On the Semantic Overlap of Operators in Stream Processing Engines

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

Alessandro Margara
Politecnico di Milano
Milano, Italy
alessandro.margara@polimi.it

Marina Papatriantafilou
Chalmers University of Technology
Gothenburg, Sweden
ptrianta@chalmers.se

ABSTRACT

Stream Processing Engines (SPEs) extract value from data streams in the Edge-to-Cloud continuum through graphs of operators that progressively transform data.

State-of-the-art SPEs are bridged into shared models based on their overlapping APIs. The overlap in their semantic expressiveness, though, goes beyond their APIs and can be formally assessed by distilling the semantics they support into minimal sets of operators, and by checking whether such sets overlap. As we show, stream Aggregates suffice to enforce the semantics of other common operators. Moreover, compositions of Aggregates can match the performance of other operators in state-of-the-art SPEs, and micro-SPEs building on a single Aggregate operator can even surpass other SPEs' performance while holding the same semantic expressiveness with a minimal code footprint.

Our approach lays down new analytical findings with practical implications in minimizing the operational effort to use SPEs, especially at the edge, while seamlessly benefiting existing distribution/parallelization techniques.

CCS CONCEPTS

• **Information systems** → **Data management systems**; • **Theory of computation** → **Streaming models**.

KEYWORDS

Stream processing, Stream Aggregates, Semantic Equivalence

ACM Reference Format:
Vincenzo Gulisano, Alessandro Margara, and Marina Papatriantafilou. 2024. On the Semantic Overlap of Operators in Stream Processing Engines. In *25th International Middleware Conference (Middleware '24), December 2–6, 2024, Hong Kong, Hong Kong*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3652892.3654790

Middleware '24, December 2–6, 2024, Hong Kong, Hong Kong

ACM ISBN 979-8-4007-0623-3/24/12.
https://doi.org/10.1145/3652892.3654790

1 INTRODUCTION

Stream Processing Engines (SPEs) allow running *queries* that distill information from data streams in the Edge-to-Cloud continuum. SPEs define queries as graphs of operators with parallel/distributed deployments in which multiple operator instances analyze parts of their input stream(s) in parallel [1].

Many SPEs have been developed over the years. While their APIs, designs, and implementations differ, the operators they offer overlap. Such an overlap led to unified models where queries expressed in an SPE-agnostic language can be compiled in specific SPEs. In this context, the state-of-the-art [2] bridges SPEs through the operators/APIs they share. The extent to which two SPEs overlap, though, goes beyond their API/operators and can be assessed by distilling the semantics supported by each SPE into minimal sets of operators. Formally, we ask: *Given a set of operators* $\mathbb{O}$*, is there a subset* $\mathbb{O}^*$ *such that compositions of operators in* $\mathbb{O}^*$ *can support the same semantics of compositions of operators in* $\mathbb{O}$*, with* $\mathbb{O}^* \subset \mathbb{O}$*?*

> EXAMPLE
> Imagine SPE_i offers operators O_1 and O_2 while SPE_j offers O_1, O_2, and O_3. Existing unified models propose high-level languages that compile queries using O_1, O_2, and O_3 to SPE_j, and queries that only use O_1 and O_2 to either SPE_i or SPE_j. For a general approach, we ask: *can* O_1 *and/or* O_2 *enforce* O_3*'s semantics?* If so, *any portion* of *any composition* from $\mathbb{O}_i \cup \mathbb{O}_j$ can be run on SPE_i or SPE_j, which is stronger than implying both SPE_i and SPE_j can run whole compositions from $\mathbb{O}_i \cap \mathbb{O}_j$.

Our work makes several novel contributions, showing that:

- being $\mathbb{O}$ the set of common SPEs' operators, there exists at least one non-empty $\mathbb{O}^*$ consisting of only the Aggregate operator, sufficient to address the question at hand,
- if an SPE offers several Aggregate operators and thus leads to possibly different $\mathbb{O}^*$s, our approach allows quantifying their design, portability, and performance trade-offs,
- compositions of Aggregate operators can in fact enforce the semantics of operators from a larger $\mathbb{O}'$ (i.e., $\mathbb{O}^* \subset \mathbb{O} \subset \mathbb{O}'$).

Together with our theoretical contribution, we empirically show with real-world data, hardware setups representative of the different ends of the Edge-to-Cloud continuum, state-of-the-art SPEs (Apache Flink [3]/Spark [4]), and a novel SPE called μSPE that builds on a single Aggregate operator and

consists of ~2000 lines of code, that (1) there exist queries for which, within the same SPE (Flink/Spark), Aggregate compositions perform comparably to that of other operators, and that (2) a minimal SPE (μSPE) can hold the same semantical expressiveness and outperform state-of-the-art SPEs (Flink/Spark). This is particularly important in edge scenarios in which users interested in stream processing struggle with the language, communication, CPU, or memory requirements of SPEs that are best fit for data centers. By building on the Dataflow model [1], μSPE is not only as expressive as other SPEs, but also allows for queries to run distributedly (different Aggregates can run at different processes/nodes) and in parallel (multiple instances of an Aggregate can process disjoint subsets of the incoming data), both with shared-nothing and shared-memory approaches [1, 3, 5, 6]. That is, its compactness does not prevent the use of well-established performance-boosting techniques in stream processing. Our code is available at [7, 8].

Note that, differently from other literature (e.g., [6, 9–13]), $\mathbb{O}_i^* \subset \mathbb{O}_i$ does not aim at defining new operators besides the ones offered by SPE_i. Also, we only consider common Dataflow operators (Map, Filter, Aggregate, Join) and not custom ones that, while supporting any semantics, build on custom user-defined code not maintained by SPE_i itself.

We believe our work can stimulate novel research: (a) show that query Q, supported by SPE_i through $\mathbb{O}_i$, is also supported by SPE_j if $\exists \mathbb{O}_i^*, \mathbb{O}_j^* | \mathbb{O}_i^* = \mathbb{O}_j^*$, (b) show that if a composition of $\mathbb{O}_i^*$ operators meets the semantics of an operator $O \notin \mathbb{O}_i$, then SPE_i can also run O, (c) study SPEs/queries design, portability, and performance trade-offs for compositions from one (or more) subsets $\mathbb{O}^*$s able to enforce the semantics of compositions from $\mathbb{O}$, (d) differentiate the semantics SPE_i supports (through $\mathbb{O}_i^*$) from SPE_i's implementation choices (through $\mathbb{O}_i$).

Outline: § 2 covers preliminaries; § 3 covers our problem formalization, § 4 discusses the existence of $\mathbb{O}^*$ (with $|\mathbb{O}^*| = 1$) for an $\mathbb{O}$ composed of common streaming operators; § 5 elaborates further on $\mathbb{O}^*$'s properties; § 6 evaluates two $\mathbb{O}^*$s and $\mathbb{O}$ in real-world use cases; § 7/§ 8 cover related work and conclusions.

2 PRELIMINARIES

2.1 Stream processing basics

A *stream* S is an unbounded sequence of *tuples*, each a list of attribute-value pairs $\langle \tau{:}v_0, a_1{:}v_1, ..., a_n{:}v_n \rangle$ [1]. Within S, every tuple t has the same attributes, called *type* and denoted as $T(t)$ or $T(S)$. The timestamp attribute (τ) is always included in the type of a tuple. We use $t[i]$ to denote t's attribute at index i (e.g., $t[0] = t.\tau$) and $t[i{:}j]$ for the attributes from i to j (inclusive).

Queries are composed of *ingresses*, *operators*, and *egresses*. Ingresses forward *ingress tuples* (e.g., events from sensors) to operators that, connected in a directed graph, process and forward/produce tuples. Eventually, *egress tuples* are fed to egresses and delivered to end-users/other applications. Multiple copies of an operator can be deployed within the same graph to analyze different portions of a given stream (e.g., the portion sharing the same key, as discussed next). For an ingress tuple t, $t.\tau$ denotes its *event time*. Operators set $t_o.\tau$ of an output tuple t_o according to their semantics, as explained next. Event time is expressed in units from a given epoch and progresses in SPE-specific δ increments (e.g., milliseconds [3]).

The set $\mathbb{O}$ of common operators we consider contains both *stateless* and *stateful* operators. Operators like Map and Filter are stateless and process tuples one-by-one:

Map $S_O = M(S_I, f_M)$ processes S_I's tuples with function f_M to produce stream S_O. Function f_M is invoked on each $t_i \in S_I$ to produce zero, one, or more t_o output tuples. Note $T(S_I)$ and $T(S_O)$ can differ and that $t_o.\tau = {}_i.\tau$ for a t_o produced from t_i.

Filter $S_O = F(S_I, f_C)$ forwards each $t_i \in S_I$ to S_O if $f_C(t_i)$ holds. Note $T(S_I) = T(S_O)$ and $t_i = t_o$ for a t_o output by processing t_i.

Stateful operators produce results from a state, dependent on one or more tuples. In this paper, we consider common [3, 14, 15] stateful operators defined over delimited groups of tuples called *time-based windows* (or simply windows): *Aggregates* and *Joins*. In § 5, we elaborate on how our analysis can be extended to more general definitions of state.

We denote as $\Gamma(WA, WS, S_I, f_K, L)$ a window specified by:

Window Advance (*WA*), Size (*WS*) define the epochs covered by Γ: $[\ell WA, \ell WA + WS)$, with $\ell \in \mathbb{N}$. We refer to one such epoch as window *instance* γ. If $WA < WS$, consecutive γs overlap, Γ is called *sliding*, and a tuple can fall into many γs. If $WA = WS$, Γ is called *tumbling* and each tuple falls in only one γ.

Input stream S_I is the input stream fed to Γ.

Key-by attribute f_K specifies the $T(S_I)$'s subset (even empty) used to keep dedicated γs for tuples with the same *key*. Note f_K affects how the operator maintaining Γ is parallelized.

Allowed Lateness L which is used to decide whether a tuple t falling in γ but received by the operator maintaining Γ after such operator has produced a result for γ should still be added to γ, potentially resulting in a new (or updated) output tuple[1].

We refer to the set of tuples falling in a γ as $\gamma.\zeta$ and to individual tuples in $\gamma.\zeta$ as if $\gamma.\zeta$ is maintained as a list (i.e., $\gamma.\zeta[0]$ is the tuple at index 0). We refer to the event time of γ's left boundary (inclusive) as $\gamma.l$. The right boundary of γ (exclusive) is computed as $\gamma.l + WS$. As common in related works [2, 3, 14], $t_o.\tau$ for an output tuple t_o created in connection to γ is set to $\gamma.l + WS - \delta$. Since γ's right boundary is exclusive:

Observation 1. *For any output tuple t_o produced from a window instance γ in which input tuple t_i falls, $t_o.\tau \geq t_i.\tau$.*

We consider the following stateful operators:

[1] We discuss L in § 2.3, after covering correctness conditions in § 2.2.

Aggregate S_O=$A(\Gamma(WA, WS, S_I, f_K, L), f_O)$ defines $f_O(\gamma)$ to compute the values of an output t_o from γ (except τ, set by A to $\gamma.l+WS-\delta$) and forwards t_o if f_O does not return $\emptyset$.

Join S_O=$J(\Gamma(WA, WS, S_{I_1}, f_K^1, L), \Gamma(WA, WS, S_{I_2}, f_K^2, L), f_P)$ defines f_P to match tuples $t_1 \in S_{I_1}$ and $t_2 \in S_{I_2}$ that fall in aligned windows γ_1 and γ_2 according to WA and WS, so that $f_K^1(t_1) = f_K^2(t_2)$, forwarding $\langle \gamma_1.l+WS-\delta, t_1, t_2 \rangle$ if $f_P(t_1,t_2)$ holds.

In the remainder, we use the following definition:

Definition 1. Tuple t' is a *successor* of t or belongs to the finite set of t's successors if t' is output by an operator O upon processing of t or a successor of t. We then write $t' \in succ(t)$ or $t' \in succ(\mathbb{T})$ when $\mathbb{T}$ is a set of tuples processed by O.

2.2 Correctness conditions

When deploying and running M, F, A, and J operators, SPEs should enforce such operators' semantics correctly. Since M and F process tuples one-by-one, correct semantics are enforced processing each tuple exactly once. A and J require greater care, though. Leaving aside late arrivals (see § 2.3), their correct execution can be defined as follows.

Definition 2. A's execution is correct if any subset of tuples sharing the same key and falling in the same γ is jointly processed by f_O exactly once and any output is fed to A's downstream peer(s). J's execution is correct if any pair of tuples $t_1 \in \gamma_1$, $t_2 \in \gamma_2$ is processed by f_P exactly once if $\gamma_1.l = \gamma_2.l$ and $f_K^1(t_1) = f_K^2(t_2)$ and the output is fed to J's downstream peer(s).

Correct execution for A/J in asynchronous systems can be achieved with the support of *watermarks* [16]:

Definition 3. The watermark W_A^ω / W_J^ω of A/J at wall-clock time ω is the earliest event time a tuple t_i fed to A/J can have from time ω on (i.e., $t^i.\tau \geq W_A^\omega / W_J^\omega, \forall t_i$ processed from ω on).

In the literature [3, 16], watermarks are commonly maintained assuming ingresses periodically output watermarks as special tuples to notify how event time advances. Upon receiving a watermark, A/J store the watermark's time, update W_A^ω / W_J^ω to the smallest of the latest watermarks from each input stream, and propagate W_A^ω / W_J^ω. Upon an increase of W_A^ω and before forwarding W_A^ω (given Observation 1), A invokes f_O on any $\gamma | \gamma.l + WS \leq W_A^\omega$ (in $\gamma.l$ order) and then discards such a γ since no more tuples will fall in it. Likewise, J can use W_J^ω to produce results and safely discard γs that cannot produce more results before forwarding W_J^ω. Specifically, J can safely discard any pair of γ_1,γ_2 defined over S_{I_1} and S_{I_2}, respectively, for which it holds that $\gamma_1.l = \gamma_2.l$ and $\gamma_1.l + WS \leq W_J^\omega$.

2.3 Handling late arrivals

As introduced in § 2.1, Γ defines L to handle late arrivals. More concretely, by delaying the purging of γs by L. Tuple t is a late arrival for operator O if $t.\tau < W_O^\omega$ when, at time ω, O processes t. According to the Dataflow model [1], t is processed, added to γ, and can result in an output tuple (potentially an update of a previous output tuple) if $\gamma.l + WS \leq W_O^\omega + L$ at ω. Note that, if $L > 0$ and watermarks are forwarded by O as described in § 2.2, results produced by O could be late arrivals for O's downstream peers. For compact notation, we omit L for a Γ if $L = 0$. In the remainder, we also make use of the following definition:

Definition 4. We say t is an output tuple *triggered* by the growth of O's watermark to W_O and write $t \in trig(W_O)$ if t is produced/forwarded by O when O's watermark grows to W_O.

3 PROBLEM FORMALIZATION

We formally show that for the set of operators $\mathbb{O} = \{M,F,A,J\}$, there exists a *semantically-equivalent subset* $\mathbb{O}^* = \{A\}$ whose As' compositions support the same semantics as compositions from $\mathbb{O}$. To show the reasoning supported by our formal approach, we also show (1) how different $\mathbb{O}^*$s with the same cardinality can be defined by means of different A implementations to argue and assess their design, portability, and performance trade-offs, and (2) how A compositions can enforce the semantics of operators from an $\mathbb{O}'$ so that $\mathbb{O}^* \subset \mathbb{O} \subset \mathbb{O}'$. Since $\mathbb{O}^*$ contains a single A operator, we denote the implementations of M,F,J from $\mathbb{O}$ as *Dedicated*, and those that rely on compositions of A operators as *AggBased*.

We consider SPEs such as [3, 4] for which the following holds:

P1 Streams with the same type can feed the same A operator. If streams S_{I_1},S_{I_2},... are fed to A, we write $\{S_{I_1}, S_{I_2}, ...\}$ to refer to the merged stream and $T(S_{I_1})$ to refer to their shared type.

P2 A stream can feed one or more A operators, delivering the same tuples/watermarks in the same order.

P3 A operators can iterate over their output stream with loops.

For **P3**, note an output t_o from A fed to A via a loop is always a late arrival since it holds that $W_A^\omega > t_o.\tau$ for t_o to be output (see § 2.2). Also, when processing t_o, any further output is a late arrival for A's downstream peers. While discussing the handling of late arrivals caused by the loops our model relies on, we do not discuss the handling of late arrivals forwarded by Ingresses themselves, already covered in [3], and assume:

C1 Each stream S delivers watermarks with a max event time distance d between W^i and W^{i+1}. If the first tuple $t^0 \in S$ precedes the first watermark W^0, then $W^0 - t^0.\tau \leq d$.

C2 An SPE handles the watermarks fed to A so that any t fed through a loop is not discarded due to being a late arrival.

C3 An SPE handles/delays watermarks emitted by A so that no output from A is a late arrival for A's downstream peers.

About **C1** note if an A operator is fed a stream for which **C1** holds, a distance d exists for A's output too. By extension, if an AggBased M, F, or J fed S produces a stream for which a d

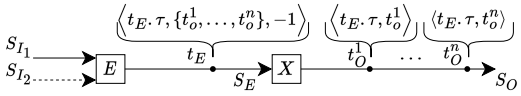

Figure 1: Composition of Embed (E) and Unfold (X) that can enforce M and J semantics.

exists according to **C1**, **C1** extends to such M, F, or J and downstream M, F, or J peers too. Also, note **C2**/**C3** are here simplified to ease exposition, see § 4.4 for a detailed formulation.

4 ENFORCING COMMON OPERATORS' SEMANTICS WITH A OPERATORS

To show that $\mathbb{O}^* = \{A\}$ is a semantically-equivalent subset of $\mathbb{O} = \{M,F,A,J\}$, we first note that $\{M\}$ is a semantically equivalent subset of $\{M,F\}$ since M, like F, can output 0 or 1 tuples per input tuple without altering the input type. We thus show that $\{A\}$ is a semantically-equivalent subset of $\{M,J\}$.

The key challenge is that M and J can output multiple tuples for each input tuple or window, respectively, while A produces up to one tuple per window (see § 2.1). We address this challenge with two *auxiliary operators*, expressible as compositions of As: Embed (E), to encapsulate the content of several tuples within one, and Unfold (X), to unfold them back to individual tuples. These auxiliary operators serve as "scaffolding" abstractions to ease subsequent theorems and proofs.

4.1 The Embed (E) and Unfold (X) Operators

Figure 1 illustrates how E/X enforce M/J semantics. E takes one input stream S_{I_1} and an optional S_{I_2} (since J has two input streams), possibly with different types, and outputs a stream S_E in which t_E tuples carry a set of tuples $\{t_o^1,\ldots,t_o^n\}$ and value -1 to specify t_E comes from E (see § 4.4 for more details). For M, t_E carries all the results produced by processing an input tuple. For J, t_E carries all the matching pairs from two aligned γs (see § 2.1). X unwraps t_E with a loop (see § 4.4) and outputs all tuples $\{t_o^1,\ldots,t_o^n\}$ in t_E. Sample executions are shown in Figure 2.

To enforce M's semantics, we note that:

Claim 1. *E and X enforce the semantics of $S_O = M(S_{I_1}, f_M)$ if for each $t \in S_{I_1}$, there is a t_E such that $t_E.\tau = t.\tau$ and $f_M(t)$ is carried in t_E's second attribute (i.e., $t_E[1]$).*

For E/X to enforce J's semantics, E should embed all matching tuples from a pair of aligned γ_1,γ_2 in a $t_E | t_E.\tau = \gamma_1.l + WS - \delta$:

Claim 2. *E and X enforce the semantics of $S_O = J(\Gamma(WA, WS, S_{I_1}, f_K^1), \Gamma(WA, WS, S_{I_2}, f_K^2), f_P)$ if, for each pair of tuples $t_1 \in S_{I_1}$ and $t_2 \in S_{I_2}$ such that $f_K^1(t_1) = f_K^2(t_2)$ and $f_P(t_1, t_2)$, and for each pair γ_1,γ_2 such that $t_1 \in \gamma_1$, $t_2 \in \gamma_2$, and $\gamma_1.l = \gamma_2.l$, E produces an output tuple $t_E = \langle \gamma_1.l + WS - \delta, \{t_o^1,\ldots,t_o^n\}, -1\rangle$ carrying t_1, t_2 in $t_E[1]$.*

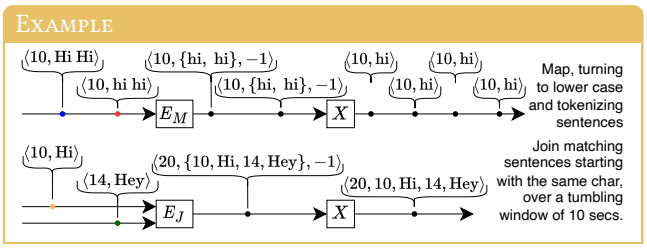

Figure 2: Sample input/output tuples of E_M/X and E_J/X. Event time is expressed in seconds and $\delta = 1$ sec.

In the remainder, we write $E_M(S_{I_1}, f_M)$ to refer to the E encapsulating M's semantics, $E_J(WA, WS, S_{I_1}, S_{I_2}, f_K^1, f_K^2, f_P)$ for the E encapsulating J's semantics, and $X(S_E)$ for X.

4.2 Using an A to Enforce E_M's Semantics

Our intuition for A's behavior to resemble that of a stateless operator is that if a tuple t_i is keyed using all its attributes then t_i is not jointly processed with other different tuples within a γ. Also, if γ is a tumbling window of δ time units, γ's output tuples share the same timestamp of input tuples. That is:

Lemma 1. *If A defines a tumbling Γ with $WA = WS = \delta$, then any t_i falls in exactly one window instance $\gamma | \gamma.l = t_i.\tau$. Also, if $t_i \in \gamma$ and t_o is produced upon invocation of $f_O(\gamma)$, then $t_o.\tau = t_i.\tau$.*

Based on Lemma 1[2], we can therefore state that:

Theorem 1. *The semantics of $S_E = E_M(S_{I_1}, f_M)$ – Claim 1 – are enforced with the A operator in List. 1.*

Listing 1: *A* operator implementing E_M's semantics

```
S_E = E_M(S_I, f_M) = A(Γ(δ,δ,S_I,T(S_I)), f_O), where:
1 Function f_O(γ)
2     T ← {} // Create empty set T
3     for t' ∈ γ.ζ do T ← T ∪ f_M(t') // Add f_M(t') to T, ∀t' ∈ γ
4     if T ≠ {} then return T, −1 // 1+ tuples from f_M
5     else return {} // No tuple from f_M
```

Figure 3's extends Figure 2 showing the tuples and state of an A enforcing E_M's semantics. Note that, since f_O is invoked on the γs of an A in order, output tuples will be sorted on event time. This, though, does not break M's correctness (see § 2.2).

4.3 Using As to enforce E_J's Semantics

Our intuition is that if matching tuples have the same type, then they can be fed to the same A operator (**P1**), contribute to the same γ, and be matched when invoking f_O on γ:

Theorem 2. *The semantics of $S_E = E_J(WA, WS, S_{I_1}, S_{I_2}, f_K^1, f_K^2, f_P)$ – Claim 2 – are enforced by composing As as in Figure 4/List. 2.*

[2]To ease exposition, all proofs are found in Appendix A.

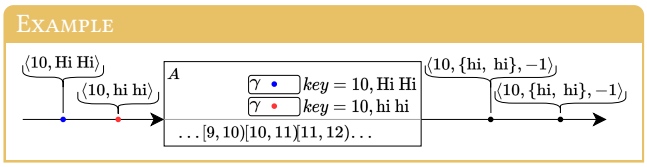

Figure 3: Tuples/states for the As of E_M from Figure 2.

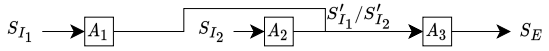

Figure 4: A operators implementing J's semantics.

Listing 2: A operators implementing E_J's semantics .

$S'_{I_1} = A(\Gamma(\delta,\delta,S_{I_1},T(S_{I_1})),f_O)$, *where:* // A_1 - Figure 4

```
1  Function f_O(γ)
2      𝕋 ← {} // Create empty set 𝕋
3      for t' ∈ γ.ζ do 𝕋 ← 𝕋 ∪ t' // Add t' to 𝕋, ∀t' ∈ γ
4      return 𝕋, {}
```

$S'_{I_2} = A(\Gamma(\delta,\delta,S_{I_2},T(S_{I_2})),f_O)$, *where:* // A_2 - Figure 4

```
5  Function f_O(γ)
6      𝕋 ← {} // Create empty set 𝕋
7      for t' ∈ γ.ζ do 𝕋 ← 𝕋 ∪ t' // Add t' to 𝕋, ∀t' ∈ γ
8      return {}, 𝕋
```

$S_E = A(\Gamma(WA,WS,\{S'_{I_1},S'_{I_2}\},f'_K),f_O)$, *where:* // A_3 - Figure 4

```
9  Function f'_K(t)
10     if t[2] = {} then return f^1_K(t[1][0]) // t is from S_{I_1}
11     else return f^2_K(t[2][0]) // t is from S_{I_2}
12 Function f_O(γ)
13     win1 ← {}, win2 ← {}, 𝕋 ← {} // Lists for in/out tuples
14     for t ∈ γ.ζ do
15         if t[2] = {} then // t is from S_{I_1}
16             for t' ∈ t[1] do // Join t's with win2 and store
17                 for t'' ∈ win2 | f_P(t', t'') do 𝕋 ← 𝕋 ∪ t', t''
18                 win1 ← win1 ∪ t' // Store t'
19         else // ... Symmetric if t is from S_{I_2}
20     if 𝕋 ≠ {} then return 𝕋, −1 // Return results
21     else return {} // There are no results to return
```

While referring to Appendix A for a formal proof, we note that since A_1/A_2 output streams with the same type (List.2,L1-8), both streams can be fed to A_3 (**P1**). A_3, who matches S_{I_1} and S_{I_2} tuples, defines an f'_K that runs f^1_K if $t \in S_{I_1}$ or f^2_K if $t \in S_{I_2}$ (List.2,L9-11), thus computing the correct key for t based on t's original stream. Since A_1/A_2 use all input tuples' attributes as key-by, if multiple input tuples are added to a set carried by an output tuple t_o, then all such tuples are identical and running f^1_K/f^2_K on the first tuple carried in $t_o[1]/t_o[2]$ consistently assigns tuples to γs based on their key-by value.

A_3's f_O runs a Cartesian product of all the tuples sharing the same key from S_{I_1} and S_{I_2}. All matching pairs are stored in $\mathbb{T}$, which A_3 forwards if not empty (List.2,L14-21).

Figure 5's extends Figure 2 showing the tuples and states of the As enforcing E_J's semantics.

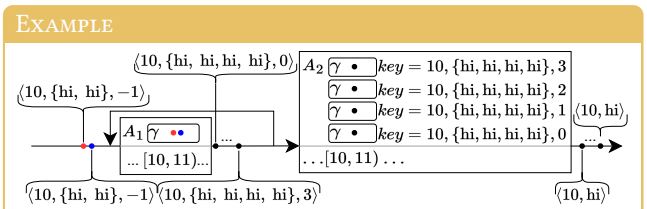

Figure 5: Tuples/states for the As of E_J from Figure 2.

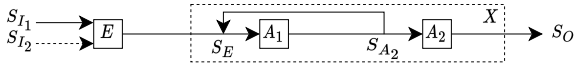

Figure 6: Composition of A operators for the X operator.

4.4 Using As to enforce X's Semantics

For an A to handle looping tuples (**P3**), we stated in § 3 three assumptions about the max distance of consecutive watermarks (**C1**), and (in a simplified formulation) about how A handles input watermarks (**C2**) and how A emits watermarks (**C3**). We first express **C2**/**C3** more precisely:

C2 W^ω_A is updated only once W^ω_A does not prevent any t_0, produced by A and fed to A via a looping stream S, from being processed on the basis of being a late arrival. That is, if $\forall t_o \in S$, t_o is still to be processed by A at ω, and being γ at A a window instance such that $t_o \in \gamma$, then it holds that $\gamma.l + WS \leq W^\omega_A + L$, and

C3 A feeds W^ω_A to downstream peers after $succ(trig(W^\omega_A))$.

Given such refined formulations, we can state the following:

Theorem 3. *The semantics of an $S_O = X(S_E)$ can be implemented by composing operators A_1 and A_2 as in Figure 6 and List. 3, where **C1** holds for S_E, **C2** and **C3** hold for A_1, and $L' \geq d$.*

Listing 3: A operators implementing X's semantics.

$S_{A_2} = A(\Gamma(\delta,\delta,\{S_E,S_{A_2}\},T(S_E),L'),f_O)$,
where: // A_1 - Figure 6

```
1  Function f_O(γ)
2      t ← γ.ζ[0]
3      if t[2] = −1 then // If t from E
4          𝕋 ← {} // Create empty set
5          for t' ∈ γ.ζ do 𝕋 ← 𝕋 ∪ t'[1] // Fill 𝕋 with outputs
6          return 𝕋, 0 // Forward 𝕋 and index 0
7      else if t[2] < |t[1]| then // If t should loop more
8          return t[1], (t[2]+1) // Increase index
9      else return ∅ // Else, done looping
```

$S_O = A(\Gamma(\delta,\delta,S_{A_2},T(S_{A_2})),f_O)$, where: // A_2 - Figure 6

```
10 Function f_O(γ)
11     t ← γ.ζ[0]
12     return t[1][t[2]]] // Forward tuple at given index
```

Note that A_1 has a tumbling Γ of size δ and Allowed Lateness L' (see § 2.2), and that S_{A_2} feeds both A_1 and A_2, according

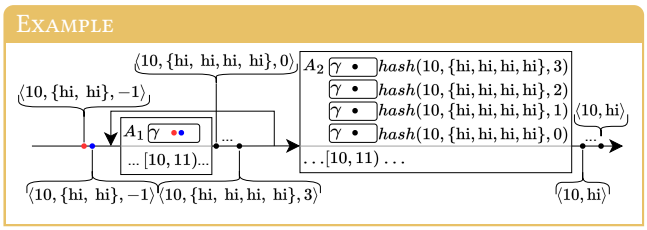

Figure 7: Tuples/states for the *A*s of *X* from Figure 2.

to **P2**. If a t from E carries -1 as the last value (List.3,L3), A_1 forwards a tuple t_o carrying all the output tuples carried by t and value 0, indicating the next tuple to be produced by A_2 is that at index 0. If t comes from A_1 itself, A_1 forwards a t_o increasing the counter at the last attribute by 1 (List.3,L8). Since A_1's f_K selects all tuples' attributes, identical tuples fed to A_1 from E will fall in the same γ and all the t_o tuples carried by them will be added to the output produced from γ. Hence:

Lemma 2. *X's A_1 (Figure 6) cannot produce duplicates.*

For the elements carried by an input tuple t in its second attribute, A_2 forwards the one at the index specified by t's third attribute (List.3,L12). Since each tuple fed to A_1 carries a finite set of t_o tuples, a t output from A_1 carries a finite set of t_o tuples too. Hence, the index at $t[2]$ will eventually grow greater than $|t[1]|$, ending the looping of t through A_1 and its feeding to A_2.

Figure 7's example extends Figure 2 showing the input/output tuples and states of the *A*s enforcing X's semantics.

4.5 Handling Watermarks in cyclic graphs

Observing not all SPEs support loops [4], we introduce two algorithms to enforce **C2**/**C3**. For Figure 6, we assume SPEs run four non-concurrent operations to handle a tuple t/a watermark W of streams S_E and S_{A_2}: $processT(t)$, when t is fed to S_E, $processW(W)$, when W is fed to S_E, $forwardT(t)$, when t is fed to A_2, and $forwardW(W)$, when W is fed to A_2.

The algorithms use (FIFO) *Queues*, with associated methods *enq* and *deq* – we write $Q[i]$ to refer to the element at Q's index i – and *TreeMaps*, maps with sorted traversal of its keys. We write $m[k]$ to refer to the key k's value in m. Method $firstKey$ returns the first key's value (without removing k nor its value). Method $remove(k)$ removes key k and its associated value.

List. 4 covers *processT* and *processW* to enforce **C2** for S_E. Simply put, *processT* and *processW* keep track of the number of tuples A_1 will send on the feedback loop for each γ. A_1's watermark is updated when there are no more tuples to wait for γs sharing a given right boundary. More concretely, S_E maintains four variables: B, a bound on the highest watermark that can be forwarded by S_E, $succ\Gamma$, a *TreeMap* that associates the left boundary of each γ at A_1 with its number of successor tuples, *pendingW*, a *Queue* of watermarks that can be forwarded by S_E, and L', A_1's Allowed Lateness (List.4,L1-4).

S_E forwards t upon its reception (List.4,L5-6) and updates $succ\Gamma$, increasing the entry for a window with left boundary $t.\tau$ (see Lemma 1) with the number of t's successors if t comes from the E operator (i.e., if $t[2] = -1$, List.4,L7), or decreasing the entry at $t.\tau$ if t is a successor of a previous tuple from E (List.4,L10). An entry in $succ\Gamma$ is removed if it decreases to 0 (List.4,L11-12). S_E proceeds setting B to ∞ if all entries in $succ\Gamma$ have been cleared, or to the left window boundary of the earliest successor yet to be received by S_E plus L' (List.4,L13-14). Finally, S_E tries to forward the latest watermark in *pendingW* that is smaller than or equal to B, discarding any other earlier watermark (List.4,L15-18).

S_E immediately forwards a watermark W upon receiving it if $W \leq B$ or stores it in *pendingW*, to try to forward W at a subsequent invocation of *processT* (List.4,L19-21).

Based on List. 4, we note that:

Lemma 3. *If **C1** holds for S_E, then List. 4 enforces **C2** for A_1.*

Listing 4: Enforcing **C2** for Stream S_E

```
1  B=∞ // Bound on W that S_E can forward
2  succΓ // TreeMap of γ's right bound-#successors
3  pendingW // Queue of W
4  L' // A1's Allowed Lateness
5  Function processT(t)
6      forwardT(t) // Forward t
7      if t[2]=−1 then
8          succΓ[t.τ]←succΓ[t.τ]+|t[1]| // Keep
               track of succ. for γ with left boundary t.τ
9      else
10         succΓ[t.τ]←succΓ[t.τ]−1
               // Decrease succ. for γ with left boundary t.τ
11         if succΓ[t.τ]=0 then // Got all succ. for t.τ
12             succΓ.remove(t.τ)
13     if |succΓ|>0 then B←succΓ.fistKey()+L'// Update B
14     else B←∞
15     nextW←−1 // Forward latest W|W≤B
16     while |pendingW|>0∧pendingW[0]≤B do
17         nextW←pendingW.deq()
18     if nextW≠−1 then forwardW(nextW)
19 Function processW(W)
20     if W≤B then forwardW(W)// If W within B, send W
21     else pendingW.enq(W)// Else, store W
```

List. 5 shows how *processT* and *processW* can enforce **C3** for S_{A_2}. Simply put, *processT* and *processW* keep track of the tuples A_1 sends on the loop and handle the forwarding of watermarks from A_1 (by postponing them or emitting extra ones) according to looping tuples that can result in more output tuples from A_1. More concretely, S_{A_2} forwards t upon its reception and updates variable $succ\Gamma$ (List.5,L4-10). Since S_{A_2} sees only the successors of a tuple t' forwarded by S_E to A_1, S_{A_2} increases the corresponding entry for a tuple t in $succ\Gamma$

to $|t[1]|-1$ (because t itself is a successor of $t'[1]$). Then, S_{A_2} forwards $t.\tau$ as a watermark if $succ\Gamma$ is empty, or the value preceding the earliest left boundary of a γ of a successor yet to be seen by S_{A_2}, if such a value is greater than the latest watermark forwarded by S_{A_2} (List.5,L11-16).

Note that, being $\mathbb{T}=succ(t)$, $\mathbb{T}[0]$ is fed to S_{A_2} before a watermark $W \geq \mathbb{T}[0].\tau$ (since A_1 needs W to output $\mathbb{T}[0]$ and feeds W to S_{A_2} after $\mathbb{T}[0]$, see § 2.2) while all other tuples in $\mathbb{T}$ are fed to S_{A_2} after W. Thus, all tuples increasing an entry of $succ\Gamma$ precede those decreasing an entry and, if the earliest key's value in $succ\Gamma$ decreases to 0 by processing $\mathbb{T}[0]$, the processing of other tuples in $\mathbb{T}$ can only decrease later keys' entries in $succ\Gamma$.

Upon invocation of $processW$, S_{A_2} forwards W if $succ\Gamma$ is empty, or the timestamp of the earliest left boundary of a window -1 of a successor yet to be seen by S_{A_2}, if such a value is greater than the latest W forwarded by S_{A_2} (List.5,L17-23).

Based on List. 5, we note that:

Lemma 4. *If **C1** and **C2** hold for S_E and A_1, respectively, then List. 5 enforces **C3** for A_1.*

Listing 5: Enforcing **C3** for Stream S_{A_2}

```
1  succΓ // TreeMap of γ's right bound-#successors
2  lastW // Last W forwarded by S_{A_2}
3  Function processT(t)
4      forwardT(t) // Forward t
5      if t[2] = −1 then
6          succΓ[t.τ] ← succΓ[t.τ] + |t[1]| − 1 // Record # of
             successors for γs starting at t.τ (t excluded)
7      else
8          succΓ[t.τ] ← succΓ[t.τ] − 1
             // Decrease successors for γ starting at t.τ
9          if succΓ[t.τ] = 0 then // Got all succ. for t.τ
10             succΓ.remove(t.τ)
11     if |succΓ| = 0 then // Forward t.τ as watermark
12         forwardW(t.τ)
13         lastW ← t.τ
14     else if succΓ.firstKey() − 1 > lastW then // Forward
         watermark based on earliest pending succ.
15         forwardW(succΓ.firstKey() − 1)
16         lastW ← succΓ.firstKey() − 1
17 Function processW(W)
18     if |succΓ| = 0 then // All pending succ. cleared
19         forwardW(W)
20         lastW ← W
21     else if succΓ.firstKey() − 1 > lastW then // Forward
         watermark based on earliest pending succ.
22         forwardW(succΓ.firstKey() − 1)
23         lastW ← succΓ.firstKey() − 1
```

After showing $\{A\}$ is a semantically equivalent subset of $\{M,F,A,J\}$ by discussing how A compositions can enforce E/X operators, we now discuss other semantically equivalent subsets and other semantics enforceable by composing As.

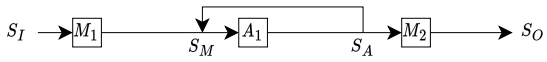

Figure 8: A operators implementing A_∞'s semantics.

5 REASONING ON $\mathbb{O}^*$ EXTRA PROPERTIES

5.1 On the trade-offs of different $\mathbb{O}^*$s

In § 1, we say two subsets that are semantically equivalent to an SPE's set of operators can quantify their model/performance trade-offs. After showing $\{A\}$ is semantically equivalent to $\{M,F,A,J\}$, we can state this holds also for $\{A^+\}$, being A^+ an Aggregate producing an arbitrary number of tuples from a single γ (e.g., as in Flink [3]), since it suffices to limit its outputs to one per γ. We can thus ask: *What model/performance benefits are given by the possibility for an Aggregate to emit an arbitrary number of tuples from a single γ?*

From a model perspective, A^+ can immediately forward the tuples that A would instead embed in $t_E[1]$ (see § 4.1), eliminating the need for X and, subsequently, for **P3** and **C1**-**C3**. We can thus conclude that compositions of A^+s (rather than As) can enforce the semantics of compositions of $\mathbb{O}$ operators without needing (1) a maximum distance d between the watermarks within each stream nor (2) the support for loops and their watermarks' handling. We empirically compare the performance improvements of A^+ over A in § 6.

5.2 On the extended semantics $\mathbb{O}^*$ supports

So far, we focused on $\mathbb{O}=\{M,F,A,J\}$. We can nonetheless ask: *is there a larger $\mathbb{O}'$ so that $\mathbb{O}\subset\mathbb{O}'$ and so that $\mathbb{O}^*$ is a semantically equivalent subset of $\mathbb{O}'$?* We believe such a question can stimulate novel research, beyond the scope of this work. As we show next, $\mathbb{O}'$ exists and contains, for example, an operator that can aggregate arbitrarily long-living states across windows without dedicated state backend/external storage.

Let us denote such an operator as A_∞. We want each tuple t fed to A_∞ to be processed exactly once and used to update a state based on all previously processed tuples. We note A_∞'s state is unbounded in terms of the event time it spans but should not have ever-growing space complexity. A_∞'s semantics can be enforced by relying on an A with a sliding Γ and a loop to transfer γ_l's state into γ_{l+1}. A_∞'s state can be defined in terms of a *state tuple* (with a type possibly different from that of input tuples). If we define $S_O=A_\infty(f_c,f_a,f_m,f_o,P,f_k,S_I)$ as a stateful operator for which $\boldsymbol{f_c}$ creates a state tuple t_s from an input tuple t_i, $\boldsymbol{f_a}$ adds t_i to an existing state tuple t_s, $\boldsymbol{f_m}$ merges two state tuples t_{s_1} and t_{s_2}, $\boldsymbol{f_o}$ produces an output tuple from t_s every P units of time, $\boldsymbol{f_k}$ is a key-by function (same as A), and $\boldsymbol{S_I}$ is A_∞'s input stream, we can state the following lemma:

Lemma 5. *The semantics of $S_O=A_\infty(f_c,f_a,f_m,f_o,P,f_k,S_I)$ can be enforced by composing As as shown in Figure 8 and List. 6 where **C1** holds for S_M, **C2**,and **C3** hold for A_1, and A_1's $L>d$.*

Listing 6: A operators implementing A_∞'s semantics.

$S_M = M(S_I, f_M)$, *where* $f_M(t)$ *returns* $\{t\},\{\}$ // M_1 - Figure 8

$S_A = A(\Gamma(P, P+\delta, S_M, f_k, L), f_O)$, *where:* // A_1 - Figure 8

```
1  Function f_o(γ)
2      first_t_s ← True
3      for t ∈ γ.ζ do // For each t ∈ γ
4          if t.τ ≠ γ.l+P−δ then // t falls only in γ
5              if t[1] ≠ ∅ then // t from M_1
6                  if first_t_s then // state not created
7                      t_s ← f_c(t[1])
8                      first_t_s ← False
9                  else t_s ← f_a(t_s, t[1]) // state created
10             else // t from A_1
11                 if first_t_s then // state not created
12                     t_s ← t[2]
13                     first_t_s ← False
14                 else t_s ← f_m(t_s, t[2]) // state created
15     return {},{t_s}
```

$S_O = M(S_A, f_M)$, *where* $f_M(t)$ *returns* $f_o(t[2])$ // M_2 - Figure 8

Note that, since A_1's γs are processed upon expiration, γ's tuples can be sorted on their type to order calls to f_c, f_a, and f_m.

6 EVALUATION

Our evaluation has two parts referred to as *intra-* and *inter-*SPE. The first studies how the performance of a query changes, within the same SPE (Flink), for AggBased and Dedicated operators. The second compares the performance of state-of-the-art SPEs (Flink and Spark) to an SPE that only builds on AggBased operators. We define a total of 24 different queries and evaluate them using 3 different SPEs and 2 hardware setups representing the different ends of the Edge-to-Cloud continuum. D refers to Dedicated implementations of $\{M, J\}$ while A/A+ refer to AggBased-compositions of A (see § 4)/A+ (see § 5.1) operators, respectively (we use the same notation for operators/implementations to ease exposition).

Datasets. Besides synthetic data, we use (1) a portion of the Wikipedia edits from [17] and (2) 2D scans from a laser scan sensor [18, 19]. For (1), tuples carry τ (`long`) and Strings *orig*, the original entry, *change*, the text being added, and *updated*, the modified entry; for (2), they carry τ (`long`), *id* (`int`), the tuple id, and *dist* (array of `double`), the distance readings.

Hardware. We use a *High-end server* (Intel Xeon E5-2637 v4@3.50GHz, 4 cores, 8 threads, 64 GB RAM) to process Wikipedia data, and an *Odroid* (Samsung Exynos5422 Cortex-A15 2Ghz, Cortex-A7 Octa core, 2 GB RAM) for 2D scans.

Software. For the intra-SPE part, we use Flink 1.15.2 [3]. Due to an open issue that can deadlock loops [20], we define our loop-handling mechanism. For D, we rely on Flink's M and J operators. For A/A+, we only compose Flink's own A operator and `process` function – which matches f_O, see § 2.1 – limiting the outputs to 1 for A or an arbitrary number for A+.

For the inter-SPE part, together with Flink, we use Spark 3.5.0 (both are used without alterations) and μSPE [8], a proof-of-concept SPE for distributed and parallel execution with exactly-once semantics of streaming applications whose API only offers to compose A+ operators. While holding the same semantic expressiveness of SPEs like Flink/Spark for arbitrary compositions of common operators, it trades features of state-of-the-art SPEs (e.g., custom memory management, integration of batched and streaming-based mode) with a minimal codebase and a resource footprint that can ease its adoption in resource-constrained edge devices.

Methodology. Table 1 lists our experiments. Performance is measured as maximum sustainable throughput, in tuples/second (t/s) for M and comparisons/second (c/s) for J, and associated average per-second latency. Latency, for M and J, refers to the delay in outputting t_o after all the inputs that jointly result in t_o have been fed to the operator producing t_o. We consider as maximum sustainable throughput the one an SPE with the query in focus sustains for 10 minutes without exceeding a latency of 15 seconds more than 3 times, excluding warm-up/cool-down phases, and average results over 3 runs.

We study various selectivity/per-tuple processing costs. Selectivity for M and J is the average number of outputs per input tuple and comparison, respectively. For M, we consider a selectivity smaller than one, to resemble an F, and equal to/-greater than 1. Since J's comparisons are quadratic in the input rate [21], selectivity values are smaller than 1 as in [13, 21].

6.1 Intra-SPE performance evaluation

Since A/A+ output extra intermediate tuples compared to D implementations, we expect A/A+'s performance degradation to be proportional to the selectivity of a given D operator. Moreover, we expect such degradation to be higher when per-tuple data movement costs dominate per-tuple processing costs. Acknowledging selectivity and processing costs are query-specific, we first validate our hypothesis by studying, with Flink and on the High-end server, the performance overheads for a synthetic M operator with selectivity of 0.1, 1, and 3, and per-tuple processing costs ranging from 0.6 to 46 μs.

Results are shown in Figure 9. Each line shows the throughput/percentage % achieved by A/A+ with respect to D for M and a given selectivity (i.e., the A/3 throughput line shows the percentage of throughput of an A-based M for selectivity 3). As we can see, the lowest throughput percentage, 3−28% for A and 30−38% for A+, is observed when the processing cost is small and is lower for higher selectivity. We can also observe that, independently of the per-tuple processing cost, a higher latency is always observable, noting A's latency is greater and grows faster than A^+'s. This is because D does not require

Table 1: List of experiments.

		High-end server (upper-case ID)			Odroid device (lower-case ID)		
Cost	**Op.**	**ID**	**Selectivity**	**Notes**	**ID**	**Selectivity**	**Notes**
Low	M	LLM	Low ($\sim 5e^{-3}$)	Find most frequent word in *orig*. Forward if length > 10 chars	llm	Low (0.2)	Convert coordinates from polar to Cartesian. Forward if avg dist. > 3m
Low	M	ALM	Avg (1)	Find the most frequent word in *orig* and forward it	alm	Avg (1)	Convert coordinates from polar to Cartesian
Low	M	HLM	High (3)	Find top-3 frequent words in *orig*. Forward as separate tuples	hlm	High (3)	Convert coordinates from polar to Cartesian, and split/forward in 3 parts
High	M	LHM	Low ($\sim 3e^{-4}$)	Find most frequent word in *orig*, *change*, and *update*. Forward them in a single tuple if all their lengths are > 10 chars	lhm	Low (~ 0.7)	Convert coordinates from polar to Cartesian from reference point. Forward if avg dist. > 3m
High	M	AHM	Avg (1)	Find most frequent word in *orig*, *change*, and *update*. Forward them in a single tuple	ahm	Avg (1)	Convert coordinates from polar to Cartesian from reference point
High	M	HHM	High (~ 2.3)	Find top-3 frequent words in *orig*, *change*, and *update*. Forward as separate triplets	hhm	High (3)	Convert coordinates from polar to Cartesian from reference point, and split/forward in 3 parts
Low	J	LLJ	Low ($\sim 1e^{-4}$)	Match distinct (case insens.) *orig* with the same length and $\|orig\| > 210$ chars. Key-by # of words in *change*, *WA*=1s, *WS*=3s	llj	Low ($\sim 8e^{-5}$)	Match two distinct scans if the sum diffs in *dist* is < 0.5m. *WA*=0.5s and *WS*=1s
Low	J	ALJ	Avg ($\sim 1e^{-3}$)	As LLJ, but $\|orig\| > 150$	alj	Avg ($\sim 8e^{-4}$)	As llj but sum diffs < 0.6m
Low	J	HLJ	High ($\sim 3e^{-3}$)	As LLJ, but $\|orig\| > 100$	hlj	High ($\sim 5e^{-3}$)	As llj, but sum diffs < 0.7m
High	J	LHJ	Low ($\sim 1e^{-4}$)	As LLJ, but *WS*=10s	lhj	Low ($\sim 6e^{-5}$)	As llj, but *WS*=2s
High	J	AHJ	Avg ($\sim 1e^{-3}$)	As LLJ, but $\|orig\| > 150$ and *WS*=10s	ahj	Avg ($\sim 7e^{-4}$)	As llj, but sum diffs < 0.6m and *WS*=2s
High	J	HHJ	High ($\sim 3e^{-3}$)	As LLJ, but $\|orig\| > 100$ and *WS*=10s	hhj	High ($\sim 3e^{-3}$)	As llj, but sum diffs < 0.7m and *WS*=2s

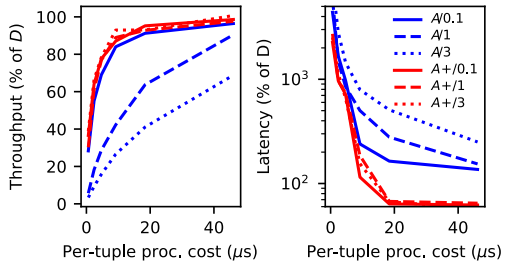

Figure 9: Throughput/latency percentage (A/A+ vs. D).

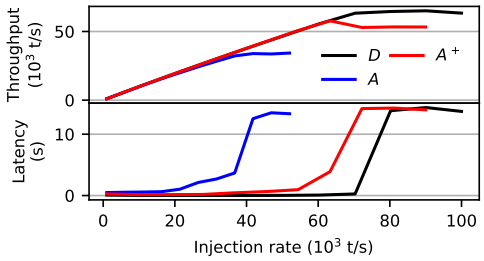

Figure 10: AHM-Throughput/latency vs. injection rate

watermarks to trigger the production of results (since M is stateless) while A/A^+ do, making their latency a function of the periodicity of ingresses' watermarks. Also, for A, the latency increase is due to the delay in the forwarding of watermarks paid to enforce **C2** and **C3** (see § 4.4). Finally, notice all our AggBased implementations undergo the additional overhead of key-by routing, since Flink extracts the key of each tuple before feeding such tuple to the corresponding A instance, while Dedicated M operators do not require such extra operation. A^+'s higher throughput and lower latency (compared to A) are due to A^+ not relying on the X operator and loops (see § 5.1). As shown, though, the throughput percentage drop becomes negligible for A+ for increasing loads independently of the selectivity, 98−100%, and even for A for low selectivity, 90−96% (i.e., when X's processing overheads are minimized). These results thus hint that, for a given SPE, cost-heavy operators can perform well when relying on A/A+ implementations. We now validate this with real-world queries for the M/J operators.

***M* operator** Figure 10 shows how throughput and latency evolve for an increasing injection rate (t/s) in the AHM experiment (see Table 1). Other experiments behave similarly.

The throughput initially increases linearly with the injection rate in all implementations but plateaus once the maximum sustainable throughput is reached, with the corresponding increase in latency. In line with the experiments with synthetic data, D's maximum throughput is higher than that of A/A^+, with a degradation of ~50% for A and ~16% for A^+.

Figure 11 compares throughput and latency across experiments. D's throughput is not significantly affected by selectivity as it is by cost. In this case too, as for previous experiments, this does not hold for A: when M's selectivity is close to 0, X only processes a few tuples. Hence, while E_M is not as efficient as M, their performance is in the same order of magnitude (e.g., 34% less throughput for A in LLM). When X's rate grows, the throughput drops, down to e.g., a 90% for HLM. This does not happen to A+, for which selectivity does not affect throughput (as for D) and thus results in a maximum sustainable throughput similar to that of D in Odroid-based experiments.

For the latency, we note that both A and A^+ consistently exhibit higher latency than D, with smaller differences between A and D on the Odroid device. We also note that A exhibits an increase in latency as selectivity increases, because of the loop

within the X operator. The growing trend in latency is also present for D and A^+ when executed on the High-end server, where D's latency is orders of magnitude lower than that of A/A^+, as expected since D is a stateless operator. Note, though, that A^+'s latency remains in the sub-second range (acceptable for many real-world queries) even on the High-end server.

***J* operator** Figure 12 shows the throughput and latency for experiment `ahj` (as for M, other experiments behave similarly). Based on the results with synthetic data, conducted on a stateless M, we expect a smaller performance gap between $A/A+$ and D, since J is usually heavier in per-tuple processing cost than M and relies on watermarks too, as an Aggregate does, for output production. D's and A^+'s behaviors are close, while A's latency grows faster as the rate increases. For A and A^+, all the comparisons for a given γ are done at once, once γ's right boundary falls before A's watermark, while for D they are done as tuples are being fed. Since A also requires the subsequent unfolding from X, A carries out higher amounts of work on γs expiration, leading to lower throughput/higher latency.

Figure 13 compares all experiments' throughput/latency. A^+ and J show negligible differences. We also note minor differences between A and D running on the Odroid, as in experiment `llj`. We also note that the growing trend of latency is mainly observed for the A operator, while in this case that A^+ and D exhibit comparable latency across all experiments.

6.2 Inter-SPE performance evaluation

After showing there exist queries for which, within an SPE, $A/A+$ and D behave similarly, we now compare M/J performance between Flink, Spark, and μSPE. Our initial experiments highlighted that Spark, which relies on micro-batching favoring throughput over latency [22], could not run on Odroid devices and resulted in high latency even on the High-end server. We thus changed the maximum latency to distinguish successful/unsuccessful experiments from 15s to

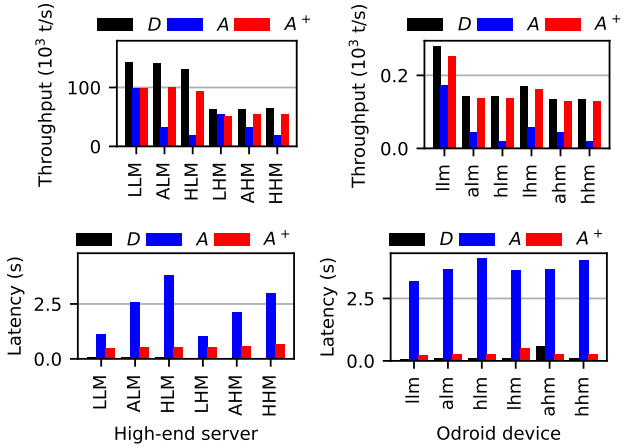

Figure 11: M-Avg. throughput/latency (all experiments).

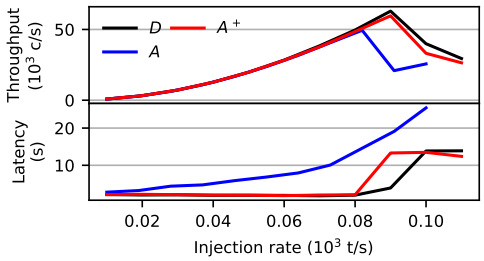

Figure 12: `ahj`-Throughput/latency vs. injection rate.

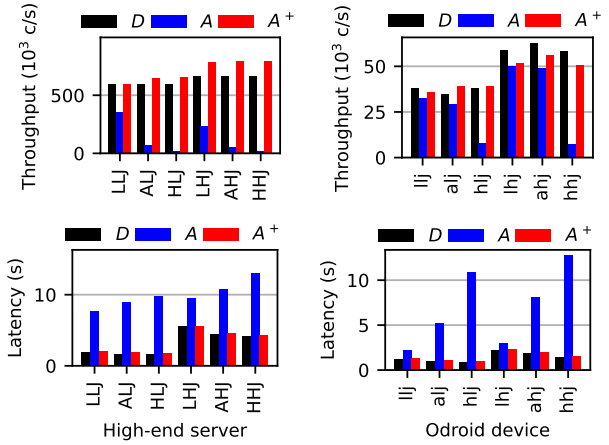

Figure 13: J-Avg. throughput/latency (all experiments).

60s for Spark, ran only experiments for the High-end server, and changed all windows to tumbling (based on their size).

Figure 14 shows the throughput and latency figures for M and J, respectively. We can observe that, in this case, μSPE is always able to provide better performance than Flink and Spark. For M, μSPE shows an average +43% throughput and comparable latency to Flink and +21% throughput and −90% latency than Spark. For J, it shows an average 2.6X higher throughput and comparable latency to Flink and 20X high throughput and −72% latency than Spark. These results highlight the trade-offs between features offered by state-of-the-art SPEs (e.g., integration of batched and streaming-based modes [3]) and the performance of a micro-SPE that, building only on Aggregates, holds the same semantic expressiveness while being easy to maintain, tune, and port across different languages/hardware.

7 RELATED WORK

We build on the Dataflow model [1], the de-facto standard in modern SPEs [23–28]. We are unaware of previous work formally or empirically covering the concept of semantically equivalent subsets in stream processing (see § 3).

Related studies that may complement future research include [29], which defines a calculus for streaming queries showing how to port higher-level languages like CQL [30]

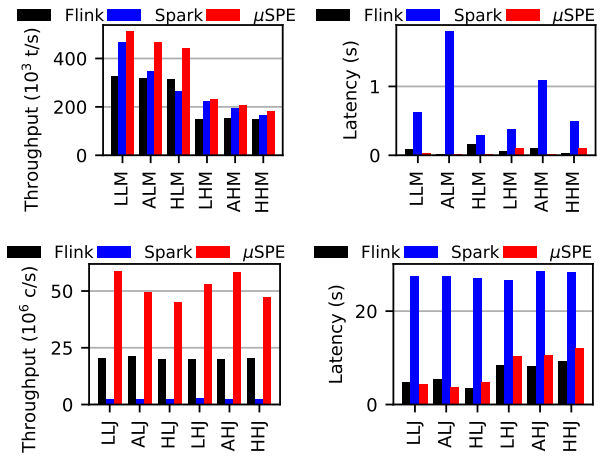

Figure 14: M/J performance for Dedicated (Flink, Spark) vs. AggBased (μSPE) implementations.

to it but by encapsulating semantics in orthogonal opaque functions, rather than basic Dataflow concepts as we do.

Some works focus on the semantics of individual aspects of SPEs. For instance, the SECRET model provides a minimum set of parameters to precisely define windows [31], which can be applied to identify possible differences across systems [32]. Akidau et al. [33] study the semantics of watermarks and the differences between the two system implementations. Gévay et al. [34] survey and analyze different approaches to handle iterations in Dataflow systems. Other works focus on higher-level abstractions on top of Dataflow-based SPEs e.g., through abstractions for relational data processing, exploiting the duality of data streams and relational tables [35]. Fernandez et al. [36] propose an object-oriented programming model that can be automatically translated to a Dataflow graph. Yet other works study how to extend the expressiveness of Dataflow-based SPEs [37]. Naiad [38] offers an extended programming model to control how stream elements traverse nested loops through explicit vector timestamps. CIEL [39] provides primitives to dynamically instantiate new operators, thus allowing for runtime definition of the graph of computations, for instance, based on the value of data. Various proposals also exist to enrich the Dataflow model with a mutable state: TSpoon [40] provides primitives to access operators' state and it lets users specify portions of the graph of computations that need to be accessed/updated with transactional semantics, while S-Store [41] implements an SPE within a relational database core with transactional semantics for stateful operators.

Alternative programming models for stream processing exist [42]. In particular, Complex Event Recognition (CER) [43] considers stream elements as occurrences of events and aims to recognize *patterns* of such events. Interestingly, the semantics of CER operators have been studied in detail [44]: our paper aims to provide a similar contribution to the Dataflow model. In CER, operators' equivalence has been used to rewrite queries to equivalent forms that optimize execution [45]. We believe that a deeper understanding of the semantic relations between operators may enable similar strategies for the Dataflow model, complementing existing optimization strategies [46], exploiting higher-level definitions of queries [47], and streamlining SPE's design and optimization [48].

8 CONCLUSIONS AND FUTURE WORK

Inspired by related work towards unified models for streaming queries, we propose a novel formal approach to reason on the semantics SPEs support (beyond their APIs) by distilling such semantics into minimal sets of operators that allow assessing the effective overlap in the semantic expressiveness of SPEs and their the design, portability, and performance trade-offs.

We show that a single Aggregate operator not only suffices to enforce the semantics of common stateless/stateful Dataflow operators, implying any portion of a query using such operators within an SPE can be ported to other SPEs that support such an Aggregate, but can also support richer analysis (e.g., maintaining arbitrarily long-living state across windows without a dedicated state backend/external storage). Since the Dataflow model [1], which we build on, allows for Aggregates to be run both distributedly and in parallel, our findings can seamlessly rely on existing scale-up and scale-out performance-boosting techniques [1, 3, 5, 6].

We combine analytical insights with an empirical assessment, using diverse hardware setups and state-of-the-art SPEs (Flink/Spark). Our findings show that, within Flink, there exist queries for which compositions of Aggregate operators can perform similarly to other operators, and that our novel μSPE, a micro-SPE with builds on a single Aggregate operator and consists of ~2000 lines of code, maintains equivalent semantic expressiveness outperforming other state-of-the-art SPEs.

Besides extensions studying distributed/parallel compositions of Aggregate operators, we believe our work sets down foundations to show compositions of Aggregate operators can enforce even richer semantics (see § 5.2). Also, the use of a concise abstraction – an operator that encapsulates the semantics of various established operators – can streamline the process for multi-target compilers, enabling them to generate streaming applications suitable for diverse hardware architectures and programming languages, addressing the heterogeneous demands of the edge-to-cloud continuum.

A PROOFS OF THEOREMS AND LEMMAS

Lemma 1 The left boundary $\gamma.l$ of the window instance to which t contributes to can be computed as $\lfloor t.\tau/WA \rfloor WA$ for a tumbling window. If $WA = \delta$, then $\lfloor t.\tau/WA \rfloor = t.\tau/WA$ and thus $\gamma.l = t.\tau$. Moreover, $t_o.\tau$ is set to $\gamma.l + WS - \delta = t_i.\tau$.

Theorem 1 Each tuple t falls in one γ only (Lemma 1). Since all tuples in γ are identical, each tuple $t' \in \gamma$ results in the same set of tuples once fed to f_M, and $\mathbb{T}$ contains the tuples of all such sets or is empty if f_M does not return any tuple. If a γ contains multiple tuples, then such tuples are identical, because A uses all attributes $T(S_{I_1})$ as key-by. If an output t_o is produced, then $t_o.\tau$ for a γ is the same as those contained in $\gamma.\zeta$ (Lemma 1). Hence, A enforces the semantics of E_M.

Theorem 2 By contradiction; if E_J does not enforce the required semantics, then there exists a pair of tuples t_1,t_2 falling in γ_1,γ_2, respectively, so that $f_P(t_1,t_2)$ holds and $\gamma_1.l = \gamma_2.l$ but t_1,t_2 is not added to $\mathbb{T}$ by A_3's f_O. According to the f_O in List.2,L12-21 this can only hold if t_1 or t_2 do not fall in γ_1 or γ_2, respectively, or if one or both tuples are not fed to A_3 in the first place, which contradicts the initial assumption.

Lemma 2 By contradiction; let us assume $t_o^l, t_o^m | t_o^l = t_o^m$ are produced by A_1. Noting that any tuple produced by A_1 for a window instance γ shares the same timestamp of the tuples that fall in γ (see Lemma 1), and noting A_1 does not alter $t_o[1]$ if $t_o[2] \neq -1$ (List.3,L7-8), the existence of t_o^l and t_o^m implies the existence of $t_o^{l'}$ and $t_o^{m'}$ produced by A_1 so that $t_o^{l'}[0{:}1] = t_o^l[0{:}1]$, $t_o^{l'}[2] = 0$, $t_o^{m'}[0{:}1] = t_o^m[0{:}1]$, and $t_o^{m'}[2] = 0$ (List.3,L3-6). Assuming $t_o^{l'}$ and $t_o^{m'}$ are produced by A_1 processing two identical tuples t_i^l and t_i^m from S_E leads to a contradiction, because if $t_i^l = t_i^m$ then both fall in the same window instance γ and γ results only in one output tuple. Assuming $t_o^{l'}$ and $t_o^{m'}$ are produced by A_1 processing two or more input tuples $t_i^{l_1}$,$t_i^{l_2}$,... and $t_i^{m_1}$,$t_i^{m_2}$,... so that $t_i^{l_j} \neq t_i^{m_j}, \forall l_j, m_j$ leads also to a contradiction because $t_i^{l_j} \neq t_i^{m_j}$ implies that the concatenations $t_i^{l_1}[1]$,$t_i^{l_2}[1]$,... and $t_i^{m_1}[1]$,$t_i^{m_2}[1]$,... differ too, and thus that $t_o^{l'} \neq t_o^{m'}$.

Theorem 3 By contradiction; $t^l = \langle \tau^l, \mathbb{T}, -1 \rangle$ with $\mathbb{T} = \{..., t_o^j, ...\}$ is fed to A_1 but A_2 does not output t_o^j. This implies A_2 did not receive $t^* = \langle \tau^l, \mathbb{T}, j \rangle$ **(1)**, or received t^* but as a late arrival that was not processed **(2)**, or received t^* but t^* was not in $\gamma.\zeta[0]$, since $\gamma.\zeta[0]$ is the only tuple considered by A_2's f_O (List.3,L10-12) **(3)**. **(1)** implies the sequence $\langle \tau, \mathbb{T}, -1 \rangle$, ..., $\langle \tau, \mathbb{T}, j-1 \rangle$ was not delivered entirely to A_1. $\langle \tau, \mathbb{T}, -1 \rangle$ being not delivered contradicts the initial assumption. If $\langle \tau, \mathbb{T}, -1 \rangle$ is processed, due to **C1** and **C2** and given that all the tuples in the sequence share the same timestamp, **(1)** results in a contradiction.

(2) implies t^l was fed to A_1. Being γ_l the window instance to which t^l falls in. If t^* was produced by A_1, then A_1 received a watermark $W^m | W^m \geq t^*.\tau + \delta$ ($\gamma_l.\tau = t^*.\tau$, see Lemma 1). Let W^m be the earliest watermark greater than or equal to $t^*.\tau + \delta$, i.e., $W^{m-1} < t^*.\tau + \delta$. **C3** ensures W^{m-1} is the latest watermark fed to A_2, because $t^* \in succ(trig(W^m))$. To be a late arrival for A_2, though, $t^*.\tau < W^{m-1}$, which leads to a contradiction.

(3) implies there exist identical tuples fed to A_2, because A_2 uses all their attributes as key-by, contradicting Lemma 2.

Lemma 3 A watermark W is always forwarded if $W \leq B$ (List.4,L15-18,L20-20). If $B = \infty$, then W is received before any tuple t since B is initialized at ∞, or $succ\Gamma$ is empty. In the first case, no pending tuple from S_{A_2} is yet to be processed. In the latter, any tuple t from E that increased $succ\Gamma[t.\tau]$ when invoking $processT(t)$ has been followed by all $t' \in succ(t)$ that, when invoking $processT(t')$, decreased $succ\Gamma[t'.\tau]$ to 0. Hence, there is no tuple yet to be processed by A_1.

If $B \neq \infty$, then the distance between W and the earliest tuple t' from A_1 yet to be processed by A_1 is smaller than or equal to L'. When t' is received at A_1 after W, the condition $t'.\tau + \delta > W^{\omega}_{A_1} + L'$ will be met (note that based on Lemma 1 t' falls in a window instance whose left boundary is equal to $t'.\tau$), since it implies $W^{\omega}_{A_1} < t'.\tau + \delta + L'$, and the latest watermark W fed to A_1 by S_E is so that $W \leq t'.\tau + L'$.

Lemma 4 If W is forwarded upon its reception (List.5,L19), then $|succ\Gamma| = 0$. Hence, $\forall t \in succ(trig(W'))$ for $W' < W$, t was fed to A_2 before W. All other invocations of $forwardW$ feed A_2 a watermark equal to (List.5,L12) or smaller than (List.5,L15,L22) the τ of an entry in $succ\Gamma$ that is not preceded by entries with a count greater than 0. Hence, τ is fed as watermark to A_2 only after any $t \in succ(trig(W'))$ so that $W' \leq \tau$.

Lemma 5 In Figure 8, the stateless M_1 defines a common set of attributes for the tuples fed to A_1, be they tuples from S_I or S_A. The core functionality is then run by A_1. Based on the Γ defined by A_1 only two consecutive window instances $\gamma_l = [lP, lP+P+\delta)$ and $\gamma_{l+1} = [(l+1)P, (l+1)P+P+\delta)$ overlap on $[(l+1)P, (l+1)P+\delta)$. Nonetheless, tuples falling in the latter interval are only processed when in γ_{l+1} (List.6,L4). Hence, A_1 processes every tuple exactly once. We also note that, as soon as the very first tuple t fed to A_1 is processed, a state tuple t_s is created invoking f_c (List.6,L7). Let γ_l be the first window in which t is processed. The resulting t_s will have $t_s.\tau = (l+1)P$ and will be then processed within the window instance γ_{l+1}, since **C1**, **C2**, and **C3** prevent t_s from not being processed on the basis of being a late arrival. Iteratively, t_s carries a state based on all processed tuples (not only within an individual window instance) from γ_l to all the subsequent windows. Finally M_2 runs f_o on a state tuple with periodicity P, since P is A_1's WA, thus enforcing A_∞'s semantics.

ACKNOWLEDGMENTS

Work supported by the Marie Skłodowska-Curie Doctoral Network project RELAX-DN, funded by the European Union under Horizon Europe 2021-2027 Framework Programme Grant Agreement number 101072456, the Chalmers AoA Energy projects DEEP and INDEED, the Swedish Energy Agency (SESBC) project TANDEM, the Wallenberg AI, Autonomous Systems and Software Program and Wallenberg Initiative Materials for Sustainability project STRATIFIER.

REFERENCES

[1] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Wittle, "The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing," *Proc. Endowment*, vol. 8, no. 12, pp. 1792–1803, 2015.

[2] "Apache Beam," https://beam.apache.org/, accessed:2024-03-14.

[3] "Apache Flink," https://flink.apache.org, accessed:2024-03-14.

[4] "Apache Spark," https://spark.apache.org, accessed:2024-03-14.

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

[6] V. Gulisano, H. Najdataei, Y. Nikolakopoulos, A. V. Papadopoulos, M. Papatriantafilou, and P. Tsigas, "Stretch: Virtual shared-nothing parallelism for scalable and elastic stream processing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 12, pp. 4221–4238, 2022.

[7] "On the Semantic Overlap of Operators in Stream Processing Engines - code repository," https://github.com/vincenzo-gulisano/semantic_overlap, accessed:2024-03-14.

[8] "μSPE," https://github.com/vincenzo-gulisano/muSPE, accessed:2024-03-14.

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

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

[11] B. Gedik, R. R. Bordawekar, and S. Y. Philip, "CellJoin: a parallel stream join operator for the cell processor," *The VLDB journal*, 2009.

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

[13] J. Teubner and R. Mueller, "How soccer players would do stream joins," in *Proc. of the 2011 ACM SIGMOD Int'l Conf. on Management of data*, 2011.

[14] "Apache Storm," http://storm.apache.org, accessed:2024-03-14.

[15] "Liebre SPE," https://github.com/vincenzo-gulisano/Liebre, accessed:2024-03-14.

[16] V. Gulisano, D. Palyvos-Giannas, B. Havers, and M. Papatriantafilou, "The role of event-time order in data streaming analysis," in *Proceedings of the 14th ACM International Conference on Distributed and Event-based Systems*, 2020, pp. 214–217.

[17] M. Faruqui, E. Pavlick, I. Tenney, and D. Das, "WikiAtomicEdits: A Multilingual Corpus of Wikipedia Edits for Modeling Language and Discourse," in *Proc. of EMNLP*, 2018.

[18] I. S. Mohamed, A. Capitanelli, F. Mastrogiovanni, S. Rovetta, and R. Zaccaria, "Detection, localisation and tracking of pallets using machine learning techniques and 2d range data," *arXiv preprint arXiv:1803.11254*, 2018.

[19] ——, "A 2d laser rangefinder scans dataset of standard eur pallets," *Data in Brief*, p. 103837, 2019.

[20] "Apache Flink - Rework streaming iteration flow control," https://issues.apache.org/jira/browse/FLINK-2497?jql=project%20%3D%20FLINK%20AND%20text%20~%20%22loop%20deadlock%22, accessed:2023-1-27.

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

[22] D. Palyvos-Giannas, G. Mencagli, M. Papatriantafilou, and V. Gulisano, "Lachesis: a middleware for customizing os scheduling of stream processing queries," in *Proceedings of the 22nd International Middleware Conference*, 2021, pp. 365–378.

[23] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica, "Apache spark: A unified engine for big data processing," *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, 2016.

[24] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink™: Stream and batch processing in a single engine," *IEEE Data Engineering Bulletin*, vol. 38, no. 4, pp. 28–38, 2015.

[25] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy, "Storm@twitter," in *Proc of the Intl Conf on Management of Data*, ser. SIGMOD '14. ACM, 2014, pp. 147–156.

[26] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, "Twitter heron: Stream processing at scale," in *Proc of the Intl Conf on Management of Data*, ser. SIGMOD '15. ACM, 2015, pp. 239–250.

[27] B. Bejeck, *Kafka Streams in Action: Real-time apps and microservices with the Kafka Streams API*. Manning, 2018.

[28] S. A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, and R. H. Campbell, "Samza: Stateful scalable stream processing at linkedin," *Proc of VLDB*, vol. 10, no. 12, p. 1634–1645, 2017.

[29] R. Soulé, M. Hirzel, R. Grimm, B. Gedik, H. Andrade, V. Kumar, and K.-L. Wu, "A universal calculus for stream processing languages," in *Programming Languages and Systems: 19th European Symposium on Programming, ESOP 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings 19*. Springer, 2010, pp. 507–528.

[30] A. Arasu, S. Babu, and J. Widom, "The cql continuous query language: semantic foundations and query execution," *The VLDB Journal*, vol. 15, pp. 121–142, 2006.

[31] I. Botan, R. Derakhshan, N. Dindar, L. Haas, R. J. Miller, and N. Tatbul, "Secret: A model for analysis of the execution semantics of stream processing systems," *Proc. VLDB Endow.*, vol. 3, no. 1–2, p. 232–243, 2010. [Online]. Available: https://doi.org/10.14778/1920841.1920874

[32] L. Affetti, R. Tommasini, A. Margara, G. Cugola, and E. Della Valle, "Defining the execution semantics of stream processing engines," *Journal of Big Data*, vol. 4, no. 1, pp. 1–24, 2017.

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

[34] G. E. Gévay, J. Soto, and V. Markl, "Handling iterations in distributed dataflow systems," *ACM Comput. Surv.*, vol. 54, no. 9, 2021. [Online]. Available: https://doi.org/10.1145/3477602

[35] M. J. Sax, G. Wang, M. Weidlich, and J.-C. Freytag, "Streams and tables: Two sides of the same coin," in *Proceedings of the International Workshop on Real-Time Business Intelligence and Analytics*, ser. BIRTE '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3242153.3242155

[36] R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch, "Making state explicit for imperative big data processing," in *Proc of the USENIX Annual Technical Conf*, ser. ATC'14. USENIX Assoc., 2014, p. 49–60.

[37] A. Margara, G. Cugola, N. Felicioni, and S. Cilloni, "A model and survey of distributed data-intensive systems," *ACM Comput. Surv.*, vol. 56, no. 1, 2023.

[38] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: A timely dataflow system," in *Proc of the Symposium on Operating Systems Principles*, ser. SOSP '13. ACM, 2013, p. 439–455.

[39] D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand, "Ciel: A universal execution engine for distributed data-flow computing," in *Proc of the Conf on Networked Systems Design and Implementation*, ser. NSDI'11. USENIX Assoc., 2011, p. 113–126.

[40] L. Affetti, A. Margara, and G. Cugola, "Tspoon: Transactions on a stream processor," *Journal of Parallel and Distributed Computing*, vol. 140, pp. 65–79, 2020.

[41] U. Cetintemel, J. Du, T. Kraska, S. Madden, D. Maier, J. Meehan, A. Pavlo, M. Stonebraker, E. Sutherland, N. Tatbul *et al.*, "S-store: a streaming newsql system for big velocity applications," *Proc of VLDB*, vol. 7, no. 13, pp. 1633–1636, 2014.

[42] G. Cugola and A. Margara, "Processing flows of information: From data stream to complex event processing," *ACM Comput. Surv.*, vol. 44, no. 3, 2012. [Online]. Available: https://doi.org/10.1145/2187671.2187677

[43] N. Giatrakos, E. Alevizos, A. Artikis, A. Deligiannakis, and M. Garofalakis, "Complex event recognition in the big data era: A survey," *The VLDB Journal*, vol. 29, no. 1, p. 313–352, jul 2019. [Online]. Available: https://doi.org/10.1007/s00778-019-00557-w

[44] A. Grez, C. Riveros, M. Ugarte, and S. Vansummeren, "A formal framework for complex event recognition," *ACM Trans. Database Syst.*, vol. 46, no. 4, dec 2021. [Online]. Available: https://doi.org/10.1145/3485463

[45] N. P. Schultz-Møller, M. Migliavacca, and P. Pietzuch, "Distributed complex event processing with query rewriting," in *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, ser. DEBS '09. New York, NY, USA: Association for Computing Machinery, 2009. [Online]. Available: https://doi.org/10.1145/1619258.1619264

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

[47] M. Guerriero, D. A. Tamburri, and E. D. Nitto, "Streamgen: Model-driven development of distributed streaming applications," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 1, 2021. [Online]. Available: https://doi.org/10.1145/3408895

[48] P. Pedreira, O. Erling, K. Karanasos, S. Schneider, W. McKinney, S. R. Valluri, M. Zait, and J. Nadeau, "The composable data management system manifesto," *Proceedings of the VLDB Endowment*, vol. 16, no. 10, pp. 2679–2685, 2023.