D^{2}IA: Stream Analytics on User-Defined Event Intervals

Ahmed Awad[1](✉), Riccardo Tommasini[2], Mahmoud Kamel[1], Emanuele Della Valle[2], and Sherif Sakr[1]

[1] University of Tartu, Tartu, Estonia
{ahmed.awad,mahmoud.shoush,sherif.sakr}@ut.ee
[2] Politecnico di Milano, Milan, Italy
{riccardo.tommasini,emanuele.dellavalle}@polimi.it

Abstract. Nowadays, modern Big Stream Processing Solutions (e.g. `Spark`, `Flink`) are working towards ultimate frameworks for streaming analytics. In order to achieve this goal, they started to offer extensions of SQL that incorporate stream-oriented primitives such as windowing and Complex Event Processing (CEP). The former enables stateful computation on infinite sequences of data items while the latter focuses on the detection of events pattern. In most of the cases, data items and events are considered instantaneous, i.e., they are single time points in a discrete temporal domain. Nevertheless, a point-based time semantics does not satisfy the requirements of a number of use-cases. For instance, it is not possible to detect the interval during which the temperature increases until the temperature begins to decrease, nor all the relations this interval subsumes. To tackle this challenge, we present D^{2}IA; a set of novel abstract operators to define analytics on user-defined event intervals based on raw events and to efficiently reason about temporal relationships between intervals and/or point events. We realize the implementation of the concepts of D^{2}IA on top of `Esper`, a centralized stream processing system, and `Flink`, a distributed stream processing engine for big data.

Keywords: Big Stream Processing · Complex event processing · User-defined event intervals

1 Introduction

Streaming data analytics has become a key enabler for organizations' success and sustainability. Data velocity is often too high, and we are forced to process data on-the-fly. To solve this challenge, Stream Processing Engines (SPEs) have been proposed. SPEs are commonly classified into two main categories: Data Stream Management Systems (DSMSs) and complex event processing (CEP) [8].

As streams are infinite sequences of partially ordered data (events), both DSMSs and CEP solutions offer special operators to deal with unboundedness. In

P. Giorgini and B. Weber (Eds.): CAiSE 2019, LNCS 11483, pp. 346–361, 2019.
https://doi.org/10.1007/978-3-030-21290-2_22

particular, DSMSs apply the concepts of temporal windowing that slice streams into finite portions [9] and then applies stateful aggregations, e.g., the average temperature over the last 5 min (Listing 1.1 - Line 1). On the other hand, CEP employs non-deterministic finite-state machines and rule-based languages to define and detect event-patterns on streams [10], e.g., emit fire if a smoke detection is followed by a temperature higher than 40 (Listing 1.1 - Lines 3, 4).

Listing 1.1. DSMS and CEP query in EPL

```
1 select avg(val) from Temperature#time(5 min) output every 5 min;
2
3 insert into Fire
4 select * from pattern [Smoke ->Temperature(val >40)] where timer:within(5 min);
```

In practice, the state-of-the-art of SPEs is vast and includes a variety of DSMSs and CEP languages, as well as hybrid approaches. For instance, EPL is an industrial stream processing language that combines DSMS and CEP features. `Esper`[1] and `OracleCEP`[2] are examples of centralized solutions that implement it. Recently, `BigSPEs` represent a new generation of distributed, scalable and fault-tolerant SPE systems (e.g. `Spark`, `Flink`) that have been designed to address the volume and velocity challenges of the Big Data wave. Nevertheless, BigSPEs' focus on scalability goes at the expense of providing less expressiveness. In particular, existing systems provide little expressive Domain Specific Languages (DSL) that do not meet the expectations raised by the centralized solutions [13]. In practice, such expressiveness is crucial in several applications, e.g., healthcare. For example, let us consider the following air traffic scenario where many events are continuously produced during flights, e.g., changes in altitude, speed, and heading of an aircraft. In such scenario, we can be interested in detecting those intervals during which a plane is in cruising mode and performs a change in altitude which is more than 10%.

Listing 1.2. Example encoded in EPL

```
create schema AltitudeChange as (starts long, endts long,
        init_alt long, fin_alt long);
create schema CruisePeriod as (onts long, offts long)
        starttimestamp onts endtimestamp offts;
insert into AltitudeChange
select minby(ts).value as init_alt, maxby(ts).value as fin_al,
   maxby(ts).ts as endts, minby(ts).ts as starts
from Altitude#time(30 minutes) output every 30 minutes;
insert into CruisePeriod select onts, offts from CruiseMode
match_recognize ( measures a.ts as onts, b.ts as offts
pattern (A B)* defines A as A.value='On', B as B.value='Off');
select ac.* from AltitudeChange as ac, CruisePeriod cp
where ac.during(cp) and abs(ac.fin_alt - ac.init_alt) / ac.init_alt >= 0.1);
```

If we implement such scenario using EPL (Listing 1.2), we observe that we cannot express it in a single query, however, we need to create a query network. Moreover, the network complexity, i.e., the number of required queries, increases when more conditions are added to the original scenario. On the other hand, if we consider BigSPEs, we observe that they lack the possibility to generate

[1] http://www.espertech.com/.
[2] https://docs.oracle.com/cd/E17904_01/doc.1111/e14476/.

and process streams of events with a duration, i.e., interval events. Figure 1 summarizes the spectrum of the streaming language operators extending the CQL model proposed in [5] with three new families of operators, i.e., Event-to-Interval (E2I), Interval-to-Interval (I2I), and Interval-to-Event (I2E) operators.

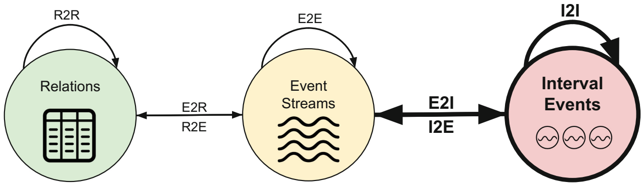


Fig. 1. Stream-Event-IntervalEvent models and operators

Among the state-of-the-art stream processing engines, only centralized solutions provide query languages that are expressive enough to process stream of events with durations [4,17]. However, these queries have to be handcrafted by the application developer connecting several queries, as shown in Listing 1.2. In order to fill this gap, in this paper, we introduce `D²IA` (`Data-driven Interval Analytics`), a *novel* family of operators that enables interval events generation and reasoning. In particular, `D²IA` allows generating data-driven event intervals from instantaneous events (**E2I**) by combining event patterns with an extensive collection of aggregation functions. It selects intervals with maximum durations to reduce the complexity of the processing (**I2I**). In addition, it enables efficient reasoning on intervals using Allen's Algebra [3] (**I2E**).

The operators of `D²IA` have been designed according to the principles of Codd's language [7,17]:: **Minimality**, i.e., a language should provide only the necessary constructs to avoid multiple ways to express the same meaning. **Symmetry**, i.e., a language construct always expresses the same semantics regardless of the usage context. **Orthogonality**, i.e., meaningful combinations of language constructs should be applicable. We demonstrate the expressiveness and practicality of `D²IA` in two ways: (i) we provide an algorithm that translates `D²IA` operators to an expressive streaming query language, `EPL`, that is employed only by centralized solutions. (ii) We implement `D²IA` operators on top of Apache Flink[3], a popular distributed stream processing engine.

The remainder of the paper is organized as follows. Necessary background is introduced in Sect. 2. Concepts behind `D²IA` are presented in Sect. 3. Section 4 describes the implementation details. Related work is discussed in Sect. 5 before we finally conclude the paper in Sect. 6.

2 Background

2.1 Complex Event Processing

Complex event processing aims at the identification of patterns that represent a complex event over input streams of low-level events [10]. Patterns are analogous to defining regular expressions over strings; they are defined w.r.t. event types

[3] https://flink.apache.org/.

and matched against event instances in the streams. Traditionally, events are seen as instantaneous and the timestamp of the last matched event is assigned to the emitted complex event. In the following, we recap these notions.

Definition 1 (Raw Event). *A raw event is an instantaneous and atomic notification of an occurrence of interest at a point in time.*

A raw event has a payload of interest to the application, a timestamp that indicates the point in time when it took place. Moreover, event instances are usually organized in classes or topics. We can represent an event as a triple $<id, payload, ts>$ where *id* is an identifier of the event source, *payload* is simply a list of key-value pairs, and *ts* is the timestamp at which the event was generated. The event type is a determined by id and payload. A complex event is derived from a collection of events using pattern matching, e.g., *FireEvent* = *SmokeEvent followed-by HighTemperatureEvent*.

Definition 2 (Complex Event). *A complex event is a composition of one or more events. The composition is obtained as a result of matching a pattern to streams of composing events. A complex event has a payload and a timestamp which depends on the semantics of the CEP language.*

2.2 Data-Driven Windows

Recently, several types of data-driven windows have been discussed in literature. Traditional window operators [5,9] lack the necessary expressiveness to capture relevant situations. For instance, consider the case of a smart home application. It might be interesting to report only those windows in which temperature goes above a certain threshold. In such case, it is not known ahead when the temperature will rise, i.e. the window width can not be provided. To tackle such use-cases, more expressive window operators have been proposed. Session windows [1] are one example of time-based windows that allows slicing the stream based on user-behavior, e.g., a click stream session. Grossenklaus et al. [12] proposed four types of data-driven windows called frames: (i) **Threshold Frames** report time intervals within a stream where an attribute of a stream element goes higher (lower) than a given threshold. (ii) **Delta Frames** report time intervals within which an attribute of a stream element changes by more than amount x. That is, we can find two elements within the interval such that the difference between their attribute values is higher (lower) than x. (iii) **Aggregate Frames** report time intervals within which an aggregate of an attribute of stream elements remains below a certain threshold. (iv) **Boundary Frames** are time intervals within which an attribute of stream elements remains within one of the predefined boundaries.

3 Operators for User-Defined Intervals Analytics

In this section, we start by defining interval events. Then, we present our `D`^{2}`IA` operators for deriving and reasoning about user-defined interval events.

As stated in Definition 2, a complex event has a payload and a timestamp similar to a raw event (Definition 1). However, while it is understandable that the definition of the derived payload is left to the application developer in terms of payloads of the composing events, the reasoning about the timestamp should be more rigorous. The *payload* is a list of key-value pairs. We define two auxiliary functions: (i) *keys* :: payload -> [keys], which returns the keys present in the payload and; (ii) $val :: payload, key- > value$ to retrieve the value associated to a given key in a payload.

The literature on CEP contains some examples that acknowledge the limitations of an instantaneous temporal model for events. Interval-based models have a richer semantics than traditional point-based ones [4]. Moreover, an interval-based temporal models can represent point events without loss of generality.

Definition 3 (Interval Event). *An interval event is a special event that has a temporal duration which is defined in term of a two time points,* start *and* end.

Interval events, a.k.a. situations, are special kind of events that instead of having a time point-based timestamp, they have a duration, i.e., a temporal interval, within which the event is observed.

Considering Codd's language design principles, introduced in Sect. 1, and inspired by the data frames, discussed in Sect. 2.2, we elicit the following requirements: (R1) **Contextual State Management**, i.e., the target language must allow the definition of contextual variables and partitioning of the state. R1 is required to ensure the feasibility of relative and absolute conditions. (R2) **Analytical Features**, i.e., the operators must enable stateful aggregations, for example employing temporal/physical windows. R2 is required to express data-driven windows. (R3) **Pattern Detection**, i.e., the operators must enable interval generation via event detection. R3 is required since complex events can be seen as interval events, considering the timestamp of the initial match and the timestamp of the last match in the provided pattern [4].

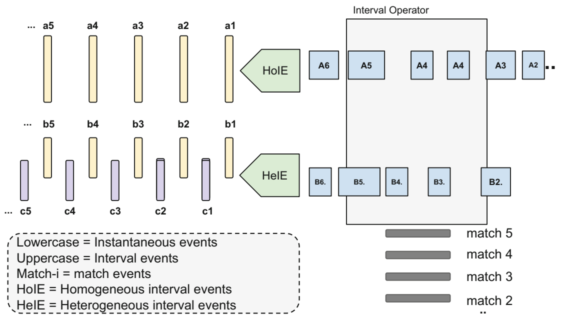


Fig. 2. D^{2}IA overview: Homogeneous/Heterogeneous Interval Event generators consume events streams and produce interval events.

In the following sections, we provide details about interval generators and operators. Figure 2 exemplifies a pipeline in which two event streams are transformed into interval events using *interval generators* and fed into an *interval operator* that reasons about the interval events.

3.1 Homogeneous Interval Events Generators

Interval event generators represent a family of `D`^{2}`IA` operators which are responsible for creating interval events out of a stream of instantaneous events. In particular, the interval generator transforms the input stream(s) into the output stream based on a pattern specification. If a single input stream is used, the resulting event intervals are *homogeneous*. Otherwise, they are *heterogeneous*.

Definition 4 (Homogeneous Interval Event Generator). *Let E be a raw event from Definition 1. A homogeneous interval event HoIE specification is defined by a tuple* $< E, Occurrence, Value, KeyBy, Condition, Within >$ *where:*

- *Event: refers to the type of the event on which the interval to be defined.*
- *Occurrence: is on the form* (min, max) *to indicate the minimum and maximum number of event instances to match. Also, wild card* $*$ *can be used to make no upper-bound on the number of occurrences.*
- $Value$*: refers to either a constant value, an expression, or an aggregation over the event payload's attribute value. Possible aggregation functions are: min, max, avg, etc. aggregates are computed over the matched raw events.*
- $KeyBy$*: specifies an attribute in the event's payload to group event instances.*
- $Condition$*: defines a filter condition over the event instances. Conditions are expressed w.r.t. event's payload attributes and can be either absolute or relative. The former compares the event instance's attribute value with a constant value; the latter compares the event instance's attribute value with an expression over other event instances' attribute values. Relative condition are expressed with the form* $start(EventInstance.value\ \theta\ v) \wedge subsequent(EventInstance.value\ \theta\ Value)$.
- $Within$*: specifies a maximum time interval to wait for the match to validate since the first event arrives. An example is* $5\,s$*,*

A homogeneous event interval is generated when one or more (raw) events of the same type are observed in succession. Using analogy with regular expressions, a homogeneous event interval is on the form `A{min,max}`, where `A` is the event type. Optionally, the interval definition can be restricted by a temporal window, a condition, or a combination thereof. The temporal window restricts the maximum temporal gap among the occurrences of event instances. The condition puts a restriction on the value of an event instance property to consider it as a match. In `D`^{2}`IA`, it is possible to define relative and absolute conditions (Definition 4). The interval event value can be obtained by applying an aggregate on the matching raw events. We use `HoIE` as an operator to build intervals specification using fluent APIs pattern[4].

Example. Assume a temperature event on the form *Temperature < sensor, temp,ts >* which refers to the *sensor* ID that generated the event, the temperature *temp* reading and the timestamp *ts* for the reading. We can define an event interval with absolute condition as

[4] https://martinfowler.com/bliki/FluentInterface.html.

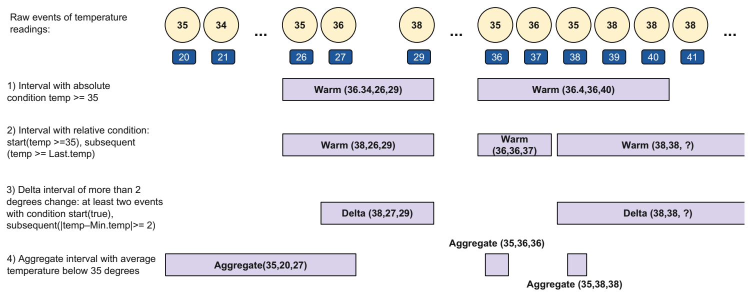

Raw events of temperature readings:
35
34
...
35
36
38
...
35
36
35
38
38
38
...
20
21
26
27
29
36
37
38
39
40
41
1) Interval with absolute condition temp >= 35
Warm (36.34,26,29)
Warm (36.4,36,40)
2) Interval with relative condition: start(temp >=35), subsequent (temp >= Last.temp)
Warm (38,26,29)
Warm (36,36,37)
Warm (38,38, ?)
3) Delta interval of more than 2 degrees change: at least two events with condition start(true), subsequent(|temp–Min.temp|>= 2)
Delta (38,27,29)
Delta (38,38, ?)
Aggregate (35,36,36)
4) Aggregate interval with average temperature below 35 degrees
Aggregate(35,20,27)
Aggregate (35,38,38)

Fig. 3. Homogeneous interval events for the different data-driven frames

Listing 3.1. Warm interval with absolute condition

```
WarmAbsolute=HoIE.Event(Temperature).Value(Aggregate.avg(
Temperature.temp)).Occurrence(2,5).KeyBy(Temperature.sensor)
.Within(5,seconds).AbsoluteCondition(Conditions.greaterOrEqual(
Temperature.temp,35))
```

We can also define an event interval with a relative condition on the form.

Listing 3.2. Warm interval with relative condition

```
WarmRelative=HoIE.Event(Temperature).Occurrence(2,5)
.Value(Aggregate.max(Temperature.temp))
.KeyBy(Temperature.sensor).Within(5,seconds).RelativeCondition
.Start(Conditions.greaterOrEqual(Temperature.temp,35))
.Subsequent(Conditions.greaterOrEqual(Temperature.temp,
Last.key(Temperature.temp)))
```

In Listing 3.1, the interval generator is instructed to generate an interval event of type `WarmAbsolute`. An instance of that interval event is generated upon observing 2 to 5 instances of the `Temperature` event. These instances have to be observed within 5 s from each other and each temperature event instance must have its *temp* value grater than 35. The generated interval instance will have its value as the `average` of the temperature readings of the matching `Temperature` event instances. In Listing 3.2, an event interval of type `WarmRelative` is defined on the same stream of `Temperature` events with the same time window. However, the relative condition indicates that the first matching `Temperature` event must have its reading greater than 35. Each succeeding matching event must have its reading greater than or equal to the previously matching event in the pattern. The value of the generated event interval will be the `maximum` temperature value from the matched raw events. In both cases, `keyBy` is used to group the raw events, temperature events in these cases, by their sensor id. Figure 3 shows a stream of temperature events on the top, for the same sensor, and the different matches and event intervals generated for the two cases on rows 1 and 2, respectively.

The ability to define relative conditions on stream elements contributing to a homogeneous event interval allows $\mathtt{D^2IA}$ to cover the four data frames discussed in Sect. 2.2. To define a threshold frame on temperature events of value 35°, an interval in $\mathtt{D^2IA}$ can be defined much like the `WarmAbsolute` interval definition from Listing 3.2. Within this context, a delta frame can be defined on the interval within which the temperature reading increases by more than 2°. This can be defined as shown in Listing 3.3.

Listing 3.3. Delta interval

```
Delta=HoIE.Event(Temperature).Occurrence(2,Occurrences.Unbounded)
.Value(Aggregate.max(Temperature.temp))
.KeyBy(Temperature.sensor).RelativeCondition.Start(true)
.Subsequent(Conditions.greaterOrEqual((Math.absolute(
Math.minus(Temperature.temp,Min.key(Temperature.temp))),2)))
```

In the delta interval definition, we have not used a start condition. Thus, the very first temperature event will start an interval as well as a new temperature event after an interval has been generated. Moreover, we require the minimum number of elements in the interval to be 2. This is to avoid cases where an interval is a singleton. Row 3 in Fig. 3 shows example delta intervals. The first three events were not included because the difference of their values to the minimum value is below 2°. Starting from the event at time 27, the first delta interval begins as the second temperature event satisfies the delta condition. Another interval begins at time 38 and continues.

Listing 3.4. Aggregate interval

```
Aggr=HoIE.Event(Temperature).Occurrence(1,Occurrences.Unbounded)
.Value(Aggregate.avg(Temperature.temp)).KeyBy(Temperature.sensor)
.RelativeCondition.Start(true).Subsequent(Conditions
.lessOrEqual(Aggregate.avg(Temperature.temp),35))
```

An aggregate frame of, e.g., average temperature threshold of 35° can be defined shown in Listing 3.4. In this interval definition, we keep adding events to the interval as long as the average of the added elements including the new one makes the condition hold true. Examples of matches to the aggregate interval definition are shown in row 4 in Fig. 3. The first interval spans the time from 20 to 27 as the average of temperature readings was less than or equal to 35. Other two singleton intervals are defined at times 36 and 38 respectively.

The definition of a boundary frame is similar to the threshold frame. Yet, several intervals have to be defined based on the required boundaries to be monitored on the range of attribute values. For example, ranges on the temperature readings of sensors can be defined as *normal*, readings until 25°, *warm* from 26 to 30°, and *hot* if above 30°. This can easily be represented with intervals with absolute conditions as the threshold interval above.

3.2 Heterogeneous Interval Events Generators

In the case of Heterogeneous event intervals, the types of events signifying the start and the end of the intervals are different. Moreover, instances of other event types might be required *not* to be observed within the interval. As an example of a heterogeneous interval, consider the execution of a business process. The whole duration of the process instance is a heterogeneous interval. Even for the individual work items within a process instance, each work item can be seen as a heterogeneous interval [18]. The interval is delimited by a *start* and *end*

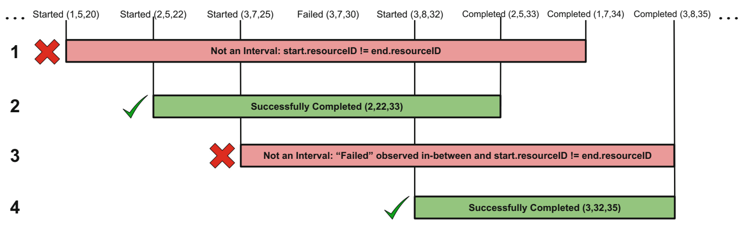

...
Started (1,5,20)
Started (2,5,22)
Started (3,7,25)
Failed (3,7,30)
Started (3,8,32)
Completed (2,5,33)
Completed (1,7,34)
Completed (3,8,35)
...
1
Not an Interval: start.resourceID != end.resourceID
2
Successfully Completed (2,22,33)
3
Not an Interval: "Failed" observed in-between and start.resourceID != end.resourceID
4
Successfully Completed (3,32,35)

Fig. 4. Heterogeneous interval events

events for the process or the work item. However, the two event instances have to belong to the same process instance. Besides event types identifying the start and the end of the interval, it is possible to refer to other event types within D^{2}IA. Instances of such event types *must not* be observed within the interval.

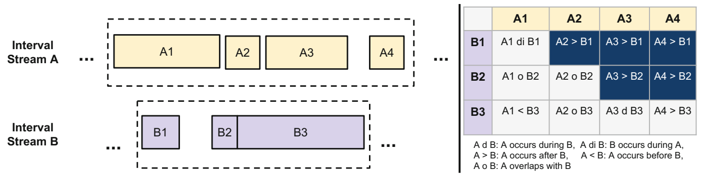
Definition 5 (Heterogeneous Interval Event). *Let E be the universe of event types. A heterogeneous interval event HeIE specification is defined by a tuple $< start, end, Exclude, KeyBy, Condition, Within >$ where:*

- *$start \in E$ refers to the type of the start event for the interval,*
- *$end \in E$ refers to the type of the start event for the interval and $start \neq end$,*
- *$Exclude \subset E$, is the set of event types not to be observed within the interval,*
- *$KeyBy$: refers to the id attribute of the raw events and/or to any $k \in Keys(value)$ of the raw events to group event instances with the same value together,*
- *$Condition$: defines the filter condition over the matching events. The condition refers to the properties of the event instances,*
- *$Within$: is defined by a time span that is defined by the number of time units.*

Example. Consider a process execution engine that emits three different event types to reflect on the evolution of work items (task instances): *Started*, *Failed*, and *Completed*. The three events share the same schema on the form $< workitemID, resourceID, ts >$ with a reference to the work item instance, the resource who would execute the work item and the timestamp ts of the event. We can define an interval for successfully completed work items as:

```
SuccessfullyCompleted=HeIE.Start(Started).End(Completed)
.Exclude(Failed).KeyBy(workitemID)
.Condition(Completed.resourceID =Started.resourceID)
```

Figure 4 shows on top a stream of the instances of the different event types. D^{2}IA will not consider $Started(1, 5, 20)$ and $Completed(1, 7, 34)$ as an interval because the condition on $resourceID$ is not satisfied, row 1 in the figure. Also, row 3 is not considered as an interval because between *Started* and *Completed* an instance of *Failed* event for the same work item was observed. Only intervals on rows 2 and 4 are valid intervals as per the specification above.


Interval Stream A
...
A1
A2
A3
A4
...
Interval Stream B
...
B1
B2
B3
...
A d B: A occurs during B, A di B: B occurs during A,
A > B: A occurs after B, A < B: A occurs before B,
A o B: A overlaps with B

	A1	A2	A3	A4
B1	A1 di B1	A2 > B1	A3 > B1	A4 > B1
B2	A1 o B2	A2 o B2	A3 > B2	A4 > B2
B3	A1 < B3	A2 o B3	A3 d B3	A4 > B3

Fig. 5. Interval-interval and interval-point temporal relationships.

3.3 Intervals Temporal Relationships

Event Interval Operators is a family of D^{2}IA operators which is based on Allen's interval relationships [3]. These operators can efficiently reason about interval temporal relationships occurring between the generated interval events. The interval operator is a binary operator that takes as one input, a stream of interval events and as the other input, either another interval stream or a point-based event stream, but not both. The operator produces point-based *match* events whenever a match is found between two interval events (see `I2E` in Fig. 1).

Unlike stateless stream processing, e.g. [19], where each element is processed independently, the derivation of interval relationships is a stateful operation [2, 14, 16]. Thus, this calls for a temporal frame (e.g. windows) to collect a finite subset of stream elements for both inputs. Therefore, we designed our interval operator to work on a tumbling window [9]. The results of the interval operator is a stream of match events. Definition 6 describes the inputs for the interval operator.

Definition 6 (Match Event). *A match event is an instantaneous event resulting from the reasoning about the relationship occurring between two interval events. Match specification is defined by a tuple* $< IntEvent1, IntEvent2, Frame1, Frame2, Relations, TSFunction >$ *where:*

- $IntEvent1$ *and* $IntEvent2$*: refer to the interval event types to reason about.*
- $Frame1$ *and* $Frame2$*: refer to the the temporal scope, i.e., the windows required to process (join) the interval event streams.*
- $Relations$*: refers to a list of temporal relationships to match between the interval events in the scopes.*
- $TSFunction$*: is a function used to assign a timestamp to the output match event which is instantaneous. Three timestamp function are available: now(), i.e., current system time; earliest (latest), i.e., assign the oldest (most recent) time instant choosing from the start-/end-points of the matched events.*

Listing 3.5 shows an example of an interval operator that works on temperature and smoke interval streams.

Listing 3.5. Interval operator specification

```
Match=IntervalOperator.Event1(TemperatureDelta)
.Frame1(5 minutes).Event2(SmokeThreshold)
.Frame2(5 minutes).Relation([Relations.During])
.Timestamp(TimestampFunctions.LATEST)
```

Grossniklaus et al. [12] defined data-dependent predicates that characterize the structure of a frame and, thus, influence the computation performance. Therefore, to the extent of computing temporal interval relationships, we define our frames to consider maximum intervals. This assumption, formalized in Definition 7, is relevant because it allows performance gain by minimizing the number of interval events to compare. Thus, our operator provides `I2I` transformation as shown in Fig. 1.

Definition 7 (Maximal Interval). *Let I be the set of all possible interval instances generated by an interval Specification. An interval $i \in I$ is maximal iff $\forall j \in I, j \neq i : i \; < \; j \vee i \; > \; j \vee i \; m \; j \vee i \; mi \; j$.*

As per Definition 7, we can be sure that the temporal relationships between the *sorted* elements of the same interval stream is always $i_j < i_{j+1}$ for $j \geq 0$. The benefit of this property is that we can efficiently calculate temporal relationships between pairs of interval instances of the left $Event1$ and the right $Event2$ interval streams without having to explicitly compare timestamps of each pair. If contents of each window on the two inputs of the join are sorted by the start timestamp, we can utilize the transitivity of temporal relationships [3] to efficiently compute the temporal inter-relationships between interval instances. For the cases where we have to compare timestamps of intervals, we rely on the efficient set-theoretic approach presented in [11].

Example. Consider the two interval streams A and B in Fig. 5. The dashed rectangles represent the content of a window over each stream. Note that windows width not necessarily should be the same. Within each window, the content is sorted by the timestamp. As per Definition 7, the intervals are either before, after or meet each other. Suppose that we want to define the temporal relationships between contents of the window on stream A and the content of the window on stream B. Namely, we need to find the relationship between $A1, A2, A3, A4$ on the one hand and $B1, B2, B3$ on the other hand. The naïve way to implement that is to perform 12 comparisons. A more efficient way is to infer the type of inter-stream interval relationships utilizing the nature of intra-stream interval relationship. Looking at the right table in Fig. 5, when we compare $A1$ with $B1$, using their start and end timestamps, we can find that $A1$ *contains* $B1$, i.e., $A1 \; di \; B1$. As we learn this relationship, we can deduce the temporal relationship between the other A intervals and $B1$. Since any interval $Ai, i > 1$ will always occur after $A1$, we can deduce the same relationship between those intervals and $B1$. This is represented with gray cells in the table. By comparing $A1$ and $B2$, we find that $A1$ overlaps with $B2$, i.e.,$A1 \; o \; B2$. We can not infer an exact relation between other intervals in the A stream and $B2$ because $B2$ ends after $A1$ does. When we compare $A2$ and $B2$, we still find $A2 \; o \; B2$. However, we can find that $A2$ ends after $B2$. Thus, we can deduce that future intervals of A will always occur after $B2$. Finally, we have to compare each A interval with $B3$.

Algorithm 1. HoIE Query Generation

```
Input: An Event type T; An Occurrence Expression Min,Max; An Absolute Condition Ca;
       An Relative Condition Cr; An aggregation function F; An aggregation key K; A
       temporal window W
1  patterns ← [] start ← e0
2  for k ← 0 to Min do
3      start ← followedBy start ei
4  if Max == Unbounded then
5      pattern ← where Ac [followedBy start constrain(Cr, ei)*]
6      add pattern patterns
7  else
8      for i ← min, i<=max, i++ do
9          pattern ← start
10         for j ← 0, j<i, j++ do
11             pattern ← where Ac [followedBy pattern constrain(Cf, ei)]
12         add pattern patterns
13 foreach p ∈ patterns do
14     output Head as select F (Kp) from p.window(W) where Ca
15 e0...ei with i=Max are of type T
```

4 Proof-of-Concept

In this section, we demonstrate how D^{2}IA fills the gap between scalable streaming data platforms and expressive centralized streaming solutions. In particular, we investigate two complementary proof-of-concepts (i) we demonstrate how we can rewrite D^{2}IA into an expressive CEP languages, i.e., EPL, that supports interval events and Allen's relations, but was not implemented on a scalable infrastructure. (ii) We present our implementation for a scalable version of D^{2}IA on top Apache Flink, a popular distributed stream processing engine.

4.1 Translation to EPL

EPL is an industrial stream processing language that combines DSMS and CEP features. The combination of pattern matching and analytical queries relies on the notion of Stream/Event type. I.e., every EPL stream has a schema that corresponds to the type of the items it contains. EPL allows the creation of new streams combing events w.r.t. the order of their appearance. A special operator called "insert into" allows composing queries in a query network. EPL natively supports the relationships of Allen's algebra. Algorithm 1 describes the steps of how HoIE operators can be translated into an EPL query network.

Algorithm 1 uses the following notations: *output T* places an event of type T to a target stream; *where* applies absolute conditions to a pattern; and *constrain* applies relative conditions to an event. Moreover, it makes use of the following DSMS/CEP operators $window$, $select$, $from$, and $followedBy$, $Kleen's\ star(*)$, and *not*. Listing 4.1 shows the delta interval from Listing 3.3 translated to EPL.

Listing 4.1. EPL code for the Delta example

```
create context PartitionedById
partition by sensor from Temperature
context PartitionedById
insert into Delta select Math.avg(temps), startp, endp
from Temperature
match_recognize (measures A as temps, A[0].ts as startp, last(A.ts) as endp
pattern (A{2,})
define A as   Math.abs(A.temp - Math.min(A.temp) >=2));
```

4.2 Implementation on Flink

`Apache Flink` is a scalable stream processing engine. It supports S2R transformation by applying different windowing on data streams. R2R is also supported by manipulation of window contents. It also supports R2S as the results of windowing can be emitted to other streams. In this context, E2E transformations can be achieved using `Flink CEP` library[5]. In particular, `FlinkCEP` is a complex event processing library defined on top of Flink. We use FlinkCEP APIs to realize the interval generator operators, E2I. We use a so-called looping pattern to define the `Occurrence` property of the interval specification. Relative and absolute conditions are implemented via so-called `IterativeConditions`. To compute the aggregate value of the interval, we leverage the feature in FlinkCEP that returns a sequence of all the matched raw events. We use the first and last elements of the sequence to obtain the timestamps that constitute interval's endpoints. Then, we create an instance of the interval type and populate its properties and add it to the respective stream, see Fig. 2. To realize the interval operator, I2E, we use the join operator of Flink. The join operator receives as input two streams. As discussed in Sect. 3.3, we use the time frames, see Definition 6, to bound the number of interval instances to check the match for. To realize I2I transformation, we employ a window operator on the generated interval stream and emit maximal intervals only. Our implementation with example intervals can be found on the project repository[6].

Evaluation. To evaluate our implementation on FlinkCEP, we use a data set of the linear road benchmark[7]. Linear Road is a simulation of a large metropolitan city which is 100 miles wide and long and consists of 10 parallel expressways. Each tuple in the data set describes a vehicle ID, its speed, road, direction and the timestamp of the record. In our evaluation, we created four different homogeneous interval

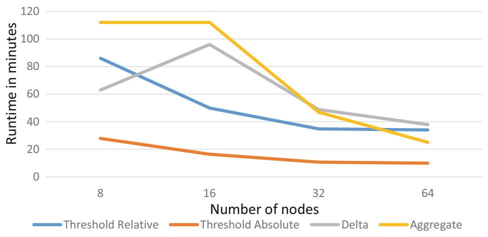


Fig. 6. Different intervals on Flink

[5] https://ci.apache.org/projects/flink/flink-docs-stable/dev/libs/cep.html.
[6] https://github.com/DataSystemsGroupUT/ICEP.
[7] http://infolab.stanford.edu/stream/cql-benchmark.html.

specifications, for the different frame types with absolute and relative conditions. We partition the data by vehicle ID and put the conditions on the speed attribute. More details about the interval specifications can be found in the GitHub repository. Figure 6 shows the scalability results in terms of time needed by the different queries to process 1 *million* tuples of the data set under an increasing number of computing nodes in a Flink cluster. Each node is equipped with 30 *GB* of main memory and 16 cores at 2.0 *GHz*. For the different interval specifications, it is clear that Flink scales well as the number of nodes increases. However, for each interval specification, the processing time varies for the same processing capabilities due to the difference in the complexity of the interval specification.

5 Related Work

TPStream [14] introduced a stand alone operator that finds temporal relationships among intervals. TPStream allows defining homogeneous intervals with absolute conditions only. $\texttt{D}^2\texttt{IA}$ interval generation operators cover both homogeneous with absolute and relative conditions and heterogeneous intervals.

ISEQ [15] is an operator for reasoning about event intervals using Allen's temporal relationships. ISEQ assumes the existence of intervals and does not provide means to define. Compared to our work, we allow the user to define the intervals from raw (point) events. Moreover, we support both homogeneous and heterogeneous intervals, allow rich conditions on matching events, and calculating aggregations over values of raw events.

CEDR [6] is an event streaming system that embraces an interval-based temporal stream model to unify query language features, handles out-of-order event delivery, and defines correctness guarantees as well as operator semantics. CEDR's events have a validity interval, which indicates the range of time when the tuple is valid from the event provider's perspective. This is used to retrieve events which are still valid at query time. This case can be seen as an example of interval algebra reasoning. However, Allen's operators are not explicit in the language.

ETALIS [4] is an event-driven approach for Complex Event Processing. The language semantics is based on a logic programming. ETALIS represents events as facts and translates complex event patterns into logic rules. Thus, complex events are derived from simpler ones. ETALIS language is very expressive. Although it is possible to express and derive interval relationship across events, ETALIS does not provide any interval event generation mechanism. Events must adopt a two-timestamps temporal model. Moreover, the language does not exploit events ordering for optimizing reasoning about event interval.

Table 1 summarizes the comparison of $\texttt{D}^2\texttt{IA}$ with related work. The implementation of $\texttt{D}^2\texttt{IA}$ on top of `Flink` supports all operators from Fig. 1. In addition, this implementation leverages the performance scalability provided by `Flink`. Functionality-wise, EPL supports all the operators. However, this has to be done manually by the developer which is error-prone.

Table 1. Operators coverage and scalability comparison

Feature/System	EPL	TP-Stream	ISEQ	ETALIS	CEDR	Flink	Flink+D2IA
Operator							
S2R	+	−	−	−	−	+	+
R2R	+	−	−	−	−	+	+
R2S	+	−	−	−	−	+	+
E2E	+	−	+	+	+	+	+
E2I	+	+	−	−	−	−	+
I2E	+	+	+	+	−	−	+
I2I	+	−	−	−	−	−	+
Scalability	−	*N/A*	−	−	−	+	+

6 Conclusion

In this paper, we presented a family of operators to specify event intervals over data streams and to reason about their temporal relationships ($\texttt{D}^2\texttt{IA}$). $\texttt{D}^2\texttt{IA}$ supports event intervals derived from single source stream by means of aggregations over timestamped events (homogeneous), and event intervals derived from two or more sources (heterogeneous). $\texttt{D}^2\texttt{IA}$ translates intervals specification into complex CEP specifications; it allows to cover a wide range data-driven intervals as rich conditions regarding events inclusion; and it supports a wide range of aggregations or references to composing events values. We realized a proof of concept using both EPL and on top of `Apache Flink`. As an interval operator, $\texttt{D}^2\texttt{IA}$ is more expressive than similar approaches as it allows relative conditions which allows defining a wider range of homogeneous intervals than related approaches.

References

1. Akidau, T., et al.: The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. PVLDB **8**(12), 1792–1803 (2015)
2. Alharbi, A., Bulpitt, A., Johnson, O.: Improving pattern detection in healthcare process mining using an interval-based event selection method. In: Carmona, J., Engels, G., Kumar, A. (eds.) BPM 2017. LNBIP, vol. 297, pp. 88–105. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-65015-9_6
3. Allen, J.F.: Maintaining knowledge about temporal intervals. Commun. ACM **26**(11), 832–843 (1983)
4. Anicic, D., Rudolph, S., Fodor, P., Stojanovic, N.: Stream reasoning and complex event processing in ETALIS. Semant. Web **3**(4), 397–407 (2012)
5. Arasu, A., Babu, S., Widom, J.: The CQL continuous query language: semantic foundations and query execution. VLDB J. **15**(2), 121–142 (2006)

6. Barga, R.S., Goldstein, J., Ali, M.H., Hong, M.: Consistent streaming through time: a vision for event stream processing. In: CIDR, pp. 363–374 (2007)
7. Codd, E.F.: A database sublanguage founded on the relational calculus. In: Proceedings of the ACM-SIGFIDET Workshops (1971)
8. Cugola, G., Margara, A.: Low latency complex event processing on parallel hardware. J. Parallel Distrib. Comput. **72**(2), 205–218 (2012)
9. Dindar, N., et al.: Modeling the execution semantics of stream processing engines with secret. VLDB J. **22**(4), 421–446 (2013)
10. Etzion, O., Niblett, P.: Event Processing in Action. Manning, Shelter Island (2010)
11. Georgala, K., Sherif, M.A., Ngomo, A.N.: An efficient approach for the generation of Allen relations. In: ECAI, pp. 948–956 (2016)
12. Grossniklaus, M., Maier, D., Miller, J., Moorthy, S., Tufte, K.: Frames: data-driven windows. In: DEBS, pp. 13–24. ACM (2016)
13. Hirzel, M., Baudart, G., Bonifati, A., Valle, E.D., Sakr, S., Vlachou, A.: Stream processing languages in the big data era. SIGMOD Rec. **47**(2), 29 (2018)
14. Körber, M., Glombiewski, N., Seeger, B.: TPStream: low-latency temporal pattern matching on event streams. In: EDBT, pp. 313–324 (2018)
15. Li, M., Mani, M., Rundensteiner, E.A., Lin, T.: Complex event pattern detection over streams with interval-based temporal semantics. In: DEBS (2011)
16. Ostovar, A., Maaradji, A., La Rosa, M., ter Hofstede, A.H.M.: Characterizing drift from event streams of business processes. In: Dubois, E., Pohl, K. (eds.) CAiSE 2017. LNCS, vol. 10253, pp. 210–228. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-59536-8_14
17. Paschke, A.: ECA-RULEML: an approach combining ECA rules with temporal interval-based KR event/action logics and transactional update logics. CoRR (2006)
18. Richter, F., Seidl, T.: TESSERACT: time-drifts in event streams using series of evolving rolling averages of completion times. In: Carmona, J., Engels, G., Kumar, A. (eds.) BPM 2017. LNCS, vol. 10445, pp. 289–305. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-65000-5_17
19. van Zelst, S.J., Fani Sani, M., Ostovar, A., Conforti, R., La Rosa, M.: Filtering spurious events from event streams of business processes. In: Krogstie, J., Reijers, H.A. (eds.) CAiSE 2018. LNCS, vol. 10816, pp. 35–52. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-91563-0_3