Aggregates are all you need (to bridge stream processing and Complex Event Recognition)

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

Gulisano, V., Margara, A. (2024). Aggregates are all you need (to bridge stream processing and Complex Event Recognition). DEBS 2024 - Proceedings of the 18th ACM International Conference on Distributed and Event-Based Systems: 66-77. http://dx.doi.org/10.1145/3629104.3666032

(article starts on next page)

Aggregates are all you need (to bridge stream processing and Complex Event Recognition)

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

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

ABSTRACT

Emerging as an alternative to databases for continuous data processing, stream processing has evolved significantly since its inception in the early 2000s, leading to the emergence of numerous Stream Processing Engines (SPEs).

Two main approaches exist to define streaming applications: to explicitly define graphs of common operators (Filters, Maps, Joins, and Aggregates) as the Dataflow model prescribes, or to express patterns of interest based on observations of low-level events within the domain under analysis, known as Complex Event Recognition (CER).

Motivated by SPEs' semantic overlap, recent research has shown Aggregates suffice for an SPE to be as semantically expressive as other SPEs. However, a question remains open: Do Aggregates possess the semantic expressiveness required to cover CER too? We address this question formally demonstrating they indeed hold such semantic expressiveness.

CCS CONCEPTS

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

KEYWORDS

Stream processing, Stream Aggregates, Complex Event Reasoning, Semantic Equivalence

ACM Reference Format:
Vincenzo Gulisano and Alessandro Margara. 2024. Aggregates are all you need (to bridge stream processing and Complex Event Recognition). In *The 18th ACM International Conference on Distributed and Event-based Systems (DEBS '24), June 24–28, 2024, Villeurbanne, France.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3629104.3666032

DEBS '24, June 24–28, 2024, Villeurbanne, France

ACM ISBN 979-8-4007-0443-7/24/06
https://doi.org/10.1145/3629104.3666032

1 INTRODUCTION

Stream processing technologies have continuously evolved over the years [14]. Research in the area focused both on defining programming models and semantics for streaming queries [12, 15] and on building Stream Processing Engines (SPEs) for efficient and scalable query execution [22].

Over the years, the Dataflow model [3] emerged as the de-facto standard for building distributed and parallel stream processing systems. This model defines streaming queries as a directed graph of independent operators. Each operator computes part of a query and feeds the results of its execution to downstream operators for further processing. As operators do not share any state but only communicate by exchanging immutable data, they can be independently deployed on the same or different hosts, possibly in multiple copies, thus enabling the scaling up and out of streaming applications.

The Dataflow model also captures notions of correctness and fault tolerance. It relies on event-time semantics [2], that is, it measures time based on timestamps that are associated with streaming elements by applications and thus reflect the notion of time for the specific application at hand. Dataflow SPEs ensure that the results of the computation are the same as if each input element was analyzed once and only once (exactly-once semantics), even in the presence of failures, and provide support to ensure that out-of-order elements do not result in missing/wrong outputs.

In summary, using the Dataflow model to express streaming queries enables scalable, correct, and fault-tolerant execution on modern stream processing platforms such as Apache Spark [27], Apache Flink [9], or Google Dataflow [3].

Each SPE offers a rich library of common operators, which partially but not completely overlap. This state of things stimulated research on discovering equivalence relations and overlapping between these operators. For instance, Apache Beam [7] offers a unified syntax to express operators that can be later translated to the specific API of individual SPEs that support those operators. More recent work has shown that many operators can be rewritten as compositions of a restricted set of elementary operators and further proved that all common operators found in state-of-the-art SPEs can be expressed as a composition of a single minimalistic Aggregate operator [18].

The studies discussed above only target streaming applications that involve data transformations such as filtering, modification of individual data elements, join, or aggregations, and can be directly expressed using the common operators in the Dataflow model. However, data transformations only cover part of the typical areas of application of SPEs [12]. In particular, stream processing is widely adopted as a tool for monitoring and decision support. In this context, the main task for the SPE is to identify situations of interest starting from observations of low-level events that occur in the domain under analysis, also referred to as Complex Event Recognition (CER) [5, 15]. CER tasks are frequently expressed using high-level languages that declaratively define the temporal patterns of events to be detected. Despite promising investigations on the semantics and expressiveness of these languages [4, 10], a mapping of CER tasks onto the common operators of SPEs has never been discussed from a formal standpoint. Instead, bringing CER tasks onto Dataflow SPEs entails defining custom operators to express pattern detection functionalities that are not directly encoded in common operators [15].

In this paper, we build on the work presented in [18] and investigate whether compositions of Aggregate operators are sufficient not only to cover simple data transformations but also the pattern-detection requirements of CER. Specifically, we observe that CER queries entail three main tasks: (1) analyze individual events or groups of events that take place at the same point in time, for instance, to identify and remove duplicate notifications of the same observations; (2) finding patterns of events that span a period of time of known length, for instance, to identify situations that always last (at maximum) for the same amount of time, but may start at any point in time; and (3) finding patterns of events that span a period of time of unknown length and potentially unbounded, for instance, because the start or the end of the pattern is determined by the arrival of a specific element in the input stream.

We prove that all these three tasks can be expressed as compositions of an Aggregate operator, relying only on basic assumptions of the Dataflow model, namely event-time, watermarks, key-by data partitioning, and basic non-nested loops. This result brings significant consequences from both a theoretical and a practical viewpoint, as it enables CER tasks to be implemented on top of virtually every distributed Dataflow-based SPE, obtaining correctness in terms of exactly-once semantics and event-time order.

An additional key contribution this paper brings to the research on stream processing and CER by demonstrating that CER tasks can be implemented on top of any Dataflow-based SPE that offers an Aggregate operator is that Aggregates have been widely studied since early engines, and features such as parallelism, distribution, elasticity, and fault tolerance are well-supported for them. Hence, proving a certain CER analysis can be expressed by composing Aggregates implies such analysis can seamlessly benefit from elasticity, parallelism, distribution, and fault tolerance in modern SPEs.

Paper organization: § 2 covers preliminaries and our system model and assumptions. § 3-§ 5 each cover one of the complex semantics operators we show can be expressed by composing Aggregate operators: § 3 covers the Sort&Pre-Process *S&P* operator, which can be used to sort and analyze events that, individually or in groups, happen at a given point in time; § 4 covers the Time-based CER (*T-CER*) operator, used to find patterns of known length in time; and § 5 covers the Rule-based CER (*R-CER*) operator, used to find patterns whose start/end time and whose duration are not known a priori but rather a function of the data itself. § 6 discusses related work. § 7 concludes the paper.

2 PRELIMINARIES AND SYSTEM MODEL

2.1 Stream processing basics

Since we build on the work from [18], we rely on an equivalent system model for stream processing (presented next) with a minor addition compared to the original model.

A *stream* S is an unbounded sequence of *tuples*. Each tuple $\langle \tau, \phi \rangle$ carries a timestamp τ and a payload ϕ. We do not impose any restriction on how many attributes ϕ carries nor their type, but we assume the payloads of two tuples t_1, t_2 are comparable to assert whether $t_1.\phi = t_2.\phi$ and assume t_1, t_2 are equal (i.e., $t_1 = t_2$) if $t_1.\tau = t_2.\tau \wedge t_1.\phi = t_2.\phi$. We use the notation $t.\phi.x$ to refer to the attribute x carried by t in its payload ϕ. We say ϕ' encapsulates ϕ if ϕ is an attribute of ϕ'. We use `null` to refer to uninitialized payloads or uninitialized payload's attributes and write $X \leftarrow \{a_1{:}v_1, a_2{:}v_2, \ldots\}$ to say object X is initialized with attribute a_1 carry value v_1, attribute a_2 carry value v_2, and so on.

Queries are composed of *operators*, connected in a directed graph, that process and forward/produce tuples. For a 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 [13]), where δ represents the time granularity of the underlying SPE.

We consider both common *stateless* and *stateful* operators. Operators like Map (M) and Filter (F) are stateless and do not maintain a state that evolves based on the tuples they process.

Map $S_O \leftarrow M(S, f_M)$ processes the tuples from S with function f_M to produce stream S_O. Function f_M is invoked on each $t_i \in S$ to produce zero, one, or more t_o output tuples. Note $t_o.\tau = t_i.\tau$ for a t_o produced from t_i.

Filter $S_O \leftarrow F(S, f_F)$ forwards each $t_i \in S$ to S_O if $f_F(t_i)$ holds. Note $t_i = t_o$ for a t_o output by processing t_i.

Stateful operators produce results from a state dependent on a set of tuples. In this paper, we consider Aggregates [13, 21, 26] defined over delimited groups of tuples called *time-based windows* (or simply windows). We denote as $\Gamma(WA, WS, S, f_K, L)$ a window specified as follows:

Window Advance (WA) and Window Size (WS) are the parameters that define the event time periods covered by Γ: $[\ell WA, \ell WA + WS)$, with $\ell \in \mathbb{N}$. We refer to one such event time period 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 exactly one γ.

Input stream S is the input stream fed to Γ.

Key-by function f_K returns a numerical key for a tuple t. Dedicated γs are then maintained for tuples sharing the same key.

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

Aggregate operators, defined next, maintain a Γ and assign their input tuples from S to Γ's instances γ depending on the WA, WS, f_K, and L parameters. In the remainder, we use the following notation for a specific γ:

- $\gamma.k$ refers to the key associated to γ. All tuples falling in a given γ share the same key. That is, if t_1 and t_2 fall in the same γ, then $f_K(t_1) = f_K(t_2)$.
- $\gamma.l$ refers to the left boundary (inclusive) of γ. The right boundary (exclusive) can be computed as $\gamma.l + WS$.
- $\gamma.\phi$ refers to the state maintained by the γ. This state depends on user-defined functions (presented next) and could e.g., maintain the input tuples falling in γ, to aggregate them later once an output tuple is to be produced from γ, or an incrementally-aggregated value, for instance, a counter used to output the number of tuples falling in γ.

As common in related works [7, 18, 26] and adopted by state-of-the-art SPEs like Apache Flink [13], $t_o.\tau$ – the timestamp of an output tuple t_o created from a given γ – is set to reflect the maximum allowed timestamp for the tuples falling in such γ, i.e., $\gamma.l + WS - 1$ (since the right boundary of γ is exclusive). For a given t_o, the event time period covered by the γ from which t_o is output is then $[t_o.\tau - WS + 1, t_o.\tau + 1)$.

When assigning tuples to Γ's instances, note that, on the one extreme, f_K can be defined so that all tuples from S fall in a single γ for a given event time period. On the other extreme, f_K can be defined so that only identical tuples share the same γ for a given event time period [18]. The former can be achieved by an f_K, which we refer to as $f_{K^\varnothing}$, that returns always the same key, e.g., the value 0, independently of the tuple fed to it. The latter can be achieved by an f_K, which we refer to as f_{K^*}, that hashes both $t.\tau$ and $t.\phi$.

We define the Aggregate operator as:

$$S_O \leftarrow A(\Gamma(WA, WS, S, f_K, L), f_U, f_O)$$

Function $f_U(\gamma, t)$ is invoked for each window γ and input tuple t (falling in γ) to update $\gamma.\phi$. Function $f_O(\gamma)$ is invoked to compute the ϕ payloads of the output tuples from γ and to forward such output values if the set returned by f_O is not empty.

Note that the semantics of a given A can be achieved by different combinations of f_U/f_O. To exemplify this, we present two sample As that show two possible ways in which f_U/f_O can be defined to aggregate S tuples by computing the first and third quartiles of their $t.\phi.v$ values over a given Γ. The first example aims at minimizing the per-tuple processing cost and only aggregate data on output production. This A is presented in Listing 1.

Listing 1: Sample A computing the first and third quartiles of $t.\phi.v$ values upon output production.

$S_O \leftarrow A(\Gamma(WA, WS, S, f_K, L), f_U, f_O)$, where:
1 **Function** $f_U(\gamma, t)$
2   **if** $\gamma.\phi$ = null **then** $\gamma.\phi \leftarrow \{V{:}\emptyset\}$
3     $\gamma.\phi.V \leftarrow \gamma.\phi.V \cup t.\phi.v$ // Store $t.\phi.v$ in $\gamma.\phi.V$
4 **Function** $f_O(\gamma)$
5     $\gamma.\phi.V \leftarrow$ sort($\gamma.\phi.V$)
6   **return** $\{fq$:firstQuart($\gamma.\phi.V$), tq:thirdQuart($\gamma.\phi.V$)$\}$

Upon reception of a tuple t, f_U is used to initialize an empty set V if t is the first tuple falling in γ (List.1,L2). Subsequently, $t.\phi.v$ is added to such V (List.1,L3). When an output tuple is to be created from γ, the respective payload is computed by f_O defining two values fq and tq, the first and third quartile computed from V (sorted), respectively, (List.1,L5-6). In the example, the two quartiles are computed with the auxiliary functions `sort`, `firstQuart`, and `thirdQuart`.

The second example aims to minimize the time required to produce an output tuple. As shown in Listing 2, f_U is in this case defined so that $\gamma.\phi.V$ maintains $t.\phi.v$ values sorted (List.2,L3) and so that the first and third quartiles, maintained in $\gamma.\phi.fq$ and $\gamma.\phi.tq$, respectively, are updated for each new tuple falling in γ (List.2,L4-5). When a tuple is to be output from a γ, the time required to do so is then minimized since f_O can simply forward $\gamma.\phi.fq$ and $\gamma.\phi.tq$.

In the following, we use the notation $f \leftarrow f'$ for the functions of a Γ or an operator when f is optional and, if not defined, is then set to f'.

Concerning the additions our model has compared to the original one [18], namely f_U, note that f_U allows updating a γ's state continuously, as tuples are added to it, but does

Listing 2: Sample A computing the first and third quartiles of $t.\phi.v$ values (the actual values are computed incrementally upon the reception of each new tuple).

$S_O \leftarrow A(\Gamma(WA, WS, S, f_K, L), f_U, f_O)$, where:

```
Function f_U(γ, t)
    if γ.ϕ = null then γ.ϕ ← {V:∅, fq:0, tq:0}
    γ.ϕ.V ← sortInsert(γ.ϕ.V, t.ϕ.v) // Store t.ϕ.v in γ
    γ.ϕ.fq ← firstQuart(γ.ϕ.V) // Update fq
    γ.ϕ.tq ← thirdQuart(γ.ϕ.V) // Update tq
Function f_O(γ)
    return {fq:γ.ϕ.fq, tq:γ.ϕ.tq}
```

not imply a higher expressiveness than that of an A that, as in [18], only defines an f_O function. This is because any operation performed by f_U can also be later performed by f_O once all the tuples falling into a given γ have been added to such γ.

2.2 Correctness conditions

When deploying and running M, F, and A operators, users expect SPEs to enforce such operators' semantics correctly. Since M and F do not maintain a tuple-dependent state, correct semantics are enforced by processing each tuple exactly once. The operator A requires greater care, though. Leaving aside late arrivals (discussed next), its correct execution requires f_U to be invoked exactly once on each tuple t to update the state of the γs to which t falls into, and f_O to be invoked exactly once on each γ once such γ contains all the tuples that should be aggregated together depending on Γ's definition.

While f_U can be immediately invoked by an A upon the reception of a tuple t, how can an A know when a certain γ contains all the tuples falling in it and can be thus passed to f_O? This is achieved with the support of *watermarks* [19]:

Definition 1. The watermark W_A^ω of A at wall-clock time ω is the earliest event time a tuple t_i fed to A can have from time ω on (i.e., $t_i.\tau \geq W_A^\omega, \forall t_i$ processed from ω on).

In the literature [13, 19], watermarks are commonly maintained assuming data sources periodically output watermarks as special tuples to notify how event time advances. Upon receiving a watermark, A stores the watermark's time, updates W_A^ω to the smallest of the latest watermarks received from each upstream peer, and propagates W_A^ω. Upon an increase of W_A^ω and before forwarding W_A^ω, A invokes f_O on any $\gamma | \gamma.l + WS \leq W_A^\omega$, in $\gamma.l$ order, forwarding the resulting output tuples in timestamp order and discarding such a γ since no more tuples will fall in it.

Handling late arrivals. As aforementioned, Γ defines L to handle late arrivals. More concretely, by delaying the purging of γs by L. Tuple t is a late arrival for A if $t.\tau < W_A^\omega$ when, at time ω, A processes t. According to the Dataflow model [3], t is processed, added to γ, and can result in an output tuple (potentially an update of a previous output tuple) if $\gamma.l + WS \leq W_A^\omega + L$ at ω. Note that, if $L > 0$ and watermarks are forwarded by A as described in § 2.2, results produced by A could be late arrivals for A's downstream peers. Also, note we account for parameter L since such a parameter is needed to support loops [18], but we do not further use it to define the operators we present in § 3-§ 5. We thus omit L from Γ in the reminder.

2.3 System model

With this work, we aim at formally showing that A operators are sufficient to enforce the semantics of CER, namely: preprocessing data (§ 3), finding patterns of events spanning a period of known maximum duration (§ 4), and finding patterns of events spanning a period of unknown duration (possibly unbounded) and location (§ 5). Note that it is not within the scope of our contribution to account for performance- and implementation-specific optimization, which we plan to focus on in future work. We consider SPEs for which the following can hold:

A1 A stream can feed one or more A operators, delivering the same tuples/watermarks in the same order.
A2 An A operator can iterate over its outputs with a loop.
A3 Each stream S delivers watermarks with a max event time distance D_W between W^i and W^{i+1}. If the first tuple $t^0 \in S$ precedes the first watermark W^0, then $W^0 - t^0.\tau \leq D_W$.

For **A2**, note that if an output tuple t_o, with $t_o.\tau = \gamma_i.l + WS - \delta$, is fed back to A immediately once A's watermark has been updated to the W_A^ω that results in t_o's production, then t_o falls into γ_i but constitutes a late arrival for A. As in [18], we assume A handles watermarks and late arrivals so that all looping tuples are processed, that any output tuples resulting from such processing are forwarded to A's downstream peers, and that A's watermarks are forwarded to downstream peers preventing A's output tuples from being late arrivals for the latter. We refer to [18] for a detailed discussion about how this can be done.

About **A3** note that if an A operator is fed an input stream for which **A3** holds, a distance D_W exists for A's output too. By extension, it also holds for all A operators of a query composed exclusively of A operators.

Note **A1**-**A3** are only needed for the operator in § 5. If streams $S_{I_1}, S_{I_2}, \ldots$ are fed to A, we write $\{S_{I_1}, S_{I_2}, \ldots\}$ to refer to the merged stream fed to A.

Finally, note that F and M have already been proven to be expressable as compositions of A operators. As such, relying on F and M operators is not in contradiction with our goal of showing how complex semantics can be enforced solely by composing A operators.

Before presenting our operators, note the following sections are structured as follows: *Definition* describes the desired semantics of the operator, *Challenges and Intuition* reasons about how the desired semantics can be achieved by relying on A operators, *Solution* contains the formal solution, while *Explanation and Proof* describe the proposed solution and provides a formal proof.

3 SORT AND PRE-PROCESS ($S\&P$)

With this operator, we show how events carried by the tuples of a stream can be sorted and pre-processed (e.g., to clean them before further processing them). Such pre-processing can e.g., shape the *selection policy* [23] dictating how events are chosen to participate in pattern detection and which events are eligible for inclusion.

For instance, when multiple instances of the same type of event occur, potentially fitting the criteria for a complex event pattern, the $S\&P$ operator can be employed to sort and remove duplicate tuples within a stream S (i.e., remove $t' \in S$ if $\exists t \in S$ so that $t = t'$). Removing duplicate events is common in CER systems: indeed, the same event may be captured from multiple sensors, resulting in duplicate notifications within the system. Likewise, different CER queries may derive the same observation and produce equivalent events in the output: in these cases, discarding redundant event derivations based on some application-level notion of equivalence is key to providing a concise view of the status of the domain under analysis to end users.

Definition. With the $S\&P$ operator, defined as:

$$S_O \leftarrow S\&P(S, f_P, f_K \leftarrow f_{K^*})$$

a user is interested in sorting the tuples and in pre-processing together, with function f_P, tuples that share the same timestamp and key (i.e., tuples that are equivalent according to the user/application). Such a key can be defined by the user with the f_K function. Alternatively, f_K is set to f_{K^*} by default (see § 2.1) implying two tuples t_1, t_2 are pre-processed together only if $t_1 = t_2$.

Challenges and intuition. Every tuple in $t \in S$ can be potentially forwarded as is or result in the production of an output tuple with a different payload once pre-processed by f_P, and possibly after being re-ordered, but carrying the same $t.\tau$. If this is to be achieved by relying on an A able to "forward" tuples from S, or transformations of such tuples, to its output stream, we can note that f_U can store incoming tuples in its $\gamma.\phi$ and later feed them to f_P upon invocation of f_O. To match the timestamp of output tuples with that of input tuples, we can rely on a tumbling Γ with WS and WA equal to δ. By doing this, each tuple t_i will fall exactly in one γ so that $\gamma.l = \lceil \frac{t_i.\tau}{WA} \rceil WA = t_i.\tau$ (note $\lceil \frac{t_i.\tau}{WA} \rceil = \frac{t_i.\tau}{WA}$ if $WA = \delta$, because δ is the event-time granularity of the SPE under consideration, see § 2.1) and the corresponding output will have a timestamp $t_o.\tau = \gamma.l + WS - \delta = \gamma.l = t_i.\tau$.

We note that, based on f_K, if a γ contains more than one tuple, then such tuples share the same key (according to the user-defined f_K or f_{K^*}) and are to be pre-processed together.

Solution. Based on the aforementioned intuition, we can now state the following theorem.

Theorem 1. *$S\&P$'s semantics can be enforced by an A operator as specified in Listing 3.*

Listing 3: A implementing $S\&P$

$S_O \leftarrow A(\Gamma(\delta, \delta, S, f_K \leftarrow f_{K^*}), f_U, f_O)$, where:
1 **Function** $f_U(\gamma, t)$
2 **if** $\gamma.\phi =$ `null` **then** $\gamma.\phi \leftarrow \{T{:}\emptyset\}$
3 $\gamma.\phi.T \leftarrow \gamma.\phi.T \cup t$ `// Store t in γ`
4 **Function** $f_O(\gamma)$
5 **return** $f_P(\gamma.\phi.T)$ `// Return the output from fP`

As shown, each tuple t falling in a given γ is stored upon invocation of f_U, once the state is initialized with a single attribute T (initially an empty set) when the very first tuple falling in γ is received (List.3,L2-3). The output(s) from f_P are then returned upon invocation of f_O (List.3,L5).

After covering Listing 3, we formally prove Theorem 1.

Proof. (Theorem 1) By contradiction, we assume t_o is an output tuple produced by f_P from a set T' of tuples fed to $S\&P$ sharing the same timestamp τ' and key k', but t_o is not produced (**C1**), produced more than once (**C2**), or produced with the wrong timestamp/payload (**C3**), where **C1** excludes **C2**, but **C2** and **C3** could be observed at the same time.

Since f_O (List.3,L5) is always invoked exactly once by A, **C1**-**C3** can only be ascribed to $\gamma.\phi.T$ not containing the exact same tuples contained in T'.

All ts in $\gamma.\phi.T$ share the same timestamp (because $WA = WS = \delta$) and key, and there is a γ for each possible value of $\gamma.l$ (because $WA = \delta$). Hence, if $\gamma.l = \tau'$ and $\gamma.k = k'$, then $\gamma.\phi.T$ cannot contain tuples other than that in T'. Moreover, f_O is only invoked on γ when $W_A^{\omega} > \gamma.l = \tau'$, so all tuples in T' are also in $\gamma.\phi.T$, which leaves as only option for **C1**-**C3** that for which not all tuples from T' are fed to $S\&P$, which contradicts the initial assumption. □

4 TIME-BASED CER (T-CER)

With this operator, we exemplify CER queries having known maximum time boundaries. These queries are frequent in CER, as they enable capturing situations of interest in which a set of events takes place within a maximum window of time: for instance, in environmental monitoring applications, physical phenomena may be derived from the observation of many anomalous event notifications received within a predefined period of time. In fact, predicating on temporal patterns and their duration is considered one of the base features that most CER languages offer [5, 10].

Definition. With the *T-CER* operator, defined as:

$$S_O \leftarrow T\text{-}CER(S, f_K \leftarrow f_{K^\varnothing}, D_C, f_C)$$

a user is interested in finding, within stream S, one or more time-bound sequences $P = \{t_a, \ldots, t_z\}$ for tuples:

- sharing the same value of a key-by function f_K,
- so that $\max_\tau P - \min_\tau P < D_C$, and
- for which a condition checked by f_C holds.

T-CER should report each found sequence exactly once. Note that the default value for f_K, $f_{K^\varnothing}$ (see § 2.1), implies all tuples from S are to be checked together. The function f_C is used to check, incrementally, if the user-defined condition holds while being fed a set of tuples T that potentially contains P. More concretely, $m, \Phi_O \leftarrow f_C(m, t)$, once fed a state m (initially `null`) and a tuple t, returns an updated state m and a set Φ_O (possibly empty) of ϕs, one for each sequence found upon the processing of t, with m holding the information about tuples processed before t and possibly part of one or more sequences P. We assume each such ϕ defines at least the attributes $\phi.\min_\tau P$ and $\phi.\max_\tau P$ for each sequence P found in T.

Challenges and intuition. Intuitively, knowing that P cannot span more than D_C event-time units implies that a Γ with $WS = D_C$ could have a γ so that P falls entirely in such γ. Note, though, we do not know the exact location in event-time of P, and näively going for a Γ with $WA = \delta$ and $WS = D_C$, which would ensure that the D_C period in which P is located falls entirely in at least one γ since Γ has γs starting at each event-time value, could be too costly. This is because this would imply each tuple t falls in $\lceil \frac{D_C}{\delta} \rceil$ γs. If D_C is 1 hour and δ is 1 ms (e.g., as in Apache Flink [13]), as an example, each t would contribute to 3.6×10^6 γs.

We can note, though, that by selecting WA/WS so that:

$$WA \leq (WS - D_C) + \delta \wedge WS \geq D_C$$

each sequence P falls entirely in at least one γ (as formally proved later). By defining a sliding window Γ of $WA = 0.5D_C$ and $WS = 1.5D_C$, for instance, any P, from the ones for which $\min_\tau P = \max_\tau P$ to the ones for which $\max_\tau P - \min_\tau P = D_C - \delta$, falls in 1 to 3 γs, as shown in Figure 1.

Note that, if P falls in more than 1 γ, it should nonetheless be reported only once according to *T-CER*'s definition. In this case, each γ_i, knowing its boundaries, could check if the subsequent γ_{i+1} also contains P in its entirety, and in such a case defer the output of the payloads in Φ_O to such γ_{i+1}.

Solution. Based on the aforementioned intuition, we can now state the following theorem.

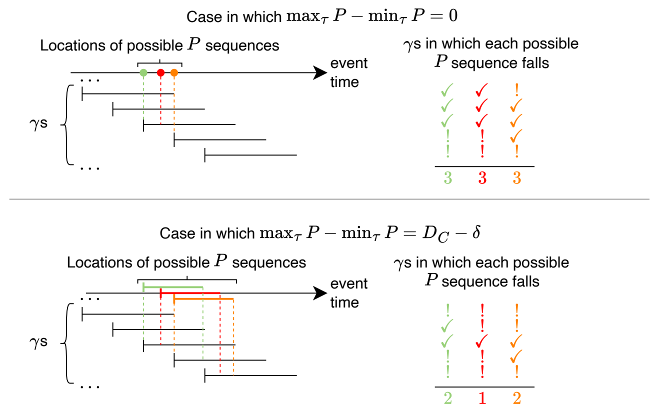

Figure 1: Example showing how a sequence P falls into 1 to 3 γs for a Γ with $WA = 0.5D_C$ and $WS = 1.5D_C$.

Theorem 2. *T-CER's semantics can be enforced by an A operator as specified in Listing 4.*

Listing 4: *A* implementing *T-CER*

$S_O \leftarrow A(\Gamma(WA, WS, S, f_K \leftarrow f_{K^\varnothing}), f_U, f_O)$, for WA and WS so that $WA \leq (WS - D_C) + \delta \wedge WS \geq D_C$ and where:

1 **Function** $f_U(\gamma, t)$
2   **if** $\gamma.\phi$ = null **then** `// setup state if` ϕ `unintialized`
3         $\gamma.\phi \leftarrow \{m{:}\text{null}, \Phi_O{:}\emptyset\}$
4     $\gamma.\phi.m, \Phi'_O \leftarrow f_C(\gamma.\phi.m, t)$ `// get new state/outputs`
5   **for** $\phi_o \in \Phi'_O$ **do** `// store each` ϕ_o `not falling in next` γ
6   **if** $\neg(\phi_o.\min_\tau P \geq \gamma.l + WA \wedge \phi_o.\max_\tau P < \gamma.l + WA + WS)$ **then**
7             $\gamma.\phi.\Phi_O \leftarrow \gamma.\phi.\Phi_O \cup \phi_o$
8 **Function** $f_O(\gamma)$
9   **return** $\gamma.\phi.\Phi_O$ `// Return all previously stored payloads`

Explanation and Proof. Before our proof about the correctness of the proposed solution, we can note from Listing 4 that, upon the reception of a tuple t and the invocation of f_U, A initializes its state to keep a `null` value that represents f_C's internal state and an initially empty set of output payloads if t is the first tuple falling in a given γ (List.4,L2-3). It then retrieves the value of f_C's internal state, passes it to f_C together with t, and retrieves the updated value of f_C's internal state, which it stores back in $\gamma.\phi$, and a set (possibly empty) of output payloads Φ'_O (List.4,L4). According to *T-CER*'s definition, each $\phi_o \in \Phi'_O$ carries two attributes, $\phi_o.\min_\tau P$ and $\phi_o.\max_\tau P$. For each $\phi_o \in \Phi'_O$, if ϕ_o does not fall in the subsequent window, which covers $[\gamma.l + WA, \gamma.l + WA + WS)$, ϕ_o is then stored in $\gamma.\phi.\Phi_o$ (List.4,L5-7).

Upon invocation of f_O, all the payloads previously stored during the invocations of f_U are returned for the SPE to forward them to the A's downstream peers (List.4,L9).

After covering Listing 4, we now formally prove Theorem 2.

Proof. (Theorem 2) By contradiction, if the proposed solution does not meet *T-CER*'s definition, then there exists a sequence P for which the user-defined condition checked by f_C holds but for which no output or more than one output is produced. Before discussing these two cases, note that if $WA \leq (WS - D_C) + \delta$ and $WS \geq D_C$, then $WS - WA$ is minimized when $WA = WS - D_C + \delta$.

Let's begin by assuming no output is produced. Then, there exists a sequence P that falls across all γs. Let's denote as γ_i one such γ and assume P falls across γ_i's left boundary. That is, $\min_\tau P < \gamma_i.l \wedge \max_\tau P \geq \gamma_i.l$. If $\gamma.l - WA \leq \min_\tau P < \gamma_i.l$, then $\gamma.l - WA + (D_C - \delta) \leq \min_\tau P + (D_C - \delta) < \gamma_i.l + (D_C - \delta)$. Since $\max_\tau P \leq \min_\tau P + (D_C - \delta)$, then $\max_\tau P < \gamma_i.l + (D_C - \delta)$, and $\gamma_i.l + (D_C - \delta) \leq \gamma_i.l + WS - WA$ (equal if $WA = WS - D_C + \delta$). Hence, P falls in $\gamma_{i-1} = [\gamma_i.l - WA, \gamma_i.l - WA + WS)$. Similarly, if $\gamma.l - 2WA \leq \min_\tau P < \gamma_i.l - WA$, then P falls in $\gamma_{i-2} = [\gamma_i.l - 2WA, \gamma_i.l - 2WA + WS)$, and so on. A similar argumentation can be made if P falls across γ_i's right boundary (i.e., if $\min_\tau P < \gamma_i.l + WS$ and $\max_\tau P \geq \gamma.l + WS$) showing that if $\gamma_i.l + WS \leq \max_\tau P < \gamma_i.l + WA + WS$, then P falls in $\gamma_{i+1} = [\gamma_i.l + WA, \gamma_i.l + WA + WS)$, and so on.

Moving now to the case in which more than one output is produced, we can assume at least two outputs are produced. For this to happen, P must fall in its entirety in two γs but, according to List.4,L5-7, these γs cannot be consecutive. Being γ_i, γ_{i+1}, γ_{i+2}, ... a series of consecutive γs, if P cannot fall in γ_i and γ_{i+1}, it must then fall in γ_i and γ_{i+2}, or in γ_i and γ_{i+3}, or in any pair γ_i and γ_{i+j} so that $j > 1$ and so that a portion of shared event-time exists for γ_i and γ_{i+j}. Notice, though, that the period of event-time shared by γ_i and γ_{i+2} is $[\gamma_i.l + 2WA, \gamma_i.l + WS)$, which nonetheless is also shared by γ_{i+1} that covers $[\gamma_i.l + 2WA, \gamma_i.l + 2WA + WS)$, thus contradicting the need for P not to fall in two consecutive γs. □

5 RULE-BASED CER (*R-CER*)

By extending the *T-CER* operator from § 4 to sequences in stream S covering a period of time for which a max duration is not known a priori, the *R-CER* operator can be used to:

- Discard all out-of-order tuples. The relevant sequences in S would then be the ones with non-decreasing timestamps.
- Forward tuples from S only after a given criterion has been met. More concretely, forward tuples from t_i onward only if a given criterion has been met over tuples $t_0, \ldots, t_{i-1} | t_j.\tau \leq t_i.\tau, \forall j \in [0, i-1]$. This would cover the case in which the relevant sequence in S is open-ended on its right boundary.
- Similarly, discard all tuples from t_i onward if a given criterion has been met. This would cover the case in which the relevant sequence in S is open-ended on its left boundary.
- Implement *consumption policies* [23] by keeping track of whether a certain event e has been observed in S and, subsequently, which later events should be considered part of the pattern initiated by such an event e.

Definition. With the *R-CER* operator, defined as:

$$S_O \leftarrow R\text{-}CER(S, f_k \leftarrow f_{K^\varnothing}, f_{M^*}, f_T \leftarrow f_{T^*})$$

a user is interested, for a given stream S partitioned according to a key-by function f_K, in applying a stateful function f_{M^*} on each of S' tuples, traversing them in event-time order, breaking ties on the order of tuples sharing the same timestamp with the function f_T. Being stateful, f_{M^*} processes each tuple t together with an evolving state that can account for all tuples before t (traversed based on their event time and f_T-order). Note the default value for f_K is $f_{K^\varnothing}$ and it implies that all tuples in S are to be processed with a single overall state. The function f_{T^*}, the default value for f_T, implies tuples sharing the same timestamps can be simply traversed in their arrival order. Note the output tuples resulting from the processing of an input tuple should share the same timestamp as such an input tuple.

Similarly to the *T-CER* operator (see § 4), the function $m, \Phi_O \leftarrow f_{M^*}(m, t)$ accepts a state m (initially `null`) and a tuple t as input parameters, and returns the updated state m and a set (possibly empty) of output payloads Φ_O.

While leaving to users the definition of the evolving state supporting the analysis of a *R-CER* operator, we note its space complexity should not grow linearly with the number of tuples being processed, since they are unbounded according to S' definition (see § 2.1). We assume **A1**-**A3** (see § 2.3) to hold.

Challenges and intuition. To begin with, we note each tuple in S can result in one or more output tuples that share the same timestamp. This, though, cannot be achieved using the same intuition we used for the *S&P* operator (i.e., that of relying on a tumbling Γ with $WA = \delta$ and $WS = \delta$, see § 3) since the lack of overlapping between γs would challenge the sharing of an evolving state across such γs.

First, we cannot set f_K to f_{K^*} since, in this case, f_K is defined by the user.

By relying on a Γ with $WA = \delta$ and $WS > \delta$, though, we would have Γ covering sequences in S starting at each possible event-time. Each γ could then be responsible for the output tuples of a given τ. More concretely, the γ whose right boundary is $n\delta$ could be responsible for storing the tuples t so that $t.\tau = (n-1)\delta$ and, accordingly, output tuples sharing

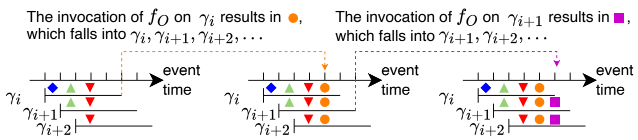

Figure 2: Example providing an intuition about how the results from γ_i, γ_{i+1}, and γ_{i+2} could be used, via a loop, to share information across such γs.

t's timestamp. In this case, the evolving state could be shared across γs by leveraging a loop as exemplified in Figure 2. Note though that, by setting $WA = \delta$ and $WS > \delta$, each t falls in several γs, while the results originating from such t should be output only once (by the γ producing results that share the same timestamp as t).

An additional challenge must be accounted for about how to share state across γs. If we consider the γ that contains the earliest t from S (or one of them if multiple ts share the same timestamp) we can intuitively imagine such a tuple, with others in γ, can be used by f_{M^*} to create the first m shared with subsequent γs upon the invocation of f_O on γ. Such subsequent γs, though, (1) might be fed tuples and (2) might be fed to f_O before the evolving state m is passed to them. In the example from Figure 2, for instance, the input tuple ▼ is already added to γ_{i+2} before the watermark triggering the invocation of f_O on γ_i is fed to A maintaining γ_i, γ_{i+1}, and γ_{i+2}.

Observing that a γ, by itself, might not have access to enough information, based on its left and right boundaries, to know if it is the γ containing the very first tuple fed by S[1], this raises a question: *if a γ is fed a tuple t before being fed a state m, is that because γ is the one containing the earliest tuple in S or is it because the m from a previous γ' has not been fed to γ yet?* The ability to distinguish between these two cases allows us to distinguish whether, upon the invocation of f_O on γ, the output payloads and state from γ can be immediately output since γ contains the earliest tuple from S, or whether the output payloads and state forwarding from γ should be delayed until a previous state being fed through the loop reaches γ. This can be achieved as exemplified in Figure 3 and explained next.

The idea is to rely on a sliding Γ with $WA = \delta$ and $WS = D_W + \delta$, where D_W is the max event-time distance between consecutive watermarks, and the τ attribute of the first tuple and the first watermark fed by S (**A3**, see § 2.3). As shown in Figure 3, the first tuple t_0 from S falls in the γs covering $[t_0.\tau - D_W, t_0.\tau + \delta)$, ..., $[t_0.\tau, t_0.\tau + D_W + \delta)$. W', the first

[1]Except for the γ covering $[0, WS)$, which nonetheless is sure to exist only under a too-strict assumption on the τ carried by the first tuple delivered by S.

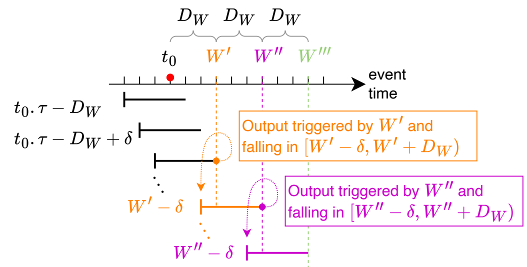

Figure 3: Example showing how a sliding Γ with $WA = \delta$ and $WS = D_W + \delta$ ensures only the γ containing the very first tuple from S is fed such tuple before other tuples or before states shared by previous γs.

watermark following t_0, cannot be far apart from t_0 more than D_W event-time units (**A3**). Hence, upon reception of W', it is possible to output a tuple for the γ covering $[t_0.\tau - \delta, t_0.\tau + D_W)$, which contains t_0 and thus allows for the forwarding of a state that depends on t_0 to contribute to the γ covering $[W' - \delta, W' + D_W)$, among others. Upon reception of W'', the watermark following W', which cannot be far apart from W' more than D_W (**A3**), it is possible to output a tuple for the γ covering $[W'' - \delta, W'' + D_W)$ with $W'' - \delta > W' + D_W$, among others, forwarding the state from t_0 to the following γ, and so on.

Within such a setup, note that, being t_1 the first tuple fed by S so that $t_1.\tau > t_0.\tau$, if t_1 falls in between t_0 and W', then t_1 falls into a γ that observes t_0 too, and that can be thus identified as a γ that is not responsible for the storing of the first tuple t_0 from S. If t_1 falls between W' and W'', it then falls into a γ that contains a state created from a γ that contained t_0. Hence, it is possible, in this case too, to identify the γ to which t_1 falls as a γ that does not contain the first tuple from S, and so on. A similar reasoning applies to t_2, the first tuple fed from S so that $t_2.\tau < t_1.\tau$, t_3, and subsequent tuples.

To complete our discussion about the challenges underlying the *R-CER* operator, note that for a loop to be used to share state across γs, A needs to define for its input tuples a payload ϕ that carries both the payload of S tuples and state-related attributes, while the output S_O should only carry the τ together with the payloads defined by f_{M^*}. This can be achieved by relying on additional F and M operators as shown in Figure 4. Note that, as aforementioned, F and M operators have already been proven to be expressable using A operators in [18].

Solution. Based on the aforementioned intuition, we can now state the following theorem.

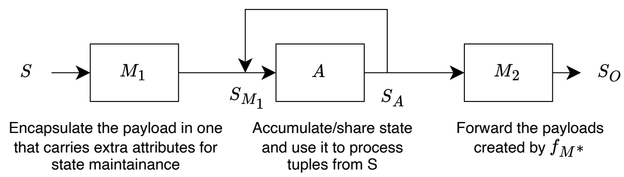

Figure 4: Graph showing how M and A operators can be composed to enforce the semantics of *R-CER* and their main tasks.

Theorem 3. *R-CER's semantics can be enforced by the operators shown in Figure 4 and specified in Listing 5.*

Listing 5: *M*s, *F*, and *A* implementing *R-CER*

$S_{M_1} \leftarrow M(S, f_M)$, where: // M_1-Fig. 4
1 **Function** $f_M(t)$
2 $\quad \phi' \leftarrow \{\phi{:}t.\phi, m{:}\texttt{null}, k{:}f_K(t), fromS{:}\texttt{True}\}$
3 $\quad$ **return** ϕ' // Encapsulate ϕ and add attributes used by A
$S_A \leftarrow A(\Gamma(\delta, D_W + \delta, \{S_{M_1}, S_A\}, f_{K'}), f_U, f_O)$, where: // A-Fig. 4
4 **Function** $f_{K'}(t)$
5 $\quad$ **return** $t.\phi.k$
6 **Function** $f_U(\gamma, t)$
7 $\quad$ **if** $\gamma.\phi = \texttt{null}$ **then** // Initialize state
8 $\quad\quad \gamma.\phi \leftarrow \{first\gamma{:}\texttt{True}, T{:}\emptyset, m{:}\texttt{null}, m\tau{:}\texttt{null}, pending{:}\texttt{True}\}$
9 $\quad$ **if** $t.\phi.fromS$ **then** // t comes from S
10 $\quad\quad$ **if** $t.\tau = \gamma.l + D_W$ **then** // γ is responsible for t
11 $\quad\quad\quad$ sortInsert($t.\phi$) // store t (sorted on f_T)
12 $\quad\quad$ **else** // this γ does not contain the earliest t
13 $\quad\quad\quad \gamma.\phi.first\gamma \leftarrow \texttt{False}$
14 $\quad$ **else** // t comes from A (via the loop)
15 $\quad\quad \gamma.\phi.m \leftarrow t.\phi.m$
16 $\quad\quad \gamma.\phi.m\tau \leftarrow t.\tau$
17 **Function** $f_O(\gamma)$
18 $\quad$ **if** $\gamma.\phi.pending \wedge (\gamma.\phi.first\gamma \vee \gamma.\phi.m\tau = \gamma.l + D_W - \delta)$ **then** // Ready to produce output payloads
19 $\quad\quad \Phi_O \leftarrow \emptyset$
20 $\quad\quad$ **for** $\forall t \in \gamma.\phi.T$ **do** // For all input ts
21 $\quad\quad\quad \gamma.\phi.m, \Phi'_O \leftarrow f_{M^*}(\gamma.\phi.m, t)$
22 $\quad\quad\quad \Phi_O \leftarrow \Phi_O \cup \Phi'_O$
23 $\quad\quad \gamma.\phi.pending \leftarrow \texttt{False}$
24 $\quad\quad \phi' \leftarrow \{\phi{:}\Phi_O, k{:}\gamma.k, m{:}\gamma.\phi.m, fromS{:}\texttt{False}\}$
25 $\quad\quad$ **return** ϕ'
26 $\quad$ **else**
27 $\quad\quad$ **return** $\emptyset$
Auxiliary functions:
28 sortInsert(t) // Add t to $\gamma.\phi.T$ sorted on f_T
$S_{M_2} \leftarrow M(S_A, f_M)$, where: // M_2-Fig. 4
29 **Function** $f_M(t)$
30 $\quad$ **return** $t.\phi.\phi$

Explanation and Proof. The operator M_1 is covered in (List.5,L1-3). Upon reception of a tuple t, M_1 encapsulates t's payload into a new payload that also carries an attribute m – initially `null` – for the state that will be later updated based on f_{M^*}, t's key based on the user-defined f_K function, and a flag stating this payload carries the payload of a tuple from S.

The A operator running the f_{M^*} function is covered in List.5,L4-28. Note that several implementations can be defined to enforce *R-CER*'s semantics. For ease of exposition, the one we present relies on f_U to store relevant tuples (tuples coming from S) as well as state m fed via A's loop, while it relies on f_O to feed the relevant tuples to f_{M^*}. Based on the previous discussions, A defines a sliding Γ with $WA = \delta$ and $WS = D_W + \delta$. It consumes the tuples from S_{M_1} and those from S_A, relying on $f_{K'}$ to partition them based on the user-defined f_K, whose value is carried in $t.\phi.k$ (List.5,L4-5).

For f_U (List.5,L6-16), we note the state $\gamma.\phi$ is initialized upon the reception of the very first tuple when $\gamma.\phi$ = `null` (List.5,L7-8). In such initialization, $\gamma.\phi$ is assumed to be the γ containing the very first tuple in S, to store an initially empty set of tuples T, to have an uninitiated state m associated with an uninitialized timestamp $m\tau$, and to indicate the results from this γ have not yet been produced by setting the attribute *pending* to `True`.

Once the state is initialized, the incoming tuple t is stored if it comes from S and its timestamp $t.\tau$ is equal to $\gamma.l + D_W$ (List.5,L9-11). As explained before, this ensures that, if an output tuple t_o is produced from t, then $t_o.\tau = \gamma.l + D_W = t.\tau$. Note that tuples are inserted in $\gamma.\phi.T$ in sorted order based on f_T through the auxiliary function `sortInsert` (List.5,L28). Alternatively, $\gamma.\phi.first\gamma$ is set to `False` if t comes from S but $t.\tau \neq \gamma.l + D_W$ (List.5,L12-13). If t comes from A, its state and its timestamp are stored in $\phi.m$ and $\phi.m\tau$, respectively (List.5,L14-16).

Moving now to f_O (List.5,L17-27), we note the set of output payloads Φ_O is populated only if a result has not been produced for γ yet (i.e., if $\gamma.\phi.pending$ = `True`), and if γ is the γ containing the very first tuple delivered by S (and potentially others sharing the same timestamp) or if the state from the previous γ', i.e., the γ' creating an output t_o so that $t_o.\tau = \gamma.l + D_W - \delta$, has been received (List.5,L18). In such a case, f_O traverses all the tuples in $\gamma.\phi.T$ and passes them to f_{M^*} while maintaining an up-to-date state $\gamma.\phi.m$. Eventually, $\gamma.\phi.pending$ is set to `False` to mark an output has been produced for this γ and the payload for an output tuple carrying Φ_O, the associated key k, the state m, and a flag stating such tuple is not from S is returned (List.5,L20-25). If the condition on List.5,L18 is not met, an empty set is returned instead (List.5,L26-27).

Finally, the operator M_2 is covered in (List.5,L29-30). As shown, M_2 forwards only the payloads produced by f_{M^*}.

Note the proposed solution implies for A that each tuple contributes to $\lceil \frac{D_W+\delta}{\delta} \rceil$ γs and, thus, that A needs to maintain such $\lceil \frac{D_W+\delta}{\delta} \rceil$ γs in parallel for each f_K value. While such a value is application- and SPE-dependent, because of D_W and δ, respectively, we note nonetheless D_W can be controlled by a data source/SPE itself and is usually in the sub-second range (e.g., 200 ms as default for Apache Flink [13]).

Before proving Theorem 3, we introduce and prove the following supporting theorem.

Theorem 4. *Being: t_0 the tuple with the earliest timestamp τ fed by S for a given key k (or one of such tuples, if multiple tuples sharing the earliest timestamp exist), γ_0 the γ storing t_0 in $\gamma.\phi.T$ (List.5,L11), and γ_{i-1}, γ_{i+1} the γs preceding and following γ_i, respectively, then the A operator in Listing 5 produces its first output tuple from γ_0, its second output tuple from γ_1, and, in general, its i-th output from γ_{i-1}.*

Proof. (Theorem 4) First, we can note that γ_0 is the earliest γ maintained by A, because γ_{-1} covers $[\gamma_0.l-\delta, \gamma_0.l+D_W)$ and t_0 does not fall in it, since $t_0.\tau = \gamma_0.l + D_W$, nor in earlier γs. Hence, upon invocation of f_O on γ_0, $\gamma_0.\phi.first\gamma$ is `True`, and the output tuple t_o so that $t_o.\tau = t_0.\tau$ will be the first output tuple produced by A. Any subsequent invocation of f_O on γ_0 will not result in new output tuples since $\gamma_0.\phi.pending$ = `False` (List.5,L23).

Since $t_0.\tau = \gamma_0.l + D_W$, then $\gamma_0.l = t_0.\tau - D_W$, $\gamma_1.l = t_0.\tau - D_W + \delta$, $\gamma_2.l = t_0.\tau - D_W + 2\delta$, and so on. No matter how small D_W is, even when $D_W = \delta$, we can observe that when fed to A, t_0 will also contribute to γ_1, which for $D_W = \delta$ covers the period $[t_0.\tau, t_0.\tau + 2\delta)$, setting $\gamma_1.\phi.first\gamma$ to `False`. Upon the invocation of f_O on γ_1 (note γs for which an output can be produced upon a watermark update are traversed in order, see § 2.2), the condition in List.5,L18 can thus be met only after t_o is produced and added to γ_1, since at that point $\gamma_1.\phi.m\tau$ will be $t_0.\tau$ and $\gamma_1.l+D_W-\delta = t_0.\tau-D_W+\delta+D_W-\delta = t_0.\tau$. Also in this case, any subsequent invocation of f_O on γ_1 will not result in new output tuples since $\gamma_1.\phi.pending$ = `False` (List.5,L23).

Moving now to γ_2, we can observe it is not necessarily true that t_0 falls in γ_2 too. More concretely, if $D_W = \delta$, γ_2 covers the period $[t_0.\tau+\delta, t_0.\tau+3\delta)$. Nonetheless, the output t_o from γ_1 does, since it carries the timestamp $t_0.\tau + \delta$ (**A2**), and is needed upon invocation of f_O on γ_2 to match the condition on List.5,L18 and result in an output tuple (exactly once).

A similar reasoning applies to all γ_2's subsequent γs. □

We can now use Theorem 4 to support the proof of Theorem 3. The proof for Theorem 3 resembles that of Theorem 1. In this case, though, accounting also for tuples fed to A through its loop.

Proof. (Theorem 3) By contradiction, if the operators in Figure 4 and Listing 5 do not enforce the semantics of *R-CER* then, being t_i an input tuple associated to key $k = f_K(t)$ and t_o an output tuple resulting from the invocation of f_{M^*} on t_i, then one of three cases holds: t_o is not produced, t_o is produced twice or more, or t_o is produced but with wrong τ and/or ϕ. Note that the non-production of t_o excludes the production of multiple t_os and the production of any t_o with wrong τ/ϕ, while the latter two cases could be observed simultaneously.

We can generalize these cases into three main cases. The non-production of t_o or the production of multiple t_o can be observed if t_i is not processed exactly once (case **C1**) or if t_i is fed to f_{M^*} with the wrong state m (case **C2**), where m is wrong if it was not correctly updated by processing all the tuples before t_i (based on $t_i.\tau$ and f_T) and that share the same key as t_i. Case **C2** could also lead to the production of a t_o with a wrong ϕ. Finally, a t_o carrying the wrong τ could also be caused by the processing of t_i in a γ for which $t_i \in \gamma$ does not lead to a t_o so that $t_i.\tau = t_o.\tau$ (case **C3**).

To prove **C1** cannot be observed, we note M_1 and M_2 forward an output tuple for each input tuple. Thus, **C1** can only hold if t_i is not stored exactly once in the $\gamma.\phi.T$ of a γ or, if once stored, it is not processed exactly once. Since A has $WA = \delta$, no matter the f_K value of t_i, A will define a γ for each possible left boundary $0, \delta, 2\delta, 3\delta, \ldots$ including the only one fulfilling the condition $t_i.\tau = \gamma.l + D_W$ in List.5,L10 (i.e., that of the γ in which t_i is stored). Moreover, t_i is fed exactly once to f_{M^*} in List.5,L21 only if $\gamma.\phi.pending$ is `True` (List.5,L18) since once t_i is fed to f_O, $\gamma.\phi.pending$ is set to `False` (List.5,L23). Hence, **C1** cannot be observed.

Note that proving **C1** cannot be observed implies **C3** is also not observable because, if t_i is added to the state of the γ so that $t_i.\tau = \gamma.l+D_W$, then an output t_o from that γ will have timestamp $t_o.\tau = \gamma.l + WS - \delta = t_i.\tau - D_W + D_W + \delta - \delta = t_i.\tau$.

Finally, **C2** can be also proven not to hold based on Theorem 4, which implies all tuples in $\gamma_i.\phi.T$ are processed (in f_T order) and used to update the state $\gamma_i.\phi.m$ before such state is passed to γ_{i+1} and used by γ_{i+1} to process its $\gamma_{i+1}.\phi.T$ tuples in order and further pass the updated state to γ_{i+2}, and so on. □

6 RELATED WORK

Our study contributes to the fields of CER and stream processing, which have witnessed substantial advancements and increased use in industrial setups over the past decade.

In the domain of CER, several formalisms have been proposed to express patterns of interest: they range from automata-based [8, 20] or tree-based [24] languages to logic-based abstractions [1, 6, 10, 11]. These formalisms offer expressive constructs to capture the needs of applications, including filtering of individual events, compositions into sequences with temporal constraints, iterations, and selection and consumption policies to precisely indicate when a pattern is satisfied and which events it should consider. Our work is orthogonal to these aspects, and shows that *A* operators are sufficient to capture patterns with known and unknown duration. The general solutions we presented in the paper can be declined to capture any of the specific formalisms presented in the literature.

A pivotal area of research within this domain has been the exploration of the expressiveness of CER languages. Studies such as those by Artikis et al. [4] and Grez et al. [16] have been instrumental in proposing abstract operators to capture the expressive capabilities of CER constructs. These contributions align with our work, albeit from a different angle. More concretely, while [4, 16] aim to establish a formal framework for CER tasks, our focus is on demonstrating the feasibility of implementing these tasks relying on the stream processing Dataflow model [3], and specifically leveraging *A* operators.

To the best of our knowledge, our work is the first to analyze, from a formal standpoint, the semantic overlap between stream processing and CER. We establish that the semantics of common CER operators can be effectively realized through compositions of common *A* operators, thereby enriching both the theoretical understanding and practical application of these concepts in streaming environments.

Among the existing literature, [18] is the study with the closest resemblance to our research, particularly in its assertion that *A*s are sufficient for enforcing the semantics of other operators. While primarily addressing common streaming operators (Filter, Map, and Join), [18] also proposes an early version of the *R-CER* operator, which is nonetheless a very preliminary approach that mostly aims at showing streaming analysis can benefit from loops to have γs' state shared across window instances. In contrast, our research offers a more refined definition of this operator and delves into comprehensive discussions on its implementation and correctness verification.

Palyvos [25] presents an alternative version of the *T-CER* operator, which is also examined in our study within the context of time-based pattern matching for events that span a period of event time of known length but unknown location. Our approach, however, extends beyond this initial exploration by proposing a generalized implementation framework. More concretely, the solution proposed in [25] is less efficient and relies on a larger set of prerequisites for the underlying streaming model. About the efficiency: it relies on two *A*s each with a γ with *WA* and *WS* set to $2D_C$, while we show a single *A* with a $WS \geq D_C$ suffices (see § 4). About the underlying model: it assumes *A* operators also define a Window Offset *WO*, thus covering the event time periods $[\ell WA + WO, \ell WA + WO + WS)$, with $\ell \in \mathbb{N}$, while we do not require such extra parameter for Γ (see § 2.1).

Finally, [17] demonstrates the potential of base operators for constructing queries capable of learning and maintaining a Bayesian Network in the context of intrusion detection systems. While [17] highlights the rich semantics achievable with base operators, it is specifically tailored to the semantics of its application domain, distinguishing it from our broader examination of CER and stream processing techniques. Also, [17] aims at showing the semantics of Bayesian Networks can be enforced using common stream processing operators, but does not study which is the minimum set of common operators required to enforce such semantics.

7 CONCLUSIONS AND FUTURE WORK

This work proposed a formal discussion about the semantic overlap that exists between Complex Event Reasoning (CER) and the operators of distributed stream processing engines (SPEs). More concretely, we formally argued how a single Aggregate operator (*A*) suffices to enforce the semantics of CER analysis that can (1) analyze individual events or groups of events that take place at the same point in time, (2) find patterns of events that span a period of event time of known length, and (3) find such patterns even when the period of event time they cover is not known in advance (and potentially unbounded).

Besides the theoretical contribution we make, these findings have also an important practical implication when it comes to existing state-of-the-art frameworks for CER analysis and how their computational costs (e.g., CPU, memory) compare with the resources made available for such an analysis. More concretely, users who wish to run CER analysis as the one in focus in our paper can do so provided they have access to an SPE that can run (compositions of) the minimalistic *A* we rely on.

The solutions proposed in this paper are purely meant to support our claim about the semantic equivalence with their CER counterpart. As pointed out in § 4 and § 5, multiple solutions can be defined while enforcing the same semantics. A possible extension of this work can thus focus on such alternatives to account for performance- and implementation-specific optimizations. In parallel with that, we plan to extend this work with an empirical performance assessment comparing state-of-the-art frameworks with a dedicated SPE that only supports the *A* we rely on.

ACKNOWLEDGMENTS

This work is supported by the Swedish Government Agency for Innovation Systems VINNOVA, proj. “AutoSPADA” under Grant DNR 2019-05884 in the funding program FFI: Strategic Vehicle Research and Innovation, by the Marie Skłodowska-Curie Doctoral Network project RELAX-DN, funded by the

European Union under Horizon Europe 2021-2027 Framework Programme Grant Agreement number 101072456, by the Chalmers AoA frameworks Energy and Production, WPs INDEED, and “Scalability, Big Data and AI", respectively, by the Swedish Energy Agency (SESBC) project TANDEM, by the Wallenberg AI, Autonomous Systems and Software Program and Wallenberg Initiative Materials for Sustainability project STRATIFIER, and by the PNRR MUR project PE0000021-E63C22002160007-NEST.

REFERENCES

[1] Asaf Adi and Opher Etzion. 2004. Amit - the situation manager. *The VLDB Journal* 13, 2 (may 2004), 177–203. https://doi.org/10.1007/s00778-003-0108-y

[2] Tyler Akidau, Edmon Begoli, Slava Chernyak, Fabian Hueske, Kathryn Knight, Kenneth Knowles, Daniel Mills, and Dan Sotolongo. 2020. Watermarks in Stream Processing Systems: Semantics and Comparative Analysis of Apache Flink and Google Cloud Dataflow. *Proceedings of the VLDB Endowment* 3 (2020).

[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 Wittle. 2015. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proc. Endowment* 8, 12 (2015), 1792–1803.

[4] Alexander Artikis, Thomas Eiter, Alessandro Margara, and Stijn Vansummeren. 2021. Dagstuhl Seminar on the Foundations of Composite Event Recognition. *SIGMOD Rec.* 49, 4 (2021), 24–27. https://doi.org/10.1145/3456859.3456865

[5] Alexander Artikis, Alessandro Margara, Martin Ugarte, Stijn Vansummeren, and Matthias Weidlich. 2017. Complex Event Recognition Languages: Tutorial. In *Proceedings of the 11th ACM International Conference on Distributed and Event-Based Systems* (Barcelona, Spain) *(DEBS '17)*. ACM, New York, NY, USA, 7–10.

[6] Alexander Artikis, Marek Sergot, and Georgios Paliouras. 2015. An Event Calculus for Event Recognition. *IEEE Trans. on Knowl. and Data Eng.* 27, 4 (apr 2015), 895–908. https://doi.org/10.1109/TKDE.2014.2356476

[7] beam [n.d.]. Apache Beam. https://beam.apache.org/ Accessed:2024-02-28.

[8] Lars Brenna, Alan Demers, Johannes Gehrke, Mingsheng Hong, Joel Ossher, Biswanath Panda, Mirek Riedewald, Mohit Thatte, and Walker White. 2007. Cayuga: a high-performance event processing engine. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data* (Beijing, China) *(SIGMOD '07)*. ACM, New York, NY, USA, 1100–1102. https://doi.org/10.1145/1247480.1247620

[9] Paris Carbone, Asterios Katsifodimos, 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, 4 (2015), 28–38.

[10] Gianpaolo Cugola and Alessandro Margara. 2010. TESLA: a formally defined event specification language. In *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems* (Cambridge, United Kingdom) *(DEBS '10)*. ACM, New York, NY, USA, 50–61. https://doi.org/10.1145/1827418.1827427

[11] Gianpaolo Cugola and Alessandro Margara. 2012. Complex event processing with T-REX. *J. Syst. Softw.* 85, 8 (aug 2012), 1709–1728. https://doi.org/10.1016/j.jss.2012.03.056

[12] Gianpaolo Cugola and Alessandro Margara. 2012. Processing Flows of Information: From Data Stream to Complex Event Processing. *ACM Comput. Surv.* 44, 3, Article 15 (2012), 62 pages. https://doi.org/10.1145/2187671.2187677

[13] flink [n.d.]. Apache Flink. https://flink.apache.org Accessed:2024-02-28.

[14] Marios Fragkoulis, Paris Carbone, Vasiliki Kalavri, and Asterios Katsifodimos. 2023. A survey on the evolution of stream processing systems. *The VLDB Journal* (2023), 1–35.

[15] Nikos Giatrakos, Elias Alevizos, Alexander Artikis, Antonios Deligiannakis, and Minos Garofalakis. 2019. Complex Event Recognition in the Big Data Era: A Survey. *The VLDB Journal* 29, 1 (jul 2019), 313–352. https://doi.org/10.1007/s00778-019-00557-w

[16] Alejandro Grez, Cristian Riveros, Martín Ugarte, and Stijn Vansummeren. 2021. A Formal Framework for Complex Event Recognition. *ACM Trans. Database Syst.* 46, 4, Article 16 (dec 2021), 49 pages. https://doi.org/10.1145/3485463

[17] Vincenzo Gulisano, Magnus Almgren, and Marina Papatriantafilou. 2014. Metis: a two-tier intrusion detection system for advanced metering infrastructures. In *Proceedings of the 5th international conference on Future energy systems*. 211–212.

[18] Vincenzo Gulisano, Alessandro Margara, and Marina Papatriantafilou. 2024. On the Semantic Overlap of Operators in Stream Processing Engines. https://doi.org/10.1145/3652892.3654790

[19] Vincenzo Gulisano, Dimitris Palyvos-Giannas, Bastian Havers, and Marina Papatriantafilou. 2020. The role of event-time order in data streaming analysis. In *Proceedings of the 14th ACM International Conference on Distributed and Event-based Systems*. 214–217.

[20] Daniel Gyllstrom, Jagrati Agrawal, Yanlei Diao, and Neil Immerman. 2008. On Supporting Kleene Closure over Event Streams. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering (ICDE '08)*. IEEE Computer Society, USA, 1391–1393. https://doi.org/10.1109/ICDE.2008.4497566

[21] Liebre SPE. 2022. https://github.com/vincenzo-gulisano/Liebre. Accessed:2024-02-28.

[22] Alessandro Margara, Gianpaolo Cugola, Nicolò Felicioni, and Stefano Cilloni. 2023. A Model and Survey of Distributed Data-Intensive Systems. *ACM Comput. Surv.* 56, 1, Article 16 (2023), 69 pages.

[23] Ruben Mayer, Ahmad Slo, Muhammad Adnan Tariq, Kurt Rothermel, Manuel Gräber, and Umakishore Ramachandran. 2017. SPECTRE: Supporting consumption policies in window-based parallel complex event processing. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*. 161–173.

[24] Yuan Mei and Samuel Madden. 2009. ZStream: a cost-based query processor for adaptively detecting composite events. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data* (Providence, Rhode Island, USA) *(SIGMOD '09)*. ACM, New York, NY, USA, 193–206. https://doi.org/10.1145/1559845.1559867

[25] Dimitris Palyvos-Giannas, Bastian Havers, Marina Papatriantafilou, and Vincenzo Gulisano. 2020. Ananke: a streaming framework for live forward provenance. *Proceedings of the VLDB Endowment* 14, 3 (2020), 391–403.

[26] storm [n.d.]. Apache Storm. http://storm.apache.org Accessed:2024-02-28.

[27] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM* 59, 11 (2016), 56–65.