from one instance to another. For a single migrated key, we observe how the state evolves and how the output tuples are produced by both the instances. Examining a single key independently will help in

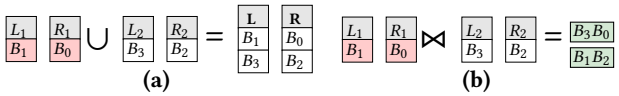


Figure 3. Merging states (∪) and outputting pending rows (⋈) in Tensile for a join operator.

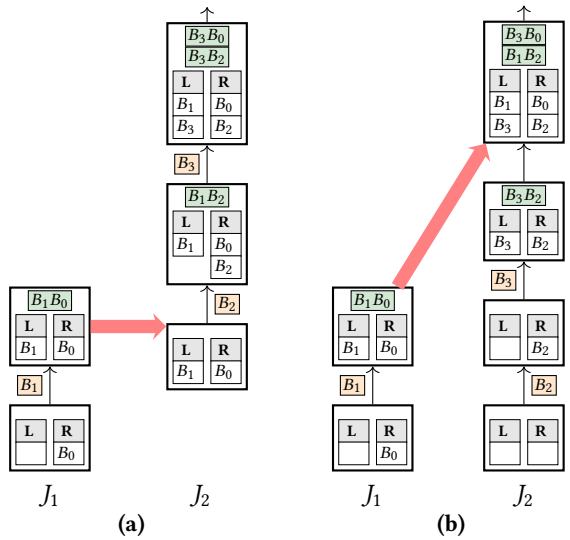


Figure 4. Comparing existing protocols with our proposed protocol for migrating a key in continuous joins. (a) In the existing protocols, new instance J_2 remains blocked until the state is received. (b) In the proposed protocol, J_2 does not block. It merges states to reach identical states (L and R tables) as in (a). Output tuples shown in green boxes are also identical in (a) and (b), produced in different orders.

presenting other operators as well as in developing a formal correctness proof.

Figure 4(a) shows how existing protocols migrate a single key B from J_1 to J_2 for a continuous join. It shows how states evolves (L and R tables) and outputs are produced (green boxes) for the two instances J_1 and J_2 with processing advancing from bottom to top. Each input tuple arrives either on the L stream or on the R stream (orange boxes shown on the left or right of each arrow). When an instance handles an input tuple, it might modify the state and output new tuples. For example, when J_1 handles the input tuple B_1 on the L stream, it outputs B_1B_0 and adds B_1 to its L table.

The old instance J_1 transfers the state (shown by the red arrow in the Figure) when the alignment phase completes, *i.e.,* when it has received reconfiguration markers from all its input channels. However in the existing protocols, new instances remain blocked until they receive the state from old instances. For example as shown in Figure 4(a), J_2 does not start processing input tuples until state migration is complete. As already discussed, this blocking of new instances loses available capacity and builds backlog.

In contrast in our proposed protocol shown in Figure 4(b), J_2 starts from an empty state, processes tuples B_2 and B_3, and then merges the incoming state with its current state to produce pending outputs B_3B_0 and B_1B_2 (recall Figure 3). It can be readily seen that J_2 reaches the same final state (tuples in L and R tables) in both reconfiguration protocols. Further, both J_1 and J_2 produce exactly the same output tuples, albeit in a different order, as in Figure 4(a).

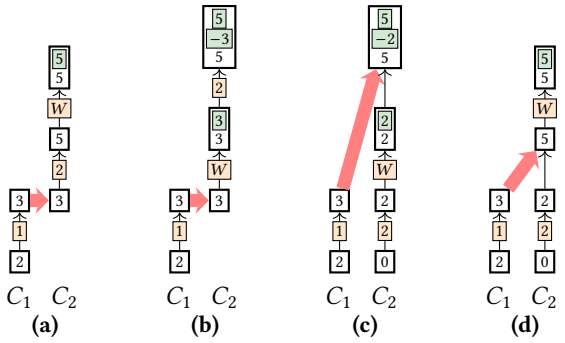


Figure 5. Reconfiguring a window sum operator. (a) Blocking reconfiguration with ideal watermarks. (b) Blocking reconfiguration with refinements for late data. (c) Non-blocking reconfiguration with refinements. (d) Non-blocking reconfiguration that delays watermarks.

2.2 Window Operators

Next, we illustrate the correct non-blocking reconfiguration of window operators. Here, we need to pay careful attention to the interaction between watermarks and reconfiguration. An ideal watermark guarantees that a tuple older than the watermark will not be received. A window operator produces the output for a window upon receiving a watermark later than the window's end time.

In deployments with arbitrarily delayed data, watermarks may not be ideal *i.e.,* they may not guarantee that a tuple older than the watermark will not be received [2, 3]. Upon receiving such late tuples, pipelines may either discard them or refine earlier results.

Consider a sliding window sum operator instance C_1 which currently contains the sums for windows 10–14 and 11–15. Upon receiving a tuple with t=12, C_1 updates the sums for both the windows. Now say a reconfiguration is triggered to migrate windows with odd closing-time *i.e.,* keep 10–14 but migrate 11–15 from C_1 to a new instance C_2. After this reconfiguration is triggered, the upstream operator updates its routing table (recall Figure 2(b)). A new tuple with say t=13 will be routed to both C_1 and C_2 since it is in both windows 10–14 managed by C_1 and 11–15 managed by C_2. C_1 can normally output the sum of 10–14 since it will receive all the tuples for this window.

We now discuss how the migrated window 11–15 gets processed. For brevity, we skip mentioning tuple timestamps arriving in the window, from the rest of this discussion. Figure 5(a) shows a blocking reconfiguration where C_2 remains blocked until it receives state from C_1. Because of this blocking, C_2 does not process any input tuples or watermarks. After C_2 receives the state, it first processes tuple 2 and then the watermark (denoted by W) that closes the window. In this simple example, C_2 outputs 5 as the window sum.

Figure 5(b) shows the same setup but now the watermark is not ideal. Here, C_2 is again blocked until it receives the state. But now C_2 first receives a watermark followed by the

late tuple 2. At watermark, C_2 outputs 3. At the late input tuple, C_2 refines its old results by retracting 3 (shown as -3) and outputting 5.[1]

In contrast, our proposed protocol allows C_2 to begin processing input tuples and watermarks without waiting for the corresponding state. This requires watermarks to be handled carefully. If watermarks were not ideal and if the pipeline expects refinements, no special handling of watermarks is needed. Figure 5(c) illustrates this. Here, C_2 starts from the initial state (sum of zero), processes an input tuple 2, followed by processing the watermark where it outputs 2. Later when it receives the state from C_1, it merges the received state with its local state by adding them together to 5. Further, it refines the results by retracting 2 and outputting 5.

If watermarks were ideal or if the pipeline did not expect refinements, we cannot allow C_2 to process watermarks before receiving the state. For example in Figure 5(c), C_2 will produce an erroneous output of 2 if it was not refined later. Figure 5(d) illustrates how our protocol handles this scenario. Here, C_2 again starts from the initial state and processes input tuples. But here the protocol delays processing watermarks until receiving and merging the state. We can observe that the final state and the output produced in Figure 5(d) is consistent with Figure 5(a).

In general, delaying watermark processing until merge, as in Figure 5(d), is sufficient for both ideal and non-ideal watermarks. Therefore, we will not discuss late data and refinements in the rest of the paper.

Summary: The key modification in the proposed reconfiguration protocol is to allow instances to start processing input tuples even before they have received the state. Doing so improves the throughput during reconfiguration and reduces backlog. When the instance eventually receives the state, it merges the received state with its local state using an operator-specific operation, *e.g.,* addition for window sum. Additionally, the instance also produces pending output tuples using the two states, like joining tables in joins. For correctness, watermark processing is delayed till merge.

The next section formalizes the correctness requirements for the merge operation and for producing pending outputs from the local and received state. Common operators satisfy these requirements.

3 Protocol Correctness

Streaming pipelines and their operators have evolved to do *out-of-order processing* [3, 18] since input tuples do not arrive ordered by their event timestamps due to differing network latencies of data sources, *e.g.,* different sensors. In other words, if streaming pipelines are given the same input tuples in any order, they produce the same output tuples (in some order). Because of this, we can represent both input and output tuples as *sets*.[2] In other words, under out-of-order processing semantics, the same *set* of input tuples lead to the same *set* of output tuples.

Table 1. Table of notations to describe reconfiguration of an abstract out-of-order processing operator.

Notation	Description
S	Set of states that the operator can have for each key.
In, Out	Set of input and output tuples.
$st_k \in S$, $t_k \in In$	State and input tuple for key k.
$S_{id} \in S$	Inital per-key state of the operator.
$+ : (S, In) \rightarrow S$	Operation to update the state when an input tuple is received.
$\circ : (S, In) \rightarrow \mathcal{P}(Out)$	Operation to output a set of tuples when an input tuple is received. $\mathcal{P}(Out)$ represents a power set of Out.
$\Diamond : S \rightarrow \mathcal{P}(Out)$	Operation to output a set of tuples when a watermark is received.
$\oplus : (S, S) \rightarrow S$	Operation to merge received and local states.
$\odot : (S, S) \rightarrow \mathcal{P}(Out)$	Operation to output a set of pending tuples from received and local states.

A reconfiguration is *output consistent* if it produces exactly the same set of output tuples as the stop-and-resume reconfiguration, given the same set of input tuples. Because of the semantics of out-of-order processing, the output tuples can be produced in a different order. Note that this definition implies exactly-once processing since stop-and-resume reconfiguration guarantees exactly-once processing.

A reconfiguration is *watermark consistent* if it necessarily outputs a tuple t before a watermark w whenever the stop-and-resume reconfiguration outputs t before w. Watermark consistency holds trivially for our proposed protocol since the protocol delays the processing of watermarks, whereas the input tuples are processed as they arrive. Further, watermarks are output only after all states are merged and pending output tuples are sent (as in Figure 5(d)). Therefore, we restrict our attention to proving output consistency.

The proposed protocol requires operators to specify: (1) a *merge* operation that merges received state with the local state and (2) a *pending* operation that produces pending output tuples by combining the received state with the local state. To show output consistency, we model all executions of an out-of-order operator as *paths on a semi-lattice*. We show that our proposed protocol eventually reaches identical lattice element as stop-and-resume reconfiguration, whenever the *merge* and *pending* operations bring the merged lattice element to the lowest-upper bound element in the lattice.

3.1 An Abstract Stateful Operator

We prove output consistency using an "abstract" operator. Say S represents the set of states that an abstract operator can keep for each key, and In and Out represent sets of input and output tuples. Abstractly, each out-of-order processing stateful operator implements three functions [18]. Table 1 summarizes the notations.

[1] This illustrates accumulating and retracting refinement. Other refinements like accumulating or discarding were also possible [3].

[2] To keep two tuples with the same data in a set (such as inputs to distinct), we assume that each tuple has some unique metadata like a tuple ID.

Algorithm 1 Out-of-order processing by an abstract stateful operator.

```
function INIT(par)
    for k ∈ par(id) do
        st_k ← S_id
function TUPLE_RECV(t_k)
    output st_k ∘ t_k
    st_k ← st_k+t_k
function WM_RECV(wm)
    for EXPIRY(k) ≤ wm do
        output ◊st_k
    broadcast wm
```

function INIT. Operator instances initialize their per-key state st_k to an initial value $S_{id} \in S$, for all the keys k in the key partition assigned to the instance.

function TUPLE_RECV. Upon receiving input tuple t_k with key k, the operator outputs tuples using its state and the input tuple (output $st_k \circ t_k$) via an abstract operation $\circ : (S, In) \rightarrow \mathcal{P}(Out)$, where $\mathcal{P}(Out)$ represent the power set of Out. The operator also uses t_k to update its state ($st_k \leftarrow st_k + t_k$) via another abstract operation $+ : (S, In) \rightarrow S$.

function WM_RECV. Upon receiving a watermark wm, window operators close windows, whose time is less than wm, and produce output tuples computed using an abstract operation $\Diamond : S \rightarrow \mathcal{P}(Out)$, applied on the expired window state. Operators then forward wm.

For example, consider the following implementation for a continuous join operator that joins left and right tuples, represented as l_k and r_k, and maintains left and right tables, represented as L_k and R_k, for key k.

- $S_{id} = (\{\}, \{\})$; Initialize two empty tables.
- $st_k \circ t_k = L_k \bowtie r_k$ or $l_k \bowtie R_k$; Join the tuple with the appropriate table and output.
- $st_k + t_k = L_k \cup \{l_k\}$ or $R_k \cup \{r_k\}$; Add the tuple in the appropriate table.
- $\Diamond st_k = \{\}$; Just forward the watermark.

As another example, consider the window sum operator that maintains a sum of tuples within a window, implemented as follows. Here the key k uniquely identifies a window. This operator outputs the window sum when the window is closed.

- $S_{id} = 0$; Initialize the sum with 0.
- $st_k \circ t_k = \{\}$; Do not produce any output when an input tuple is received.
- $st_k + t_k = st_k + t_k$; Update the sum.
- $\Diamond st_k = \{st_k\}$; Emit the sum when the window closes.

3.2 Out-of-Order Processing as a Semi-lattice

Since processing of tuples with one key does not affect processing of tuples with other keys, we can analyze operations on a single key in isolation, as we did in Section 2. For any given key, we would like to *represent out-of-order executions of an abstract operator as paths on a monotonic upper semi-lattice with a bottom element.*

An upper semi-lattice with a bottom element $(\mathbb{S}, \leq, \sqcup, \perp)$ has elements from $\mathbb{S}$ with a partial order $\leq$. The bottom element $\perp$ is less than all other elements $\forall s \in \mathbb{S} : \perp \leq s$. Any two elements $\{s_1, s_2\} \in \mathbb{S}$ have a unique lowest-upper bound *i.e.,* $s = s_1 \sqcup s_2 \in \mathbb{S}$ such that $s_1 \leq s$, $s_2 \leq s$, and $\forall s' \in \mathbb{S}, (s_1 \leq s' \wedge s_2 \leq s') \Rightarrow s \leq s'$. $\sqcup$ is idempotent, commutative, and associative. *Monotonic* upper semi-lattice with a bottom element $(\mathbb{S}, \leq, \sqcup, \perp, f)$ additionally has a function $f : \mathbb{S} \rightarrow \mathbb{S}$ such that $\forall s \in \mathbb{S} : s \leq f(s)$.

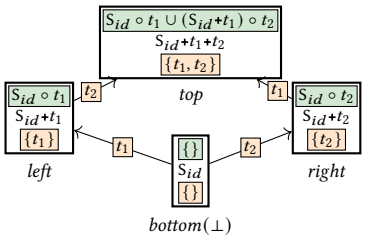


Figure 6. Each lattice element contains a set of input tuples (shown in orange boxes inside the element), a state, and a set of output tuples (shown in green boxes). When an operator processes an input tuple, it moves to a higher lattice element (shown as each arrow marked with an input tuple). Since processing input tuples in a different order leads to the same state and the same set of output tuples, out-of-order processing of tuples can be seen as different *paths* on this semi-lattice. For example, processing t_1 before t_2 or t_2 before t_1 leads to the same *top* lattice element.

For any given key, it is possible to represent out-of-order executions of an abstract operator as *paths* on a monotonic upper semi-lattice with a bottom element with elements from $\mathbb{S} : (\mathcal{P}(In), S, \mathcal{P}(Out))$ representing set of input tuples, state, and set of output tuples respectively. For example, Figure 6 shows the semi-lattice when an abstract operator (Algorithm 1) processes two input tuples t_1 and t_2. As shown, each lattice element has a set of input tuples (shown in orange boxes inside the element), a state, and a set of output tuples (shown in green boxes).[3]

The bottom element represents the beginning of the execution: no input and output tuples, initial operator state *i.e.,* $\perp = (\{\}, S_{id}, \{\})$ as shown in Figure 6. Each edge of the semi-lattice represents a processing step for a new input tuple: the new tuple becomes part of the input tuples, state is updated, and some output tuples are generated. This step can be thought of as an application of the monotonic function f of the semi-lattice. For example in Figure 6, each edge on the lattice is marked with an input tuple (t_1 or t_2). When t_1 is processed from the *bottom* element (edge to *left* element), the set of input tuples is updated to $\{t_1\}$, state is updated to $S_{id} + t_1$, and the set of output tuples is updated to $S_{id} \circ t_1$ (Algorithm 1).

Since processing a tuple is the monotonic function for the lattice, the $\leq$ ordering on this lattice can be defined as:

[3]This representation differs from the figures in Section 2. Here, the input tuples are also part of the lattice element, along with all output tuples *generated so far.*

$(I_1, S_1, O_1) \leq (I_2, S_2, O_2)$ iff $I_1 \subseteq I_2$. In other words, operators move higher in the lattice as they process more input tuples. For example, in Figure 6, processing t_1 moves the operator from the *bottom* element to a higher *left* element. The semi-lattice can be infinite, such as for continuous joins and distinct, or finite, such as for window operators.

We are able to represent state and output tuples within a common semi-lattice *because* all out-of-order executions are *equivalent*. Two different runs seeing identical input tuples out-of-order should produce identical output tuples (maybe out-of-order) and come to identical states [18]. Figure 6 shows out-of-order processing of tuples t_1 and t_2 leading to identical states and identical outputs (*top* lattice element).

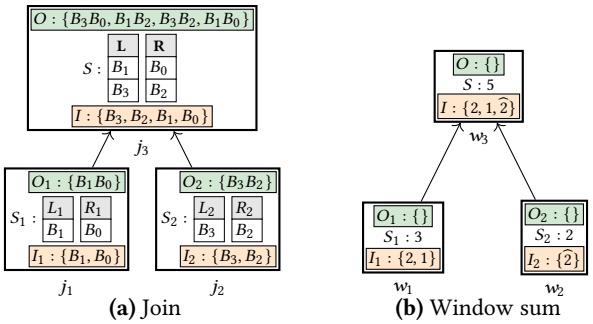
In other words, for a given set of input tuples, there is a *unique* lattice element. Therefore, the lowest upper bound $\sqcup$ of this semi-lattice can be given as: $(I, S, O) = (I_1, S_1, O_1) \sqcup (I_2, S_2, O_2)$ where $I = I_1 \cup I_2$, S is the state reached and O is the set of output tuples after processing all the input tuples in I. For example in Figure 6, the *top* element is the lowest-upper bound for the *left* and the *right* elements because $\{t_1, t_2\} = \{t_1\} \cup \{t_2\}$.

3.3 Conditions for Correct Reconfiguration

For our proposed reconfiguration protocol, the operator needs to specify two abstract operations: (1) that *merges* the received state with the local state, $\oplus : (S, S) \rightarrow S$, and (2) that produces *pending* output tuples by combining the received state with the local state, $\odot : (S, S) \rightarrow \mathcal{P}(\text{Out})$. We claim that our proposed reconfiguration protocol is output consistent if:

1. *merging* states S_1 and S_2 via $\oplus$ gives *exactly* the state S as in the lowest-upper bound of two lattice elements, *i.e.*, $(I, S, O) = (I_1, S_1, O_1) \sqcup (I_2, S_2, O_2)$ where $I_1 \cup I_2 = I$, $S_1 \oplus S_2 = S$, *and*
2. *pending* outputs obtained from S_1 and S_2 via $\odot$ gives *exactly* the remaining outputs $(O \setminus (O_1 \cup O_2))$ in the lowest-upper bound, *i.e.*, $S_1 \odot S_2 = O \setminus (O_1 \cup O_2)$.

Figure 7(a) shows the same example of join discussed in §2 as a semi-lattice. Recall from Figure 4(b), the instance J_1 received tuples B_1 and B_0. Therefore, J_1 is at the lattice element j_1 in Figure 7(a) with B_1 and B_0 in L and R tables and having output a B_1B_0 tuple. Similarly, instance J_2 is at the lattice element j_2, before it receives the state from J_1. The lowest-upper bound of j_1 and j_2 is shown as j_3 *i.e.*, $(I, S, O) = (I_1, S_1, O_1) \sqcup (I_2, S_2, O_2)$. In j_3, all the input tuples from both j_1 and j_2 have been received ($I = I_1 \cup I_2$), state S contains all these inputs in L and R tables, and join outputs O are produced (shown in the green box). As was shown in Figure 3, the merge operation $\oplus$ unions L and R tables from two states and the pending operation $\odot$ does a join between the two states. It can be seen that (refer Figure 3), $\oplus$ gives *exactly* the state in j_3, *i.e.*, $S_1 \oplus S_2 = (L_1 \cup L_2, R_1 \cup R_2) = (L, R) = S$. And $\odot$ gives *exactly* pending outputs (B_3B_0, B_1B_2) in j_3, *i.e.*, $S_1 \odot S_2 = (L_1 \bowtie R_2 \cup L_2 \bowtie R_1) = O \setminus (O_1 \cup O_2)$.



(a) Join **(b)** Window sum

Figure 7. Revisiting join and window sum reconfiguration examples from §2. After merging received state (at j_1 and w_1) with local state (at j_2 and w_2) and outputting pending tuples, we reach the lowest-upper bound lattice element (j_3 and w_3). Whenever we can define such a merge operation that reaches the lowest upper bound, the proposed reconfiguration protocol provides output consistency.

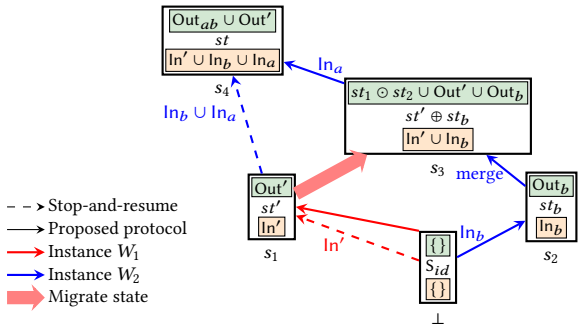


Figure 8. Output consistency: stop-and-resume reconfiguration protocol and the proposed protocol output identical tuples and reach identical states (see lattice element s_4).

Figure 7(b) shows the window sum example from §2.2.

3.4 Proof of Correctness

We can break all the input tuples for a single key into three disjoint sets: $\text{In} = \text{In}' \cup \text{In}_b \cup \text{In}_a$ where In': input tuples received before reconfiguration, In_b: input tuples received after reconfiguration but *before* merging states, and In_a: input tuples received after reconfiguration and *after* merging states. Sets In' and In_b are finite; In_a can be infinite. Say In' are received on an instance W_1 and $\text{In}_b \cup \text{In}_a$ are received on an instance W_2.

Stop-and-resume reconfiguration (follow dotted arrows in Figure 8) starts at the $\perp$ lattice element. W_1 first consumes *all* the before-reconfiguration tuples In' to come to a state, say st', and produces output tuples, say Out', reaching a lattice element, say s_1. Then the state is transferred to W_2 (not shown in the lattice). W_2 consumes the after-reconfiguration input tuples $\text{In}_b \cup \text{In}_a$ producing output tuples, say Out_{ab}, to reach lattice element, say s_4.

Under the proposed reconfiguration protocol (follow solid arrows in Figure 8), instance W_1 processes all the before-reconfiguration tuples In' to come to the same lattice element s_1 as stop-and-resume, i.e, it outputs Out' and has state st'. Then it transfers the state st' to W_2 (shown with a broad red arrow). In the meantime, instance W_2 processes In_b to produce outputs Out_b and comes to state st_b, reaching the lattice element s_2. Upon receiving the state st', it merges st_b with it to reach a lattice element s_3. Finally, it can consume In_a to reach the same lattice element s_4 as stop-and-resume, *i.e.,* having identical state and having outputted identical output tuples (maybe in a different order).

Note that each output tuple is produced by the correct worker: W_1 outputs Out'; W_2 outputs all the rest. The eventual equivalence of states and output tuples at the lattice element s_4 relies on two facts: (1) there is a *unique* lattice element for each set of input tuples and (2) $st_3 = st' \oplus st_b$ and $O_3 = st' \odot st_b \cup Out' \cup Out_b$ gives the lowest upper bound, *i.e.,* $s_3 = s_1 \sqcup s_2$. This is because of the requirement in §3.3: $(I, S, O) = (I_1, S_1, O_1) \sqcup (I_2, S_2, O_2)$ where $I = I_1 \cup I_2$, $S = S_1 \oplus S_2$, and $S_1 \odot S_2 = O \setminus \{O_1 \cup O_2\}$. The eventual equivalence of states and output tuples at the lattice element s_4 proves output consistency.

Due to page limit, we show the pseudocode for Tensile in Appendix B. In Appendix, we also show that both merge and pending operations can be easily defined for common operators Therefore, Tensile can support high-throughput reconfiguration for common pipelines, such as all of the NEXMark benchmark queries [5].

4 Evaluation

The primary objective of our proposed reconfiguration protocol is to achieve high-throughput during reconfigurations, thereby causing less impact on latency. For brevity, we call our proposed reconfiguration protocol as Tensile. We evaluate the following research questions:

- Is Tensile effective in providing high-throughput, when compared to existing reconfiguration protocols? (§4.2)
- What is the impact of high-throughput reconfiguration on aggregate latency and backlog in the system? (§4.3)
- Does maintaining local state in Tensile, before merging, add substantial memory overheads? How does the merging time compare against the state transfer time? (§4.4)
- During reconfiguration, is Tensile effective in utilizing CPU, memory, and network resources, compared to existing protocols? (§4.5)

4.1 Experimental Setup

Baselines: We compare Tensile against two state-of-the-art on-the-fly reconfiguration protocols: all-at-once reconfiguration [19, 20] and fluid reconfiguration [15].

Software: We implement Tensile in Flink. However, due to Flink being written in Java, we found that its resource utilizations are noisy. Flink's implementation of baselines was less flexible: they do not support upscaling by more than 1 operator instance, and does not support downscaling and migration. In this section, we use a 12k lines of Rust implementation instead. The Rust implementation is similar to other SPEs [7]. Key groups serve as the atomic units for redistributing keyed state across stateful operators' instances. We set the batch size to 128 for fluid reconfigurations. In Appendix C, we report Flink results, effectiveness for other types of reconfigurations, like migration and downscaling, and migration size sensitivity analysis for fluid reconfigurations.

Table 2. We benchmark Tensile using five NEXMark queries. For each query, we increase the input load by 3x, which is an unsustainable load for the original configuration. The reconfiguration scales up highlighted bottleneck operator by 3x more instances.

Query	Operator Graph	Scaling	Load (M/s)
Q3	Source → **Join**	1 → 4	0.43 → 1.72
Q5	Source → **WinAggr** → Aggr	2 → 8	1.7 → 6.8
Q6	Source → **Join**→ Aggr	1 → 4	1 → 4
Q7	Source → **WinAggr** → Aggr	2 → 8	1.38 → 5.52
Q8	Source → **WinJoin**	2 → 8	0.88 → 3.52

Hardware: We perform our evaluation on a cluster of four machines and a separate dedicated machine for the controller. Each machine has an Intel i9 CPU with 8 cores, 32 GiB of RAM and is running Ubuntu 24.04. The machines are interconnected over a 1 Gbps Ethernet network and synchronized via a common NTP server, with network RTT around 1 ms. For our evaluations, we spawn up to 8 Tensile workers on each of the four machines.

Workload: To evaluate our proposed protocol, we employ five queries from the NEXMark benchmark suite [29]: Q3, Q5, Q6, Q7 and Q8. We exclude Q1 and Q2 as they are stateless and therefore do not exercise the proposed stateful reconfiguration protocol. Q4 is omitted since it is a batch, not a streaming query. We generate NEXMark events using a generator. This setup is similar to other evaluations [10, 15, 17, 23]. We use 16 sources for all the queries, with 4 sources on each of the 4 machines. Operator graph for each query is shown in Table 2 with the operator that is being scaled up highlighted.

Reconfiguration: Each experiment runs for a total duration of 200 seconds and comprises of four distinct phases: (1) a 100-second warm-up phase to establish a steady state under a sustainable input rate; (2) a 3x increase in the load at the 100-second mark, exceeding the processing capacity of the currently allocated operator instances, as shown in Table 2; (3) a reconfiguration phase in which the bottleneck operator (highlighted in Table 2) is scaled up by 3x, initiated within 5 seconds of the load increase; and (4) a post-scaling stabilization phase to observe system behavior after reconfiguration. For each query, we start with either one or two instances of the bottleneck operator[4] on one machine and scale up by

[4]For Q3 and Q6, we start with 1 join instance. If we started with 2 join instances, source operators became bottleneck before the join instances when the input rate is increased by 3x. Note that filters are pushed down

3x to the other three machines. In all our queries since we are scaling the operators by 3x, it results in 768, out of 1024 total, key group transfers. This evaluation approach differs from previous evaluations in that the increase in input load is *unsustainable* for the starting configuration.

Measuring latency: For each experiment, we measure the latency using the technique described in [17]. In more detail, source operators inject current system timestamp in every tuple. And each downstream operator assigns to its output the maximum timestamp among all input tuples that contributed to producing it. At sink, we periodically sample end-to-end instantaneous latency by subtracting the system timestamp with the tuple timestamp. When the job finally finishes we also aggregate all the latencies measured during the run, to report latency percentiles of the job.

Measuring throughput: System throughput can be measured by just observing the source's throughput at each instant [17]. However, this approach fails to provide instantaneous throughput estimation under temporary backlog. For example, say if a downstream operator was blocked, the upstream source could still push tuples to its buffer giving a false measurement of system's instantaneous processing capacity. To prevent this, we use the following technique– we instrument the sources to periodically emit *throughput markers* after generating a fixed number of tuples (a microbatch). These markers propagate through the dataflow graph alongside normal data tuples. Upon receiving a throughput marker at the sink, we can infer that the corresponding microbatch has been fully processed. For the throughput markers received in a one-second window, we sum the number of tuples in their microbatches to get the instantaneous throughput. Now if some operator is blocked, the throughput markers will also get blocked, and we will correctly observe a lower instantaneous throughput.

4.2 Reconfiguration Performance

Figure 9 shows instantaneous latency (top) and throughput (bottom) of the system while reconfiguring four queries from NEXMark benchmark with the three different protocols (Q7 results are similar to Q5 and we hence omit it). The red vertical line is the time at which reconfiguration is initiated. The black dotted line in the bottom figure represents the applied load on the system. Note that before the load increases, the system's throughput is equal to the applied load (*i.e.,* the load is sustainable). However after the load increases, system is unable to keep up with the load (load is unsustainable) and we see rise in latency (from 100 seconds to red line). Latency eventually stabilizes again after load becomes sustainable due to reconfiguration. We will now analyze the performance of each NEXMark query individually after the reconfiguration is initiated. We discuss queries in the order of migrated state's size.

NEXMark Q3 performs a join between the auction and person streams using seller ID from auctions as the join key. In this query, state size is large (~2GB) and hence the transfer time is large. In all-at-once reconfiguration, the new instances have to wait for the state to arrive, before they can start processing new tuples. Hence as shown in Figure 9(a), the system's throughput is below the applied load until the state arrives at E inducing high latency of 12s at ◯. The high throughput increase (higher than the load) that we see after the reconfiguration at F is because the system is clearing the backlog that was created before and during the reconfiguration (while the throughput was lower than the applied load).

Fluid reconfiguration tries to mitigate this issue by migrating only a fraction of the key groups at a time, allowing new instances to gradually take over the workload instead of remaining idle, at the cost of increased reconfiguration completion time. However, in scenarios where the load becomes unsustainable, it is preferable to complete the reconfiguration as quickly as possible to recover maximum throughput. Indeed, we see that fluid reconfiguration performs poorly: it completes the state migration at ◯, after experiencing several throughput dips, with peak latency of 11.2s at B.

In Tensile, the new instances start processing immediately and do not wait for the state. We see that the throughput increases as soon as the new instance is spawned at ◯. The throughput is briefly higher than the applied load since the system is clearing the backlog built during the unsustainable load before the reconfiguration was triggered. The peak latency is of 3s just before the reconfiguration is triggered; the latency drops quickly at C.

NEXMark Q8 performs a windowed join between the person and auction streams. As shown in Figure 9(b), the latency and throughput trends observed for this query are similar to those in Q3, albeit not as large since the state size in Q8 is ~500MB. The peak latency for all-at-once, fluid and Tensile is 7.6s, 9.4s and 1.7s respectively. We again see that Tensile is able to recover the throughput immediately at ◯.

NEXMark Q6 computes the average selling price per seller over their last ten closed auctions. This query also transfers ~500MB of state during reconfiguration. From Figure 9(c), we see that Tensile brings down the latency quickly at I compared to all-at-once and fluid at J and K respectively. This is again because the throughput of Tensile is recovered immediately after the reconfiguration at L.

NEXMark Q5 performs a windowed aggregation over the bid stream and outputs the auction ID with the highest number of bids within a sliding window. As shown in Figure 9(d), neither of the reconfiguration protocols have large latency peaks. This is because the state transfer size in this case is small (~10MB). Therefore, unlike Q3, Q6, and

into the sources for all the queries. Therefore, the source operator actually has to serialize and send different number of tuples, depending on the filters' selectivity. For example in Q3, while sources generate 80M/s tuples, they only send 1.72M/s tuples to the join instances, as shown in Table 2. Whereas in Q6, sources generate only 4.5M/s tuples, but they send 4M/s tuples to the join instances.

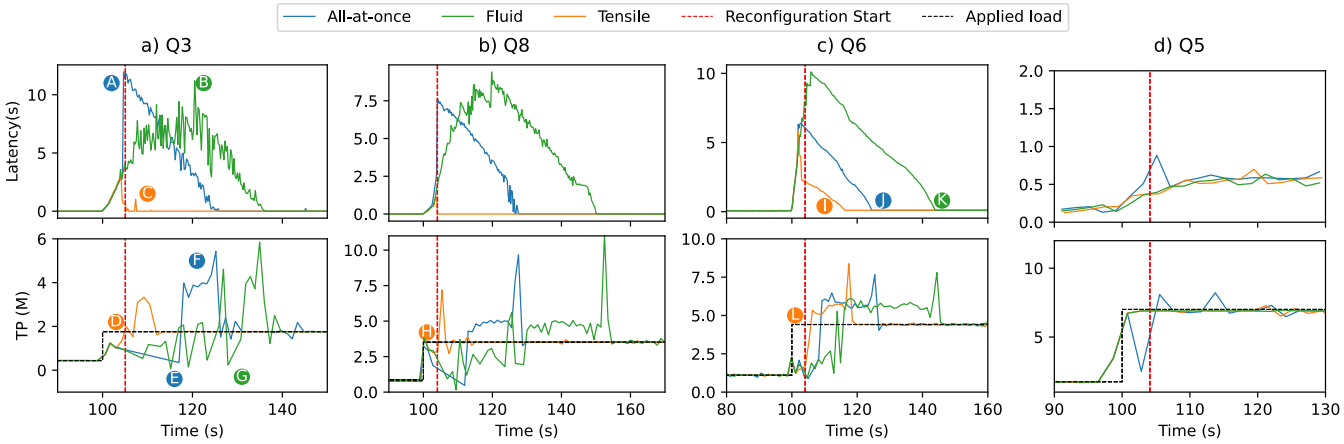


Figure 9. Each subfigure shows latency (top), and throughput (bottom) over time for Q3, Q6, Q8, and Q5 under reconfiguration. The black line in throughput represents the input load, which is made 4x at the 100-second mark. The red dashed vertical line marks the point at which reconfiguration is triggered. TENSILE exhibits lower latency spikes, faster latency recovery, and faster throughput recovery across all queries, when compared to all-at-once and fluid reconfigurations.

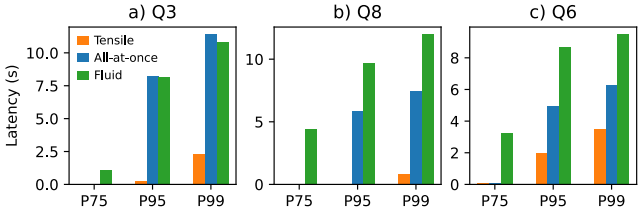


Figure 10. P75, P95, and P99 latency comparison for Q3, Q8, and Q6, across the three reconfiguration protocols. TENSILE consistently achieves the lowest P95 and P99 latency.

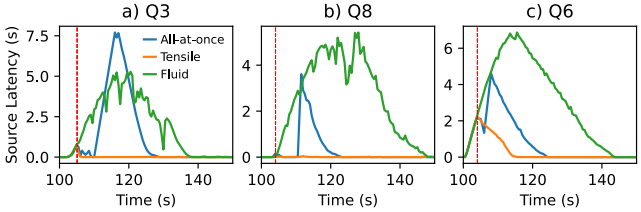


Figure 11. Source operator latency over time for Q3, Q8, and Q6. TENSILE consistently exhibits the lowest source latency, indicating minimal backlog and faster system recovery.

Q8, the operator instances are not blocked for a long time waiting to receive states. Q7, similarly, transfers small state and exhibits similar behavior to Q5. We omit it for brevity.

Overall, we observe that TENSILE *can start providing high throughput soon after reconfiguration is initiated. This is because the new operator instances are not blocked waiting for states. Hence,* TENSILE *provides the most performance advantage when the migrated state is large. Because of its non-blocking reconfigurations,* TENSILE *exhibits lower peak latency and is able to recover low latency sooner than baselines.*

4.3 Overall Latency and Backlog Buildup

Overall latency: While Figure 9 (top) showed improvements in instantaneous latencies, we further quantify the overall improvements in latency over the 200s runs of the queries. Figure 10 shows the 75th, 95th, and 99th percentile latencies for Q3, Q8, and Q6. It shows that TENSILE reduces latencies, particularly at higher percentiles, compared to both the baselines. For instance in Q3, TENSILE keeps the P99 (P95) latency below 2s (0.3s), while the baselines have them above 10s (8s). Similarly, TENSILE achieves 1.8x (2.6x) and 8.8x (94x) lower P99 (P95) latencies compared to the best-performing baselines in Q6 and Q8 respectively.

Backlog buildup: When downstream operators are blocked or are unable to process tuples at the rate at which they are produced, the buffers in the system become full. When sources' outgoing buffers become full, they start getting blocked and generate tuples later than their correct time. This difference between tuple's expected generation time and actual generation time is the system's backlog. We plot this in Figure 11 for Q3, Q8, and Q6 undergoing reconfiguration. The Figure shows that TENSILE experiences the least backlog buildup during reconfiguration and mitigates any existing backlog quickly. In contrast, all-at-once and fluid reconfigurations exhibit significantly higher backlog, since new operator instances do not process incoming tuples until the corresponding migrated state has been fully received.

Overall, we see that TENSILE *(i) lowers P99 and P95 latencies by up to 8.8x and 94x respectively, and (ii) builds much less backlog in the system compared to baselines.*

4.4 Overhead Analysis

New TENSILE operator instances first build local state, which is later merged with the migrated state. During the merge process, the operator pauses processing of incoming tuples. To avoid a costly, all-at-once merge, state migration is done at the granularity of key groups. As soon as the state for a key group arrives, it is merged with the corresponding local state. Before merging any key group, we measure the size of its unmerged local state.

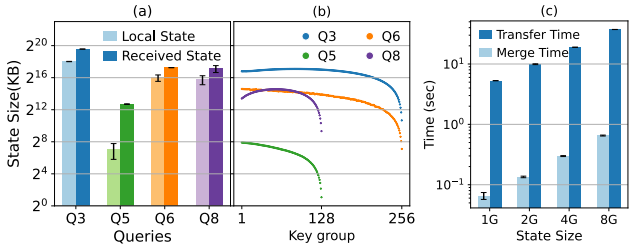


Figure 12. Analyzing merge operation. (a) Size of total local state before merging compared to the size of total received state. (b) Instantaneous local state size at the time of merging each key group. (c) Time taken to receive states compared to the time taken to merge states.

We define *instantaneous local state size* as the total size of all unmerged local states across key groups at a specific point in time. We define *total local state size* as the cumulative extra state accumulated by the operator over the entire reconfiguration period.

In this section, we evaluate three questions. (1) How does the total local state size compare to the migrated state size? (2) What is the instantaneous memory overhead from the instantaneous local state? (3) How does the state merging time compare with the state transfer time?

(1) Figure 12(a) compares the total local state size with the received state size by a new operator instance during rescaling in Tensile. We see that the received state is ~2x, ~16x, ~4x and ~16x larger than the local state for Q3, Q5, Q6 and Q8 respectively. In Q3, local size goes as high as 256MB.

(2) Figure 12(b) shows the instantaneous local state size at the time of merging i^{th} key group for each query. For Q5 and Q8, a new instance merges 128 key groups since there are 6 new instances (refer Table 2), among which 768 key groups are distributed. Whereas Q3 and Q6 start 3 new instances, hence each instance merges 256 key groups. For Q3, even though the total local state size is ~256MB (Figure 12(a)), highest instantaneous local state size is ~130MB since each key group's local state gets merged and cleaned up as soon as its migrated state is received.

(3) To contrast merge time with state transfer time with different received state sizes, we modify Q3 as follows. Recall that when we increase the input load at 100 seconds, Q3 transfers ~2GB of state. We change the input load increase time from 100 seconds to 50 seconds, 200 seconds, and 400 seconds to build ~1GB, ~4GB, and ~8GB of transferred state respectively. In Figure 12(c), we observe that both the state merging time and the state transfer time scales linearly with the size of the migrated state. However, transfer time is always substantially larger (~100x) than merging time since state transfer is primarily an I/O bound task while merging is CPU bound.

Overall, we conclude that (1) local state size is 2x-16x lesser than the received state size, (2) local state gets gradually cleaned up as key group states are merged, reducing instantaneous local state size, and (3) CPU-bound state merging is much faster than the IO-bound state transfer. Because of (1) and (2), as we will also show next, Tensile *does not require much more instantaneous memory than existing protocols. Because of (3), new* Tensile *operator instances block processing of input tuples for a much shorter time than existing reconfiguration protocols.*

4.5 Resource Utilization

We next compare CPU, memory, and network utilization of Tensile with our baselines when reconfiguring Q3. We evaluate the resource usage for two representative machines: the machine hosting the join instance before reconfiguration (referred to as the old machine), and a machine where a new join operator instance is created during reconfiguration (referred to as the new machine). We read */proc/meminfo*, */proc/stat* and */sys/class/net/enp2s0/statistics/* at 250ms interval to measure instantaneous memory, CPU and network respectively. Figure 13 shows the instantaneous resource utilizations. We mark three regions in the Figure: the red region shows the duration between load increase and reconfiguration start; the yellow region shows the reconfiguration duration, and the green region shows the post-reconfiguration duration.

Memory: For the new worker during fluid reconfiguration (◯ in Figure 13(a)), we observe 6-step increases in memory utilization. corresponding to 6 sub-reconfiguration. Whereas, the memory increases in one step for all-at-once (B) and Tensile (C). Since, Tensile merges local states as and when key group states are received, we do not observe any noticeable increase in required memory, compared to baselines[5].

CPU: Figure 13(b) shows that at 100 seconds, the system load increases at ◯, causing the old machine's CPU usage to spike to nearly 100%. The new machine's CPU utilization also increases slightly as it also hosts source operators. During the reconfiguration phase (yellow region), Tensile's new machine CPU usage immediately jumps to 100% at (E), since the new operator instance begins processing incoming tuples right after initialization, without waiting for the full migrated state. In contrast, the all-at-once protocol shows a delayed CPU ramp-up (F), only reaching full utilization after the entire state transfer completes at ◯. Meanwhile, fluid reconfiguration takes the longest to ramp up CPU usage, increasing gradually with sub-reconfigurations ◯.

Network: In all-at-once protocol, we observe that during reconfiguration (yellow region in Figure 13(c) top), the new machine is only receiving state (In) (I). Its CPU usage is low, and its output rate is 0 (Out) (J). In fluid reconfiguration, state is sent 6 times (K). In Tensile, we see that the new machine is simultaneously receiving state over the network and processing new tuples in parallel (high In and non-zero Out at (L)).

[5]Note that for the old machine, we do not see the memory reducing even though it has transferred some of its state. This is because the Rust memory allocator did not return memory to the OS immediately after it is freed. Therefore, after reconfiguration, the old machine memory also does not grow (for the time shown in the Figure).

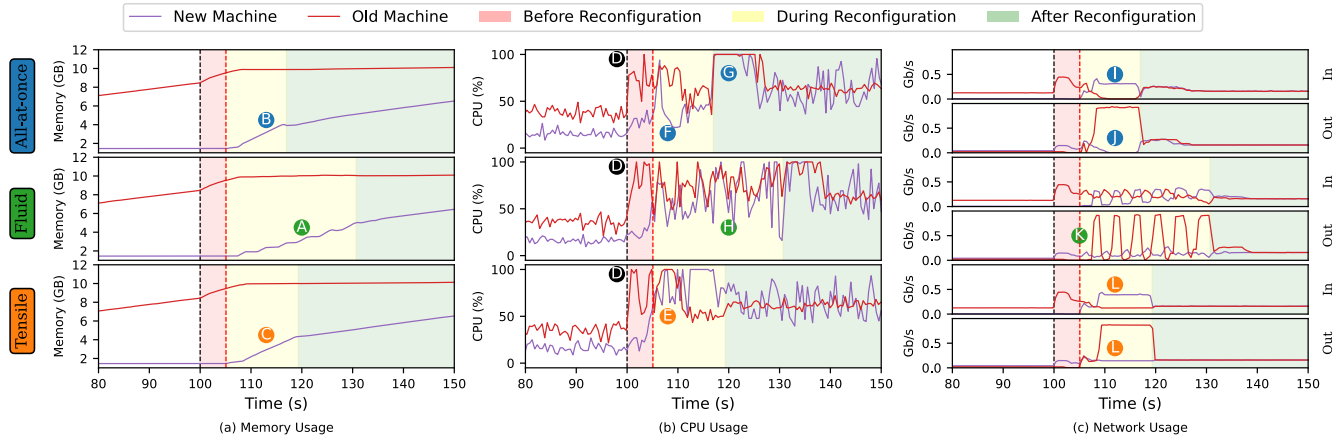


Figure 13. Comparison of memory, CPU, and network usage across reconfiguration protocols. (a) Memory usage shows that despite additional local state maintenance on new machine, Tensile's memory footprint remains comparable to baselines. (b) CPU usage during reconfiguration reveals how Tensile immediately start processing on new machine. (c) Network usage during reconfiguration shows multiple state transfer bursts in fluid, and one state transfer burst in both Tensile and all-at-once.

Overall, Tensile*'s memory overhead is negligible, compared to baselines. By overlapping state transfer with tuple processing,* Tensile *is able to utilize both network and CPU simultaneously for new workers. In contrast, existing protocols under-utilize CPU on new workers while waiting for state transfer.*

5 Related Work

There has been extensive work in managing state such that the stream processing engines can recover from crashed or slow workers [1, 9, 24, 32]. Stop-and-resume reconfiguration [4, 7, 31] grew from these works, where the pipeline is reconfigured using the last checkpoint. Reconfiguration decisions can either be triggered manually or by automatic scaling controllers [11, 16] which monitor pipelines to quickly find the right reconfigurations for maintaining performance under varying workloads. Instead of relying on checkpoints, several faster on-the-fly reconfigurations [6, 10, 13–15, 19, 20, 22, 27] have been proposed that directly migrate state between operator instances. Tensile is another on-the-fly reconfiguration protocol that provides higher throughput during reconfigurations.

Meces [12] performs *prioritized* state migration, *i.e.,* migrate states in the order in which tuples arrive to unblock their processing. However, Meces requires adding and managing an external key-value store during migrations, through which, the state migrates back-and-forth between old and new instances. In contrast, Tensile unblocks processing of tuples immediately without waiting for migrated states. It does not need an external key-value store and does not migrate states back-and-forth.

Optimizations in Rhino [10] are complementary to Tensile. Rhino optimizes reconfigurations with huge state (*e.g.,* over 100GB), by modifying how distributed file systems store checkpoints. During reconfigurations, it starts new operator instances on the file system nodes already containing the state from a previous checkpoint. At the time of reconfiguration, only the updates after the checkpoint needs to be transferred. However, as shown in [12], the updates after the previous checkpoint can also be large. Tensile will provide higher throughput while migrating these updates.

DRRS [25] is a recent work that (1) reduces blocking during the marker alignment phase by prioritizing messages in buffers, and (2) transfers states in a more fine-grained manner than fluid reconfigurations [15]. However, operator instances still need to wait for migrated states. In contrast, Tensile removes alignment delays by making alignment phase non-blocking. As our evaluation with unsustainable input rate shows (Figure 9), fluid reconfigurations can actually perform worse than all-at-once reconfigurations since fluid reconfigurations recover processing capacity more slowly.

Other reconfiguration protocols [13, 23, 26, 30] run the old and the new configuration in parallel until the migration is complete. However, this replication-based mechanism is not cost-efficient and not work conserving. For example, Chronostream [30] needs to do input duplication, output de-duplication, and requires $m + n$ workers to reconfigure from m workers to n workers. Falcon [23] removes output de-duplication and the need for $m + n$ workers, but it still does input duplication making it ill-suited for high throughput pipelines. For example, Falcon was evaluated with only 1500 tuples per second throughput for edge-cloud pipelines, whereas reconfiguration of large scale pipelines are evaluated with several million tuples per second throughput, as in Tensile and others [10, 15, 17].

We represent out-of-order executions as different paths on a semi-lattice, inspired from vector clocks [21]. Our output consistency proof is inspired from the eventual consistency proof for convergent replicated data types [28].

6 Conclusion

In this paper, we present a high-throughput reconfiguration protocol that starts processing input tuples even before the state migration is complete. We show that common stateful operators can be correctly reconfigured due to their out-of-order processing capability. Our experiments show that compared to existing on-the-fly reconfiguration mechanisms, Tensile can start providing high throughput almost immediately after a reconfiguration is triggered. Because of this, it reduces P99 latencies by up to 8.8x compared to existing reconfiguration protocols.

References

[1] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. 2013. MillWheel: fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment* 6, 11 (Aug. 2013), 1033–1044. https://doi.org/10.14778/2536222.2536229

[2] Tyler Akidau, Edmon Begoli, Slava Chernyak, Fabian Hueske, Kathryn Knight, Kenneth Knowles, Daniel Mills, and Dan Sotolongo. 2021. Watermarks in stream processing systems: semantics and comparative analysis of Apache Flink and Google cloud dataflow. *Proceedings of the VLDB Endowment* 14, 12 (July 2021), 3135–3147. https://doi.org/10.14778/3476311.3476389

[3] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. 2015. The dataflow model: a practical mechanism to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment* 8, 12 (Aug. 2015), 1792–1803. https://doi.org/10.14778/2824032.2824076

[4] Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, and Matei Zaharia. 2018. Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark. In *Proceedings of the 2018 International Conference on Management of Data*. ACM, Houston TX USA, 601–613. https://doi.org/10.1145/3183713.3190664

[5] NEXMark Benchmark. 2002. NEXMark: A Benchmark for Queries over Data Streams. http://datalab.cs.pdx.edu/niagaraST/NEXMark.

[6] Alain Biem, Eric Bouillet, Hanhua Feng, Anand Ranganathan, Anton Riabov, Olivier Verscheure, Haris Koutsopoulos, and Carlos Moran. 2010. IBM infosphere streams for scalable, real-time, intelligent transportation services. In *Proceedings of the 2010 ACM SIGMOD* (Indianapolis, Indiana, USA) *(SIGMOD '10)*. Association for Computing Machinery, New York, NY, USA, 1093–1104. https://doi.org/10.1145/1807167.1807291

[7] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. 2017. State management in Apache Flink®: consistent stateful distributed stream processing. *Proceedings of the VLDB Endowment* 10, 12 (Aug. 2017), 1718–1729. https://doi.org/10.14778/3137765.3137777

[8] Paris Carbone, Asterios Katsifodimos, † Kth, Sics Sweden, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink™: Stream and Batch Processing in a Single Engine. *IEEE Data Engineering Bulletin* 38 (01 2015).

[9] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. 2013. 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* (New York, New York, USA) *(SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 725–736. https://doi.org/10.1145/2463676.2465282

[10] Bonaventura Del Monte, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2020. Rhino: Efficient Management of Very Large Distributed State for Stream Processing Engines. In *Proceedings of the 2020 ACM SIGMOD*. ACM, Portland OR USA, 2471–2486. https://doi.org/10.1145/3318464.3389723

[11] Avrilia Floratou, Ashvin Agrawal, Bill Graham, Sriram Rao, and Karthik Ramasamy. 2017. Dhalion: self-regulating stream processing in heron. *Proceedings of the VLDB Endowment* 10, 12 (Aug. 2017), 1825–1836. https://doi.org/10.14778/3137765.3137786

[12] Rong Gu, Han Yin, Weichang Zhong, Chunfeng Yuan, and Yihua Huang. 2022. Meces: Latency-efficient Rescaling via Prioritized State Migration for Stateful Distributed Stream Processing Systems. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 539–556. https://www.usenix.org/conference/atc22/presentation/gu-rong

[13] Vincenzo Gulisano, Ricardo Jimenez-Peris, Marta Patino-Martinez, Claudio Soriente, and Patrick Valduriez. 2012. StreamCloud: An Elastic and Scalable Data Streaming System. *IEEE Trans. Parallel Distrib. Syst.* 23, 12 (Dec. 2012), 2351–2365. https://doi.org/10.1109/TPDS.2012.24

[14] Thomas S. Heinze, Yuanzhen Ji, Lars Roediger, Valerio Pappalardo, , Phillip Meister, Zbigniew Jerzak, and Christof Fetzer. 2015. FUGU: Elastic Data Stream Processing with Latency Constraints. *IEEE Data Eng. Bull.* 38 (2015), 73–81. https://api.semanticscholar.org/CorpusID:11509723

[15] Moritz Hoffmann, Andrea Lattuada, and Frank McSherry. 2019. Megaphone: latency-conscious state migration for distributed streaming dataflows. *Proceedings of the VLDB Endowment* 12, 9 (May 2019), 1002–1015. https://doi.org/10.14778/3329772.3329777

[16] Vasiliki Kalavri, John Liagouris, Moritz Hoffmann, Desislava Dimitrova, Matthew Forshaw, and Timothy Roscoe. 2018. Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows. In *Proceedings of the 13th USENIX OSDI* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 783–798.

[17] Jeyhun Karimov, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen, and Volker Markl. [n.d.]. Benchmarking Distributed Stream Data Processing Systems. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)* (2018-04). 1507–1518. https://doi.org/10.1109/ICDE.2018.00169

[18] Jin Li, Kristin Tufte, Vladislav Shkapenyuk, Vassilis Papadimos, Theodore Johnson, and David Maier. 2008. Out-of-order processing: a new architecture for high-performance stream systems. *Proceedings of the VLDB Endowment* 1, 1 (Aug. 2008), 274–288. https://doi.org/10.14778/1453856.1453890

[19] Luo Mai, Kai Zeng, Rahul Potharaju, Le Xu, Steve Suh, Shivaram Venkataraman, Paolo Costa, Terry Kim, Saravanan Muthukrishnan, Vamsi Kuppa, Sudheer Dhulipalla, and Sriram Rao. 2018. Chi: a scalable and programmable control plane for distributed stream processing systems. *Proceedings of the VLDB Endowment* 11, 10 (June 2018), 1303–1316. https://doi.org/10.14778/3231751.3231765

[20] Yancan Mao, Yuan Huang, Runxin Tian, Xin Wang, and Richard T. B. Ma. 2021. Trisk: Task-Centric Data Stream Reconfiguration. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SoCC '21)*. Association for Computing Machinery, New York, NY, USA, 214–228. https://doi.org/10.1145/3472883.3487010

[21] Friedemann Mattern et al. 1988. *Virtual time and global states of distributed systems.* Univ., Department of Computer Science.

[22] Matteo Migliavacca, David Eyers, Jean Bacon, Yiannis Papagiannis, Brian Shand, and Peter Pietzuch. 2010. SEEP: Scalable and elastic event processing. (11 2010), 4. https://doi.org/10.1145/1930028.1930032

[23] Pritish Mishra, Nelson Bore, Brian Ramprasad, Myles Thiessen, Moshe Gabel, Alexandre Da Silva Veith, Oana Balmau, and Eyale De Lara. 2024. Falcon: Live Reconfiguration for Stateful Stream Processing on the Edge. In *2024 IEEE/ACM Symposium on Edge Computing (SEC)*. 234–248. https://doi.org/10.1109/SEC62691.2024.00026

[24] Shadi A Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H Campbell. 2017. Samza: stateful scalable stream processing at LinkedIn. *Proceedings of the VLDB Endowment* 10, 12 (2017), 1634–1645.

[25] Yunfan Qing and Wenli Zheng. 2025. Towards Fine-Grained Scalability for Stateful Stream Processing Systems. arXiv:2503.11320 [cs.DC] https://arxiv.org/abs/2503.11320

[26] Sumanaruban Rajadurai, Jeffrey Bosboom, Weng-Fai Wong, and Saman Amarasinghe. 2018. Gloss: Seamless Live Reconfiguration and Reoptimization of Stream Programs. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Williamsburg, VA, USA) *(ASPLOS '18)*. Association for Computing Machinery, New York, NY, USA, 98–112. https://doi.org/10.1145/3173162.3173170

[27] Mehul Shah, Joseph Hellerstein, Sirish Chandrasekaran, and Michael Franklin. 2003. Flux: An Adaptive Partitioning Operator for Continuous Query Systems. *Proceedings - International Conference on Data Engineering*, 25–36. https://doi.org/10.1109/ICDE.2003.1260779

[28] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-free replicated data types. In *Stabilization, Safety, and Security of Distributed Systems: 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings 13*. Springer, 386–400.

[29] P. Tucker, K. Tufte, V. Papadimos, and D. Maier. 2002. *NEXMark—A Benchmark for Queries over Data Streams*. Technical report. OGI School of Science & Engineering at OHSU. Draft.

[30] Yingjun Wu and Kian-Lee Tan. 2015. ChronoStream: Elastic stateful stream computation in the cloud. In *2015 IEEE 31st International Conference on Data Engineering*. 723–734. https://doi.org/10.1109/ICDE.2015.7113328

[31] Le Xu, Boyang Peng, and Indranil Gupta. 2016. Stela: Enabling Stream Processing Systems to Scale-in and Scale-out On-demand. In *2016 IEEE International Conference on Cloud Engineering (IC2E)*. 22–31. https://doi.org/10.1109/IC2E.2016.38

[32] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the twenty-fourth ACM symposium on operating systems principles*. 423–438.

A TENSILE can correctly reconfigure common operators

Table 3 shows that both merge and pending operations can be easily defined for common window operators such as max, min, count, sum, mean, and median; as well as common continuous operators such as join.

For correct non-blocking reconfiguration of the distinct operator, a slight modification to the protocol is required. Distinct maintains a set of observed tuples. Upon encountering a new tuple, the operator outputs it and adds this tuple to its set of observed tuples. For example in Figure 14(a), instance D_1 outputs B_1 (shown in a green box) because it was not present in the set of observed tuples $\{B_0\}$.

Figure 14(a) shows how existing reconfiguration protocols migrate a key B from an instance D_1 to D_2. Again, the new instance D_2 remains blocked until it receives the state from D_1. In this simple example, D_1 outputs B_1 and D_2 outputs B_2.

Figure 14(b) shows an incorrect non-blocking reconfiguration of distinct. Here, the instance D_2 initializes from an empty state and starts processing input tuples. It outputs B_1 and B_2 tuples. When D_2 receives the state $\{B_0, B_1\}$, it merges

Table 3. Common relational stateful operators have $\odot$ to produce pending outputs and $\oplus$ to merge states. Therefore, they can be correctly reconfigured using the proposed high-throughput reconfiguration protocol.

Window operators. $st_k \circ t_k = \{\}$. $st_k \odot st'_k = \{\}$				
Operator	S_{id}	st_k+t_k	$\Diamond st_k$	$st_k \oplus st'_k$
Max	$-\infty$	$max(st_k, t_k)$	$\{st_k\}$	$max(st_k, st'_k)$
Min	∞	$min(st_k, t_k)$	$\{st_k\}$	$min(st_k, st'_k)$
Count	0	$st_k + 1$	$\{st_k\}$	$st_k + st'_k$
Sum	0	$st_k + t_k$	$\{st_k\}$	$st_k + st'_k$
Avg.	$sum = 0$ $count = 0$	$sum_k + t_k$ $count_k + 1$	$\frac{sum_k}{count_k}$	$sum_k + sum'_k$ $count_k + count'_k$
Median	$\{\}$	$st_k \cup \{t_k\}$	$median(st_k)$	$st_k \cup st'_k$

Continuous operators. $\Diamond st_k = \{\}$					
Op.	S_{id}	$st_k \circ t_k$	st_k+t_k	$st_k \odot st'_k$	$st_k \oplus st'_k$
Join $\bowtie$	$L = \{\}$ $R = \{\}$	$L_k \bowtie r_k$ or $l_k \bowtie R_k$	$L_k \cup \{l_k\}$ or $R_k \cup \{r_k\}$	$L_k \bowtie R'_k \cup$ $L'_k \bowtie R_k$	$(L_k \cup L'_k,$ $R_k \cup R'_k)$
Distinct	$\{\}$	$\{t_k\}$ if $t_k \notin st_k$ $\{\}$ otherwise	$st_k \cup t_k$	$st_k \setminus st'_k$	$st_k \cup st'_k$

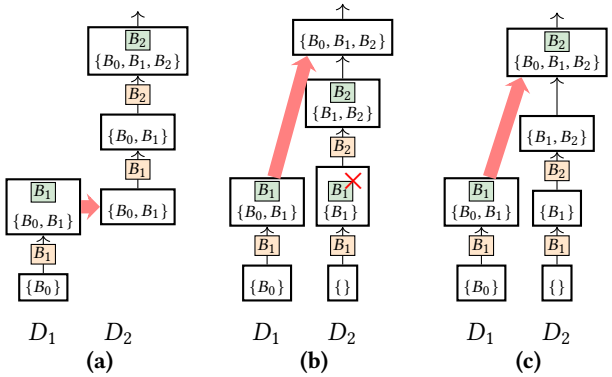


Figure 14. Non-blocking reconfiguration of the distinct operator suppresses outputs on the new instance D_2 until merge for correctness. (a) Blocking reconfiguration. (b) Incorrect: D_2 outputs an extra B_1. (c) Correct reconfiguration.

with its local state $\{B_1, B_2\}$ by taking a union. It can be seen that while the final state $\{B_0, B_1, B_2\}$ is consistent with Figure 14(a), D_2 produces an extra B_1 tuple (highlighted with a red cross) violating the semantics of distinct.

Fixing this is easy. D_2 does not emit any output for key B until it has received the corresponding state from D_1. However, D_2 still processes incoming tuples from its input channels (B_1 and B_2), updating its internal state ($\{\}$ to $\{B_1, B_2\}$), as illustrated in Figure 14(c). This reduces blocking and hence reduces backlog. Once the state transfer is complete, D_2 merges the states by taking a union. Following the merge, D_2 emits tuples observed by it and not by D_1, *i.e.*, it outputs B_2 but not B_1. It can be seen that with this simple change

of suppressing output tuples on D_2 before merge, the final states and outputs are consistent with Figure 14(a).[6]

A.1 Proof of correctness

While setting the *merge* operation $S_1 + S_2$ as $S_1 \cup S_2$ gives exactly the state S, a *pending* operation cannot be identified for distinct, where $S_1 \odot S_2 = O \setminus (O_1 \cup O_2)$. By suppressing outputs on the new instance (O_2), the proposed protocol provides output consistency when $S_1 \odot S_2$ is $O \setminus O_1$ instead of $O \setminus (O_1 \cup O_2)$.

B Reconfiguring an Abstract Operator with Tensile

We tweak the three event handlers in Algorithm 1 and add three new handlers for doing reconfigurations, as illustrated in Algorithm 2. Operators need to provide two additional operations $\oplus : (\mathsf{S}, \mathsf{S}) \to \mathsf{S}$ to merge local state with received state, and $\odot : (\mathsf{S}, \mathsf{S}) \to \mathcal{P}(\mathrm{Out})$ to generate pending outputs from local and received states (recall Figure 3). Operators like distinct also ask Tensile to *suppress* outputs before receiving all the states. In case operators are unable to provide $\oplus$ or $\odot$, we fall back to blocking reconfigurations.

function INIT We additionally track keys that the instance *has*. *has* is initialized to the initial set of keys.

function TUPLE_RECV If we are in the middle of a reconfiguration and have not yet received the state corresponding to the tuple's key ($k \notin has$), then we additionally suppress outputs if requested by the operator (*e.g.*, distinct).

function WM_RECV Watermarks are handled similarly as in Algorithm 1 with the additional caveat that output tuples and watermark can be produced iff the instance *has* all the keys (refer §2.2).

function FIRST_MARKER Reconfiguration markers carry the new partitioning function *par*. Scaling operator upon seeing the *first* marker from any of the input channels, notes the keys (*my_keys*) it is now responsible for, and the *old* keys that this instance is no longer responsible for. The instance does not yet have the state for the new keys, keys that this instance is newly responsible for. It initializes the state for new keys to S_{id}.

function LAST_MARKER Upon seeing the *last* marker from all the input channels, operator instance can send the state for old keys to its new owner. Finally it forwards marker to all the output channels.

function STATE_RECV The state sent by the old owner is received by a different instance responsible for executing the corresponding operator logic. This state pertains to the newly assigned keys and is combined with the current state to emit any pending outputs using $\odot$. Subsequently, it is merged into the local state using $\oplus$. Each instance continues to keep track of the keys for which it *has* the state.

[6] If the pipeline expects refinements [3], another solution was to let D_2 *refine* its output tuples by *retracting* B_1 after merge.

Algorithm 2 Tensile high-throughput reconfiguration protocol

```
1:  function INIT(par)
2:      keys ← has ← par(id)
3:      for k ∈ keys do
4:          st_k ← S_id

5:  function TUPLE_RECV(t_k)
6:      if k ∈ has ∨ ¬suppress then
7:          output st_k ∘ t_k
8:      st_k ← st_k + t_k

9:  function WM_RECV(wm)
10:     if keys ⊈ has then return
11:     for k ∈ keys ∧ Expiry(k) ≤ wm do
12:         output ◊st_k
13:     broadcast wm

14: function FIRST_MARKER(par)
15:     my_keys ← par(id)
16:     old ← keys \ my_keys
17:     for k ∈ my_keys \ keys do
18:         st_k ← S_id
19:     keys ← my_keys

20: function LAST_MARKER
21:     for k ∈ old do
22:         new_owner ← Instance(k)
23:         Send(new_owner, st_k)
24:         has ← has \ {k}
25:     broadcast ‖

26: function STATE_RECV(st'_k)
27:     output st_k ⊙ st'_k
28:     st_k ← st_k ⊕ st'_k
29:     has ← has ∪ {k}
```

C Additional evaluations

C.1 Reconfiguration Performance on Flink

Setup. We also implement Tensile in Apache Flink [7] and compare with existing all-at-once and fluid reconfiguration implementations in Flink. We run NEXMark Q3.

We use the same four-machine cluster described in Section 4.1. Each machine runs one Flink TaskManager deployed via Docker Swarm. The Q3 source operators are distributed across 24 parallel instances (6 per machine). The bottleneck `Join` operator starts with parallelism 1 and is scaled out to parallelism 2, transferring 64 out of 128 key groups to the new instance. The source generates 210 K events/s for the first 700 seconds, at which point the input rate is stepped up to 390 K events/s, exceeding the processing capacity of the single join instance. Reconfiguration is triggered 6 seconds later at the 706-second mark.

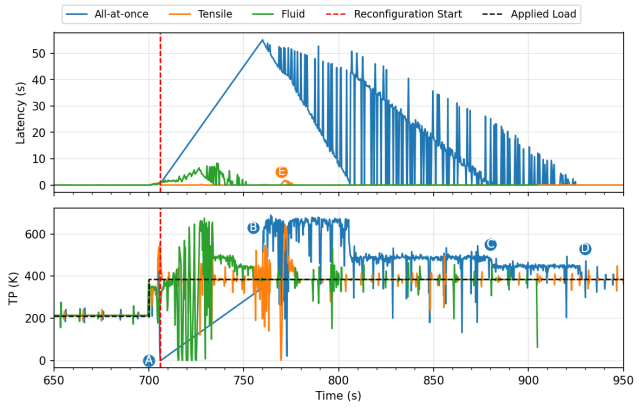


Figure 15. NEXMark Q3 in Apache Flink. The black dashed line shows that the applied input load is increased at 700 s. The join operator is scaled from parallelism 1 to 2 at the red dashed line (706 s). TENSILE recovers high throughput immediately, leading to lower latencies.

The experiment's parameters differ from the one in Table 2. The baseline Flink implementations only support scaling up by one instance, whereas in our Rust experiments we had scaled up by three instances. Being written in Java, Flink could provide lower sustainable throughput than our Rust implementation. Similar to Megaphone [15], we therefore run the pipeline longer to accumulate more state: by the time reconfiguration begins, accumulated state is ~700 MB.

Results. Figure 15 shows instantaneous P50 latency (top) and source throughput (bottom) for the three protocols. The broad findings are similar to those in §4.2. Both fluid and all-at-once reconfiguration throughput drops to zero due to blocking of channels. This loss of throughput builds backlog and the protocols exhibit higher latency. In contrast, TENSILE provides non-blocking reconfiguration where the new join instance immediately begins processing incoming rows even before the state has arrived. Therefore, its throughput recovers to applied load immediately after reconfiguration, leading to much lower latency.

We discuss differences from the results with our Rust implementation in Figure 9. Flink is slower in serializing the operator state and transmitting it. Because of this, the new join instance remains blocked for longer. Since the new join instance has its channels blocked, the sources are also backpressured due to which they stop sending rows to the old join instance too. This leads to more severe backlog. At ◯, the system comes to a near-total halt, leading to rapidly rising latencies. When processing resumes, both the old and the new join instances begin clearing the backlog at B. The old join instance has a smaller backlog and it clears it at C. The new join instance continues to clear it till ◯. In contrast to Figure 9(a), TENSILE also experiences a short latency spike at E during state merging due to slight implementation differences. Nonetheless, TENSILE comfortably outperforms both baselines.

Overall, experiments with Flink echo the findings from §4. TENSILE's non-blocking reconfiguration provides substantially lower latency and faster throughput recovery.

C.2 Performance during Other Types of Reconfiguration

In this section, we evaluate two other types of reconfigurations: (1) migration, where operator instances are migrated from existing machines to new ones, useful for doing planned maintainance; and (2) downscaling, where operator instances are removed, useful for saving operational cost.

For evaluating migration, we start with the same configuration as in Table 2 for all the queries. But instead of increasing the load, we move all the instances of the bottleneck operator to another physical machine. Note that we do not migrate the Source operator. For evaluating downscaling, we do the opposite reconfiguration from the one in Table 2, *i.e.,* start each experiment with 4x load and operator instances, and scale down to 1x load and 1x instances. The rest of the setup is same as described in §4.1. We skip plots for queries other than Q3 and Q8 since they transfer less state (recall §4.2).

Migration. Figure 16 shows instantaneous latency, throughput, and CPU utilization during migration. We observe that when migration is triggered (shown by the vertical red line), the throughput for all-at-once reconfiguration drops significantly ◯. This is because the old instance stops receiving tuples and the new instance cannot start processing tuples until it receives the migrated states: 2.8 GB for Q3 and 1.3 GB for Q8. This leads to large latency spikes B: upto 10 seconds and 6 seconds for Q3 and Q8 respectively. Since both the instances are not processing any tuples, all-at-once reconfiguration under-utilizes available CPU resources during reconfiguration C[7]. After the migration is complete, new machine's CPU utilization spikes up ◯ to clear the built-up backlog, gradually recovering low latency.

Fluid reconfiguration transfers key-groups gradually, hence processing halts only for the key-groups whose state is in transit. Therefore, it experiences a lower throughput drop E with the peak latency reaching upto 2.1s for Q3 and 1.6s for Q8. In contrast, TENSILE shows no degradation in latency and throughput since the new instance starts processing tuples without waiting for the migrated state. Hence, CPU utilization of the new machine is higher during migration with TENSILE, compared to other protocols F.

Downscaling Figure 17 shows the latency and throughput for Q3 and Q8 during descaling. We observe that all-at-once and fluid reconfiguration protocols show peak latencies of

[7]CPU utilization never becomes zero since we are not migrating Source operator instances.

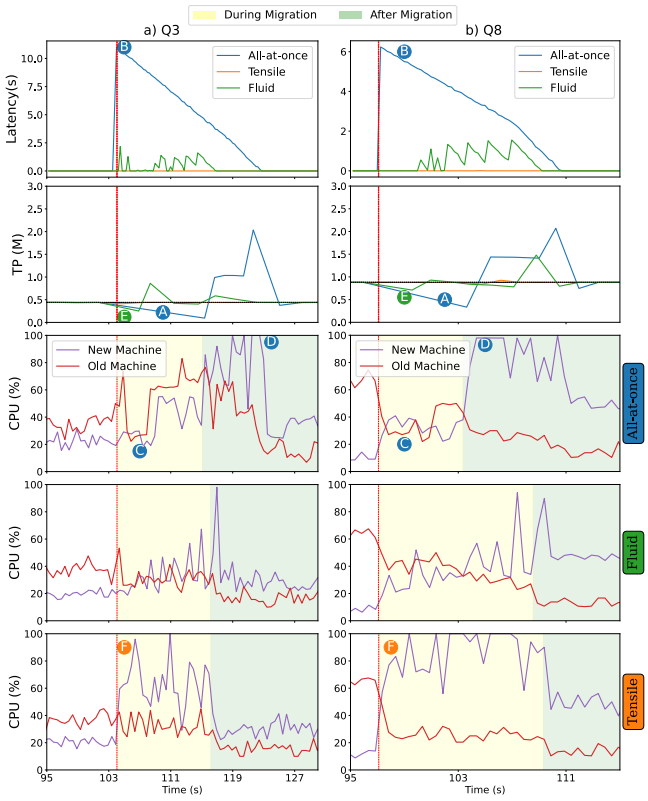


Figure 16. Latency, throughput, and CPU utilization over time for Q3 and Q8 while migrating operator instances. TENSILE exhibits lower latency spikes, faster latency recovery, and faster throughput recovery across both the queries, when compared to all-at-once and fluid reconfiguration protocols.

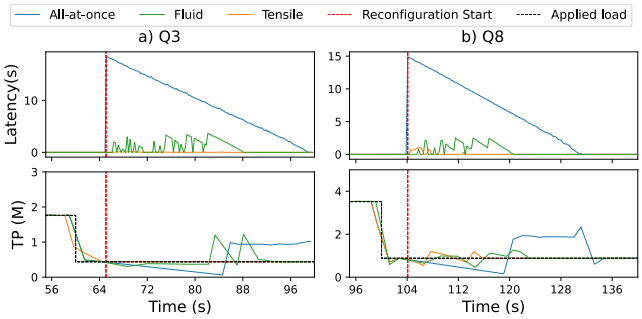


Figure 17. Instantaneous latency and throughput for Q3 and Q8 while downscaling operator instances.

18s, 3.6s for Q3 and 15s, 2.6s for Q8 respectively. TENSILE outperforms both of them with peak latencies of 0.1s for Q3 and 1s for Q8. Similar to upscaling and migration, this is because TENSILE suffers lesser throughput degradation compared to existing reconfiguration protocols, where instances have to block processing waiting for migrated states.

Overall, we see that TENSILE *suffers minimal throughput degradation during migration and downscaling, due to which it reduces peak latencies by 2.5x compared to existing protocols.*

TENSILE*'s advantage becomes more pronounced when the state transfer size is large.*

C.3 Key-group Size Sensitivity Analysis for Fluid Reconfiguration

In this section, we vary the number of key-groups to transfer in one sub-reconfiguration for fluid reconfiguration, and compare the performance with TENSILE. We repeat upscaling (§4.2) experiments for Q3 and Q8. In each experiment, we vary the key-groups in a sub-reconfiguration from 32 to 1024.

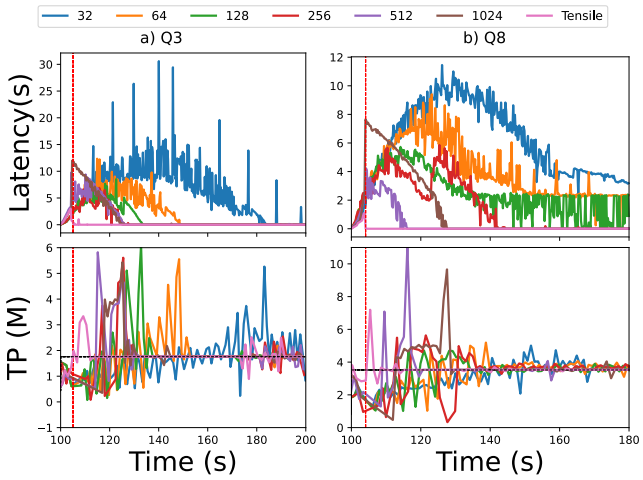


Figure 18. Impact of batch size on latency and throughput for queries Q3 and Q8.

From Figure 18, we observe that a smaller number of key-groups leads to higher latency. This is because the system recovers available capacity much more slowly. Therefore, when the input load is unsustainable, such as in these experiments, it takes longer to clear the backlog. TENSILE provides lower latency in all key-group size settings for both the queries, since it utilizes available capacity immediately after a reconfiguration is initiated.