D^{2}IA: User-Defined Interval Analytics on Distributed Streams

Ahmed Awad[a,b], Riccardo Tommasini[a], Samuele Langhi[a], Mahmoud Kamel[a], Emanuele Della Valle[c], Sherif Sakr[a]

[a] *University of Tartu, Tartu, Estonia*
[b] *Cairo University, Giza, Egypt*
[c] *Politecnico di Milano, Italy*

Abstract

Nowadays, modern Big Stream Processing Solutions (e.g. `Spark`, `Flink`) are working towards being the ultimate framework 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 for 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 `Flink`, a distributed stream processing engine for big data.

Keywords: Big Stream Processing, Complex Event Processing, User-defined

Email addresses: `ahmed.awad@ut.ee` (Ahmed Awad), `riccardo.tommasini@ut.ee` (Riccardo Tommasini), `samuele.langhi@ut.ee` (Samuele Langhi), `mahmoud.shoush@ut.ee` (Mahmoud Kamel), `emanuele.dellavalle@polimi.it` (Emanuele Della Valle), `sherif.sakr@ut.ee` (Sherif Sakr)

1. Introduction

Streaming data analytics has become relevant as never was before. Together with Data Volume and Variety, the raise of Data velocity is forcing many organizations to embrace the real-time paradigm shift. A data stream, which is an unbounded sequences of partially-ordered data, is a convenient abstraction when data naturally comes over time, e.g., data from a sensor network. Intuitively, the unbounded nature of streams impacts the way data-systems handle query-answering. Information needs to become continuous, i.e., from an unbounded input an unbounded output is expected. To this extent, a new generation of Stream Processing engines (SPE) for Big Data (BigSPE) is emerging to process vast, heterogeneous, and noisy data streams [16].

SPEs are commonly classified into Data Stream Management Systems (DSMSs) and Complex Event Processing (CEP) [10] systems. The state-of-the-art on DSMSs and CEPs is vast and includes a variety of Domain Specific Languages (DSL) to analyse data streams. Most of these DSLs are declarative and expose special operators to deal with streams' unboundedness. In particular, most of DSMSs adopt time-based windows to slice the input streams into finite portions, upon which they can perform stateful aggregations [11]. On the other hand, CEP engines employ regular languages to detect events patterns over streams [12] using Non-deterministic Finite State Automata (NFSA).

Listing 1.1: DSMS query in EPL

```
1 select avg(val)
2 from Temperature#time(5m)
3
4 output every 5 min;
```

Listing 1.2: CEP query in EPL

```
1 insert into Fire
2 select *
3 from pattern
4 [Smoke -> Temperature(val>40)];
```

Listing 1.1 and Listing 1.2 show a DSMS and a CEP query, respectively. The former calculates the average temperature over the last 5 minutes, while

the latter emits a fire event whenever it detects a smoke event *followed-by* a temperature event that reports a value higher than 40. Both listings make use of an industrial DSL called Event Processing Language (EPL) [1]. EPL combines DSMS and CEP features into a hybrid solution that is very expressive. Interestingly, existing EPL implementations like `Esper`[2] and `OracleCEP`[3] can only scale-up. While vertical scalability is sufficient for a variety of use-cases, Big Data applications often call for fault-tolerant and horizontally-scalable BigSPEs. Nonetheless, the need for democratizing Big Data brought many BigSPEs to adopt SQL-like DSL for stream processing.

In this paper, we advocate that the trade-off between expressiveness and scalability led BigSPEs to design APIs and DSLs that do not meet the expectations raised by the centralized solutions [16].

Nevertheless, such expressiveness is crucial in several applications like the following air traffic scenario inspired to Bombardier's C Series jetliner. Such plane, designed in 2015, is fitted with 5,000 sensors that generate up to 10 GB of data per second. Many events are continuously produced during flights, e.g., changes in altitude, speed, and heading of an aircraft. In such a scenario, we can be interested in detecting those events *during* which a plane is in cruising mode and performs a change in altitude which is more than 10%. We can use EPL to design a solution for this scenario (cf Listing 1.3). However, to the best of our knowledge, solutions like Flink or Spark Streaming do not provide such feature out-of-the-box and require some customization to be used. Indeed, while the query above requires to process events that have a duration, existing BigSPEs adopt a point-based time semantics.

The literature on Stream Processing contains many examples that acknowledge the limitations of a point-based time model vs an interval-based one. The latter has a richer semantics than the former and can still represent point events

[1] `https://docs.oracle.com/cd/E13213_01/wlevs/docs20/epl_guide/overview.html`
[2] `http://www.espertech.com/`
[3] `https://docs.oracle.com/cd/E17904_01/doc.1111/e14476/`

without loss of generality [3].

Listing 1.3: 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);
```

In the remainder of the paper, we address the problem of *enabling expressive yet horizontally scalable stream processing*. To this extent, we designed and implemented D^{2}IA (Data-driven Interval Analytics), a *novel* family of operators that enables interval events generation and reasoning. This paper is an extension of a previous one presented in [6] adding the following contributions:

- The paper provides two implementations of the D^{2}IA operator family, based on alternative design decisions, on a large-scale stream processing systems, i.e., Apache Flink;

- It presents a systematic and comparative evaluation of the alternative implementations using a well-known reference benchmark for stream processing, i.e., Linear Road Benchmark;
- It discusses about the portability of the approach in relation with other Big SPEs and how the trend towards a StreamingSQL can foster expressive yet horizontally-scalable stream processing;
- It improves the general presentation of the paper and includes a more detailed background section.

The remainder of the paper is organized as follows. Necessary background is introduced in Section 2. Concepts behind D^{2}IA are presented in Section 3. Section 4 describes the implementation details, and Section 5 presents the evaluation. Related work is discussed in Section 6, and a discussion on the relation between D^{2}IA and competing alternatives is presented in Section 7. We finally conclude the paper in Section 8.

2. Background

In this section, we summarize the state-of-the-art on DSMS and CEP presenting the main concepts that are required to understand the content of the paper.

2.1. Data-Stream Management Systems

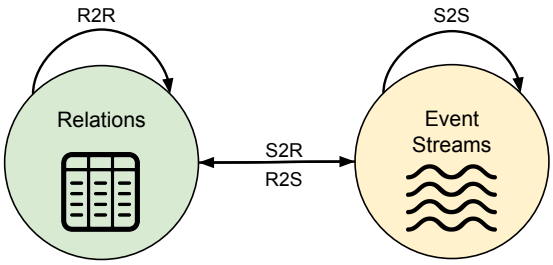


Figure 1: Operators of the Continuous Query Language [4]

Data-Stream Management Systems (DSMS) are an extension of Database Management Systems to process data streams. As mentioned in Section 1, data streams differ from traditional data because they are unbounded. The Continuous Query Language (CQL) is a query model that extends traditional database query languages to deal with the unboundedness of the data [4]. Figure 1 summarizes CQL's model. Stream-to-Stream (S2S) transformations are generally stateless queries. Additionally, three families of stateful operators allow moving from stream to relation and vice-versa, i.e., Stream-to-Relation (S2R) operators that produce a relation from a stream, e.g., windows; Relation-to-Relation (R2R) operators that produce a relation from one or more other relations, e.g., relational algebra; and Relation-to-Stream (R2S) operators that produce a stream from a relation, e.g., RStream, which will be explained shortly.

Windows are the canonical S2R operators that slice an unbounded stream into finite chunks [4, 11, 15]. Window operators can be time-based or data-driven (also known as frame [13]).

Time-based windows divide the streams into intervals of the same *width*. In case of *Time-Based Sliding Windows* the intervals can overlap. These windows are used to monitor phenomena whose duration is known or can be derived from the application domain. For instance, in case of a smart home application, we can decide to monitor the temperature in the last 15 minutes, knowing the transitory will not exceed the window width. On the other hand, data-driven windows are used when the duration of a phenomena is unknown, but it can be estimated from the input data. In the same smart-home application, we can count the number of people currently in the house to decide to turn on or off the air conditioning.

Several data-driven windows have been discussed in literature, but a general formulation of their specification is missing. In the following, we list those cases that are relevant from the content of the paper, i.e., data-driven based on aggregations called Frames [15]: (i) **Threshold Frames** divide the stream into intervals whenever an attribute of a stream element goes higher (lower) than a given threshold. (ii) **Delta Frames** divide the stream into intervals whenever

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) **Boundary Frames** divide the streams into intervals until an attribute of the stream elements remains within one of the predefined boundaries.

The original R2S operator family includes three operators that create a data stream out of a relation, i.e.,: (a) The result stream (RStream) that outputs all elements at a certain instant in the source relation. (b) The insert stream (IStream) outputs all *new* entries w.r.t. the previous instant. (c) The delete stream (DStream) outputs all deleted entries w.r.t the previous instant.

2.2. Complex Event Processing

Complex event processing aims at the identification of patterns that represent complex events over input (raw event) streams (pattern-matching) [12]. Patterns are analogous to regular expressions over strings. They are defined as sequences of event types and evaluated using NFSA against the input streams.

As mentioned in the introduction, BigSPEs adopt a point-based time semantics using a unique timestamp. Therefore, for backwards compatibility, we consider the Raw Events as instantaneous too (cf Definition 1).

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

We represent a Raw 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.

An Interval Event is derived from a sequence of events using pattern matching (cf Definition 2). The pattern is expressed using event types, which are determined by id and payload. For instance, consider the following pattern *FireEvent = SmokeEvent followed-by HighTemperatureEvent.*

Definition 2 (Interval Event). *An interval event is an event derived by the composition of one or more events. It has a temporal duration which is defined in terms of two time points,* start *and* end*.*

The resulting interval event also has a payload like a raw event (Definition 1), but also includes a duration. The payload of the interval event is a function of the input events' payloads, but its specification is up to the developer in terms of selection and aggregation functions. To this extent, we introduce two auxiliary functions that support the custom payload creation: (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.

On the other hand, the reasoning about the duration should be more rigorous. We calculate it as the difference between *end*, which is the timestamp of the event that terminates the pattern-matching, and *start*, which is the timestamp of the event that initiated it (e.g.,FireEvent.start=SmokeEvent.ts; FireEvent.end=HighTemperatureEvent.ts) Moreover, Allen's interval Algebra [2] is probably the most known formalism to deal with temporal reasoning. The algebra contains 13 binary relations intervals for representing temporal information and addresses the problem of reasoning about such intervals. The problem of computing all the relation closure is NP-Hard [24], but many fragments have been identified where reasoning about time can be efficient [21].

3. Operators for User-defined Intervals Analytics

In this section, we present a family of operators for analytics contextual reasoning about events called $\texttt{D}^2\texttt{IA}$. $\texttt{D}^2\texttt{IA}$ allows generating data-driven Interval Events from Raw Events, and reasoning about time interval using Allen's Algebra. In particular, $\texttt{D}^2\texttt{IA}$ is designed to offer:

R.1 **Event Generation**, i.e., the operators must enable interval generation via event detection and vice versa.

R.2 **Analytical Features**, i.e., the operators must enable (a) stateful aggregations, for example employing temporal/physical/data-driven windows,

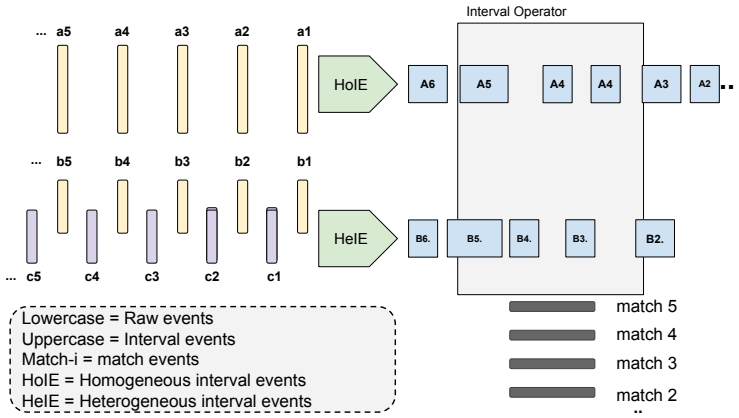


Figure 2: D^{2}IA overview: Homogeneous/Heterogeneous Interval Event generators consume events streams and produce interval events.

and they must allow (b) the definition of contextual variables and partitioning of the stream.

R.3 **Stream Reasoning**, i.e., the operators must enable reactive reasoning about interval events.

Figure 2 exemplifies a pipeline in which Raw Events from two input streams are transformed into Interval Events using *interval generators* and fed into an *interval operator* that reasons about the interval events.

To design D^{2}IA, we followed Codds principles for language design: **Minimality**, i.e., a language must provide only the necessary constructs avoiding alternative equivalent expressions; **Symmetry**, i.e., any language construct must always express the same semantics regardless of the usage context, and **Orthogonality**, i.e., any combinations of language constructs should be applicable [9, 22].

Figure 3 summarizes the spectrum of the streaming language operators extending the CQL model proposed by Arasu et al. [4] with three new families of operators, i.e., Stream-to-Interval (S2I), Interval-to-Interval (I2I), and Interval-to-Stream (I2S) operators. D^{2}IA allows detecting patterns of instantaneous

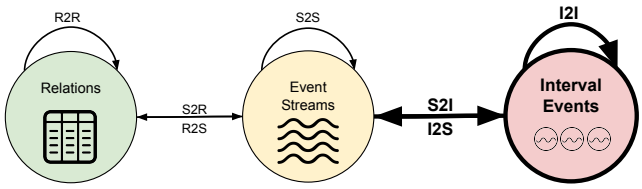


Figure 3: Stream-Event-Interval Event Models and Operators

events to compose interval events (**S2I**); it enables efficient reasoning about interval events using Allen's Algebra [2] (**I2I**). Finally, it allows instantaneous event generation and aggregation from intervals (**I2S**).

3.1. Events Generators

Event generators represent a family of $\mathtt{D^2IA}$ operators which are responsible for creating interval events out of a stream of instantaneous events and vice versa. In particular, the interval generator transforms the input stream(s) into the output stream based on a pattern specification.

Definition 3 (Homogeneous Interval Event Generator). *A homogeneous interval event generator is defined as a tuple* $< ET, MinO, MaxO, Window >$ *where:*

- ET *refers to the type of the event on which the interval is defined.*
- $MinO$ *indicates the minimum number of event instances to match.*
- $MaxO$ *indicates the maximum number of event instances to match. Also, wild card* $*$ *can be used to make no upper-bound on the number of occurrences.*
- *Window: specifies a maximum time interval to wait for the match to validate between successive events. An example is* 5 *seconds,*

A homogeneous Interval Event is generated when one or more events of the same type are observed in succession. Similarly to regular expressions, a homogeneous event interval has the form $\mathtt{A\{min,max\}}$, where $\mathtt{A}$ is the event type.
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 homogeneous event interval as shown by Listing 3.1.

Listing 3.1: Warm interval with absolute condition

```
1 IntervalTemperature=HoIE.Event(Temperature).Occurrence(2,5)
2 .Window(Within.of(5,seconds))
```

Definition 4 (Heterogeneous Interval Event Generator). *A heterogeneous interval event generator is defined as a tuple* $< ET1, ET2, ES >$ *where:*

- ***ET1*** *refers to the type of the start event for the interval;*
- ***ET2*** *refers to the type of the end event for the interval;*
- ***ES*** *refers to a set of event types not to be observed within the interval.*

An Heterogeneous Interval Event is generated when one or more (raw) events of different types are observed in succession. Differently from Homogeneous Events, D^{2}IA requires that the start and the end of the intervals are specified. Moreover, instances of other event types might be required *not* to be observed within the interval. As an example of a heterogeneous interval generation, consider the following example.

Example. Assume a critical temperature event that notifies when the temperature is too high. Moreover, consider a *Smoke* event that notifies the presence of smoke in a room. Finally, consider an event from the cooling system that notifies it is started. We can define an heterogeneous event interval as shown by Listing 3.2.

Listing 3.2: Heterogeneous Interval Generation

```
1 FireRisk=HeIE.Start(CriticalTemperature).Exclude(Cooling)
2 .Until(Smoke)
```

The whole FireRisk event is a heterogeneous interval delimited by a *CriticalTemperature* and *Smoke* events, which have different types. Moreover, the

two event instances should not be interrupted by a *Cooling* event to determine the risk of fire in the room.

Last but not least, we conclude this section by introducing the dual operator of interval generator. Indeed, as Figure 3 indicates, D^{2}IA allows the space of instantaneous events to be returned back from the space of interval events (I2S).

The instantaneous events generator responsible to generate instantaneous events out of interval events.

Definition 5 (Instantaneous Event Generator). *The instantaneous events generator is defined as* $< ETI, ETR, TS >$ *where:*

- ETI *refers to the type of the Interval Event.*
- ETR *refers to the type of the instantaneous event.*
- TS *is a temporal selector that defines the output timestamp. In particular, we consider the following alternatives: (i)* ***now*** *assigns the current system time; (ii)* ***earliest*** *assigns the start timestamp; (iii)* ***latest*** *assigns the end timestamp;*

3.2. Analytical Operators

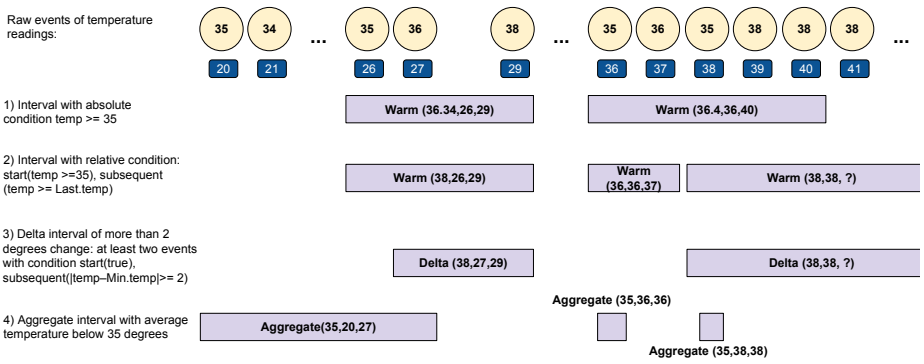


Figure 4: Homogeneous interval events for the different data-driven frames

Stateful aggregations and filters are analytics features typical of DSMSs. Since D^{2}IA aims at providing a unifying set of operation, it includes the following

analytical operators, which can be combined with the aforementioned event generators:

- *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*, *relative*, or *aggregate*. Absolute conditions compare an attribute of any event instance in a sequence with a constant value. Relative conditions compare the values of two attributes in the sequence of event instances. Finally, aggregate conditions compare the aggregated value of an attribute in the sequence of event instances with a constant value.

Listing 3.3: Critical temperature interval with absolute condition

```
1 Warm=HoIE.Event(Temperature).Occurrence(2,5)
2 .Within(5,seconds)
3 .Value(Aggregate.avg(Temperature.temp))
4 .KeyBy(Temperature.sensor)
5 .Condition(Conditions.greaterOrEqual(Temperature.temp,35))
```

Example. Completing the Temperature example, we can define an event interval with warm temperature using an absolute condition. In Listing 3.3, the interval generator is instructed to generate an interval event of type `Warm`. 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 seconds 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.

Example. We can also define a Warm event interval to be detected when the temperature keeps increasing after an alerting threshold using a relative condition as shown in Listing 3.4, an event interval of type `Warm` 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 4 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.

Listing 3.4: Critical temperature interval with relative condition

```
1 Warm=HoIE.Event(Temperature).Occurrence(2,5)
2 .Within(5,seconds).Value(Aggregate.max(Temperature.temp))
3 .KeyBy(Temperature.sensor)
4 .Condition(Conditions.greaterOrEqual(Sequence.first(
5 Temperature.temp),35)).Condition(Conditions
6 .greaterOrEqual(Sequence.current(Temperature.temp),
7 Sequence.last(Temperature.temp))
```

Example. Finally, it is possible to define an event interval using a delta frame which observes whether the temperature reading increases by more than 2 degrees. This can be defined as shown in Listing 3.5 or as an aggregate frame of, e.g., average temperature threshold of 35 degrees can be defined shown in Listing 3.6. In these interval definitions, we keep adding events to the interval as long as the conditions to trigger are met. Example matches for the delta frame are illustrated on row 3 in Figure 4. Examples of matches to the aggregate interval definition are shown in row 4 in Figure 4. 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.

Listing 3.5: Delta interval

```
1 Delta=HoIE.Event(Temperature).KeyBy(Temperature.sensor)
2 .Occurrence(2,Occurrences.Unbounded)
3 .Value(Aggregate.max(Temperature.temp))
4 .Condition(Conditions.greaterOrEqual(Math.absolute
5 (Math.minus(Sequence.current(Temperature.temp),
6 Aggregate.min(Temperature.temp))),2))
```

Listing 3.6: Aggregate interval

```
1 Aggregate=HoIE.Event(Temperature).KeyBy(Temperature.sensor)
2 .Occurrence(1,Occurrences.Unbounded)
3 .Value(Aggregate.avg(Temperature.temp))
4 .Condition(Conditions.lessOrEqual(
5 Aggregate.avg(Temperature.temp), 35))
```

3.3. Stream Reasoning

Event Interval Operators is a family of D^2IA operators which is based on Allen's interval relationships [2]. These operators can 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 composite interval events whenever a match is found between two interval events as per Definition 6.

Definition 6 (Composite Interval Event Generator). *A Composite Interval Event Generator is defined as* $< IE1, IE2, A >$ *where:*

- $IE1$, $IE2$ *refer to the interval event types to reason about.*
- A *refers to a list of temporal relationships to match between the interval events in the scopes.*

The resulting composite interval event corresponds to the union of the input intervals, i.e., (start=min($IE1$.start,$IE2$.start), (end=max($IE1$.end,$IE2$.end))).

Listing 3.7 shows an example of an interval operator that works on temperature and smoke interval streams. In this specification, we assume two input streams of intervals `TemperatureDelta` and `SmokeThreshold`. Based on the specification, a new stream of `Match` is generated for each pair of input intervals where a `TemperatureDelta` interval occurs *during* a `Smokethreshold` interval. As a result, the generated `Match` is the union of the two intervals.

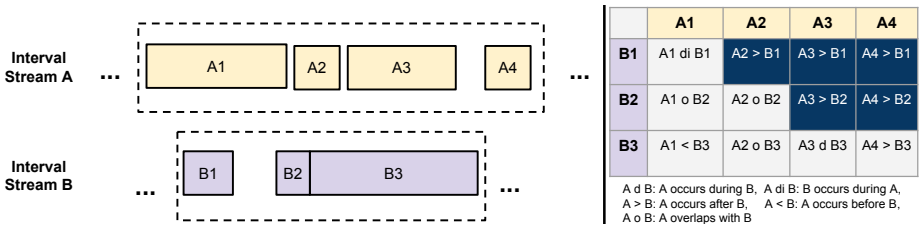
Listing 3.7: Interval operator specification

```
1 Match=IntervalOperator.Event1(TemperatureDelta)
2 .Event2(SmokeThreshold)
3 .Relation([Relations.During])
```

Grossniklaus et al [15] 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 maximal 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 Figure 3.

Definition 7 (Maximal Interval). *Let I be the set of all possible interval instances generated by an interval generator. An interval $i = [s, e] \in I$ is maximal iff $\forall j = [s, e] \in I, j \neq i : i.e \ < \ j.s \vee i.s \ > \ j.e \vee i.e \ = \ j.s \vee i.s \ = \ j.e$.*

Definition 7 ensures 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 $IE1$ and the right $IE2$ interval streams without having to explicitly compare timestamps of each pair. This is a stateful comparison that requires maintaining the transitivity table while comparing the intervals. Like for stream-to-stream join, we must make use of windowing to



	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

Figure 5: Interval-interval and interval-point temporal relationships.

perform the comparison statefully. Sorting the windows' elements by the start timestamp allows us to utilize the transitivity of temporal relationships [2] to efficiently compute the temporal inter-relationships between interval instances. For the cases where we have to compare the timestamps of intervals, we rely on the efficient set-theoretic approach presented by Georgala et al. [14].

Example. Consider the two interval streams A and B in Figure 5. The dashed rectangles represent the content of a window over each stream. Note that it is not necessary that the width of the windows to 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 Figure 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 navy blue 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 $B3$. in this way and for this example, all navy blue comparisons have been deduced without need to actually compare the intervals.

4. Implementation

In this section, we present how we implement `D`^{2}`IA` on top of a scalable infrastructure, i.e., `Apache Flink`, which is a fault-tolerant and a scalable distributed stream processing engine. Flink supports stateless and stateful stream processing; it supports several types of windows, queryable state, and, recently, also complex event processing.

4.1. CEP-based Implementation

In [6], we realized a proof-of-concept of S2I operators on top of Flink's CEP library. We extended that proof-of-concept into a full validation prototype that we will describe in the following. The idea behind this implementation is starting from a CEP engine and extending it with data-driven windowing capabilities (e.g. frames).

`Flink CEP` approaches pattern matching by means of constructing a non-deterministic finite automaton (NFA). The implementation faithfully follows the approach presented by Agrawal et al. [1]. For the pattern specified, an NFA is derived and at runtime, the library tries to find matches to the pattern by instantiating the NFSA for each incoming event. Upon reaching an accepting state, the library declares a match and delivers the collection of events matched to the pattern and invokes the user-defined code to formulate the complex event that will be generated on the respective stream.

In the context of `D`^{2}`IA`, S2I transformations can be achieved by defining the interval as the result of the pattern match. 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`. For the absolute condition case, we use the `where` filter, whereas with the relative condition, we use the `until` filter. The former keeps accepting matches as long as the condition is true. The `until` filter specifies a stop condition for the looping pattern.

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 Figure 2.

For the generation of maximal intervals, we employ the `skip past last event` strategy. That is, `Flink CEP` will consume all events contributing to the current intervals and will skip matching them to new intervals, that is no overlapping of intervals. The choice of maximal intervals boosts the performance as the library will maintain at most one partial NFSA per unique key at runtime.

`Flink CEP` defines a *within property*. However, the semantics is different from our intention. It tries to find the match within the time window specified in the property. If no match is found, the partial NFSA is dropped by the end of the window. However, in our settings, the *within* property is limiting the time span between two consecutive raw events contributing the the interval, cf. Definition 3. So, we implement the within semantics as part of the `IterativeCondition`.

As an example, in Figure 6 we are interested to find threshold intervals over a stream of temperature readings, for those events that have a temperature above 35 degrees. Moreover, we are interested in building maximum intervals. The figure shows on the left the HoIE specification. The CEP operator takes this specification as input. On the right, Figure 6 shows an instance of the finite-state machine at runtime. Event-by-event, the machine checks whether the element validates the matching condition. The operator state is updated for each partial match, while any event that does not respect the threshold breaks the match

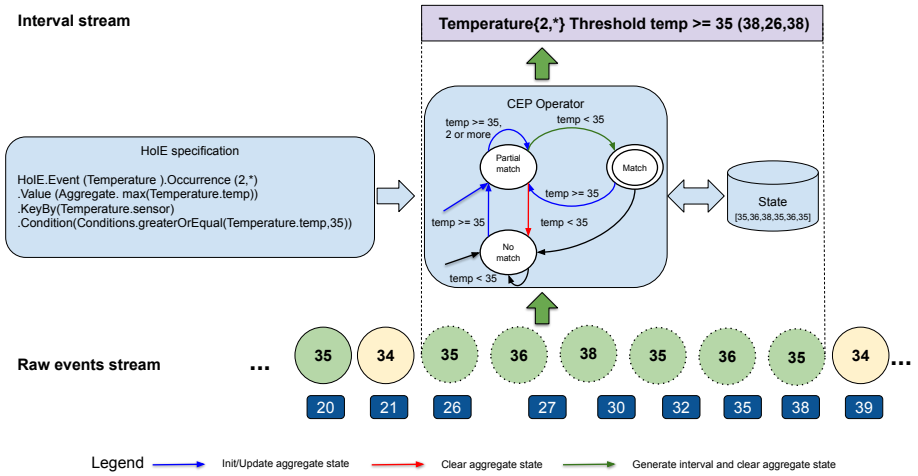


Figure 6: Interval generation using Flink's `CEP library`. Arrows indicate the valid transition across states.

and causes the automaton to restart. Intermediate stages are persisted within the state storage to guarantee fault tolerance. For matching results the opening and closing events are used to compute the window boundaries, as indicated by the dashed lines.

4.2. Window-Based Implementation

A different approach to $\texttt{D}^2\texttt{IA}$ implementation is the extension of a DSMS with pattern matching features. Therefore, we realized the S2I operators on top of Flink's stream processing libraries that provide DSMS operators such as windowing and stream-to-stream joining.

A stream-to-stream join operator receives two input streams and requires a window operator to bound the scope of the computation. We chose a time-based tumbling window to slice the input streams into finite portions. The elements belonging to these fixed intervals are delivered to a `ProcessWindowFunction` that applies the application-specific logic and *emits* the results.

As discussed in Section 3.3, we use the time frames to generate interval events with maximal intervals only. Therefore, it is very likely for an interval to span over two or more tumbling windows as shown in Figure 7. To alleviate

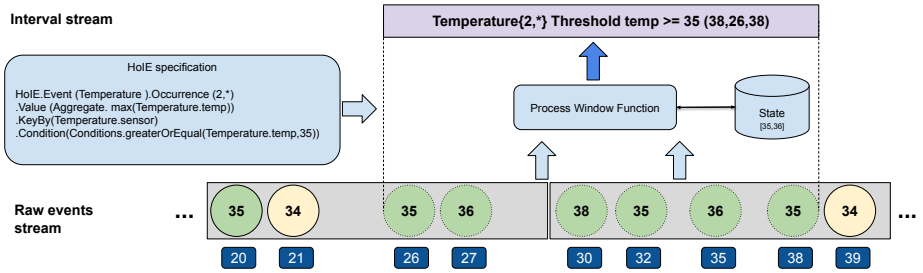


Figure 7: Tumbling window-based interval generation using Flink's `ProcessWindowFunction`

this limitation, we make use of the so-called `Keyed State` in Flink. Keyed state allows applications to maintain arbitrary state per unique key over data streams. Access to the state is allowed through so-called rich transformations. In our case, we buffer partial matches to intervals in a keyed state. So, whenever Flink signals a completion of a tumbling window, we retrieve the state, if any, and add the content of the new window to it, sort the whole set by their timestamp and then programmatically apply the interval generation logic on the elements. Whenever an interval is found, it is emitted on the respective stream and the rest of the elements, if any, are put back in the keyed state to be retrieved with the next invocation of the `ProcessWindowFunction`. Our implementation with example intervals can be found on the project repository[4].

As an example, in Figure 7, we are interested in finding threshold intervals over a stream of temperature readings where the readings are above 35 degrees. The interval reports the maximum reading observed as its value. Assuming a tumbling window of 10 seconds width, Flink groups the data as shown at the bottom of the figure. With the end of each window, Flink invokes the process window function. In this case, data are sorted by their timestamp. Then, from the element at timestamp 26 the beginning of a new interval is detected. However, by the end of the processed elements, it is not clear whether the interval has ended, as the following element, e.g. the one with timestamp 30

[4] `https://github.com/DataSystemsGroupUT/ICEP`

might still belong to the same interval. Thus, the partial match of elements 26 and 27 are stored in Flink's keyed state. In the mean time, Flink is collecting elements for the next tumbling window. Similarly, at the end of the new window, process window function is invoked again. At the beginning, the partial interval is restored from the keyed state and the elements of the new window are added to it. By processing the event with timestamp 39, we observe that the interval has been completed, as the temperature drops below the threshold. At this time, the interval on the top of the figure is emitted on the respective interval stream. The element with timestamp 39 is discarded and nothing is stored back in the keyed state and the process continues with the next tumbling window.

5. Evaluation

In this section, we present the evaluation of the two alternative $\texttt{D}^2\texttt{IA}$ implementations, i.e., CEP-based and Window-Based, against a baseline implementation on top of Esper. We have chosen Esper as the state of the art centralized stream processing engine that provides a declarative language to express queries. We have encoded the different interval specifications as EPL queries. These EPL queries are listed in Appendix Appendix A.

5.1. Setup

We have conducted our experiments on a Flink cluster running Flink version 1.8.1 with 3 workers where each worker can host up to 3 task managers. Each task manager has a single task slot with a single core running at 2.0 GHz. Each worker has a maximum of 24 GB of main memory. Esper was running on a single node with the same amount of main memory. We ran Esper version 8.0

We used the dataset of the linear road benchmark[5]. 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 car ID, its

[5] `http://infolab.stanford.edu/stream/cql-benchmark.html`

speed, road, direction and the timestamp of the record. The data set contains around $24M$ tuples.

We have used Kafka as a data source. More specifically, we have setup 3 Kafka brokers, coordinated by 1 Zookeeper instance. Then, we have created a partitioned topic, with 9 partitions and a replication factor of 1, and uploaded the dataset to Kafka partitioned by the car ID.

In our experiments with Flink, we have evaluated with parallelism $1, 3, 6,$ *and* 9. With each setup, we were dividing the main memory among the task managers. That is, if we run with parallelism 1, we allocate the whole 24 GB to the task manager. If we run with parallelism 9, each task manager is allocated 2 GB of main memory. The reason for fixing the total amount of memory available, is to better isolate the effect of parallelism on the performance.

The rationale behind this setup is to learn when a scalable implementation (Flink) will be able to outperform a centralized implementation (Esper) [20].

5.2. Experiments

In our evaluation, we created four different homogeneous interval specifications as presented in Listings 5.1- 5.4. We partition the data by car ID and put the conditions on the speed attribute. For each of the interval specifications, we were concerned with generating maximal intervals. This is to ensure that the results can be used to compare the different implementations.

Listing 5.1: Threshold interval with absolute condition over linear road data set to find intervals in which speed ≥ 50

```
1 ThresholdAbs=HoIE.Event(Speed).KeyBy(Speed.carID)
2 .Occurrence(2,Occurrences.Unbounded)
3 .Value(Aggregate.max(Speed.value))
4 .Condition(Conditions.greaterOrEqual(
5 Sequence.current(Speed.value),50))
```

Listing 5.2: Threshold interval with relative condition over linear road data set to find intervals in which average speed ≥ 67

```
1 Aggr=HoIE.Event(Speed).KeyBy(Speed.carID)
2 .Occurrence(2,Occurrences.Unbounded)
3 .Value(Aggregate.max(Speed.value))
4 Condition(Conditions.greaterOrEqual(Speed.value,30))
5 .Condition(Conditions.greaterOrEqual(
6 Sequence.current(Speed.value),1.1*Sequence.last(Speed.value)))
```

Listing 5.3: Delta interval over linear road data set to find intervals in which average speed ≥ 67

```
1 Aggr=HoIE.Event(Speed).KeyBy(Speed.carID)
2 .Occurrence(2,Occurrences.Unbounded)
3 .Value(Aggregate.avg(Speed.value))
4 .Condition(Conditions.greaterOrEqual(
5 Math.absolute(Sequence.current(Speed.value) -
6 Sequence.first(Speed.value)),5))
```

Listing 5.4: Aggregate interval over linear road data set to find intervals in which average speed ≥ 67

```
1 Aggr=HoIE.Event(Speed).KeyBy(Speed.carID)
2 .Occurrence(2,Occurrences.Unbounded)
3 .Value(Aggregate.avg(Speed.value))
4 .Condition(Conditions.greaterOrEqual(
5 Aggregate.avg(Speed.value),67))
```

For the window-based implementation, we experimented with different window sizes. We experimented with window of size 10, 100, and 1000 seconds. The larger the window size, the less number of tumbling windows Flink needs to generate and thus less processing overhead in triggering the process window function and access to the state, cf. Figure 7. Yet, with larger windows, there could be a higher latency in reporting intervals. On the other hand, for the CEP implementation, there is no mechanism offered to control the level of batching elements and passing them to the automata to find a match. From

our experiments, this is done on an record-by-record basis.

We compare the results based on the *throughput*, processed records per second. We compute the metric by dividing the total processed records, $24M$ tuples, by the total runtime of the job needed to process them in each setup. We repeated each experiment three times and computed the average runtime to calculate the throughput.

5.3. Results

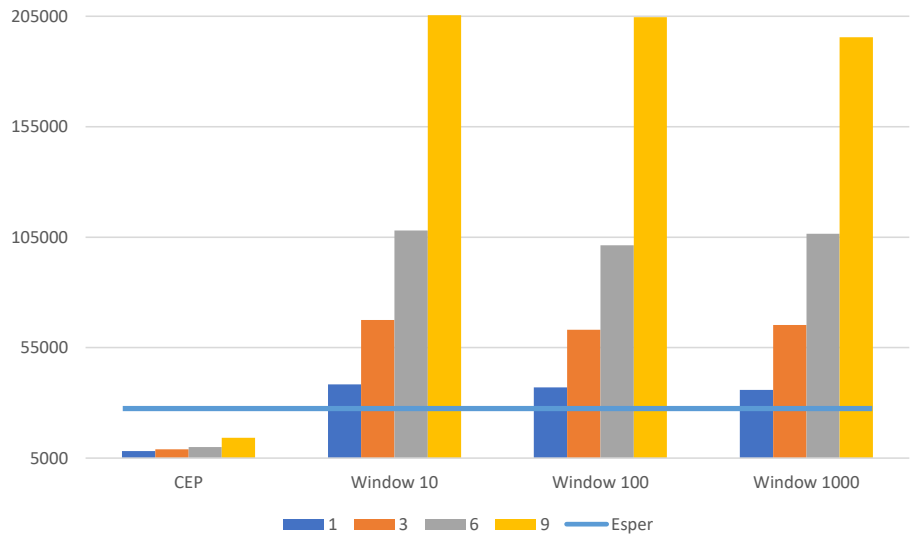
Figure 8 shows the throughput in records per second for the different implementations/configurations of the respective interval specifications from listings 5.1- 5.4. We also show the effect of increasing parallelism on the throughput of both window-based and CEP-based implementations. The horizontal line in each sub-figure represents the throughput of Esper for the respective interval specification.

In general, the window-based implementation outperforms CEP-based implementation. Starting from parallelism 3, the window-based implementation outperforms Esper. In the case of a threshold interval with absolute condition, the window-based implementation outperforms Esper for parallelism 1 (Figure 8a)

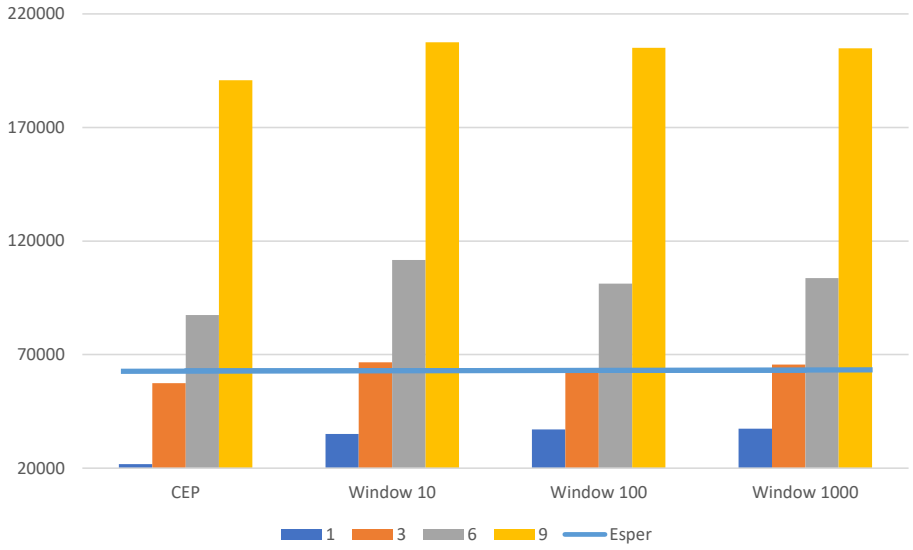
For the window-based implementation, by increasing the window size, we observed an enhancement of the throughput. This can be understood as the overhead of invoking the window function by Flink will be less and also the overhead of sorting the elements in the window. However, in a production setting, the larger the window size, the more the delay in reporting about detected intervals.

The magnitude of throughput increase, though, varies from one interval specification to the other. Threshold interval with relative condition (Figure 8b), delta interval (Figure 8c) have gained the most from increasing the parallelism. For both interval specifications, the throughput almost doubles with each increase in the parallelism for all the window sizes.

The same can be said about the aggregate interval (Figure 8d), although



(a) Threshold interval with absolute condition

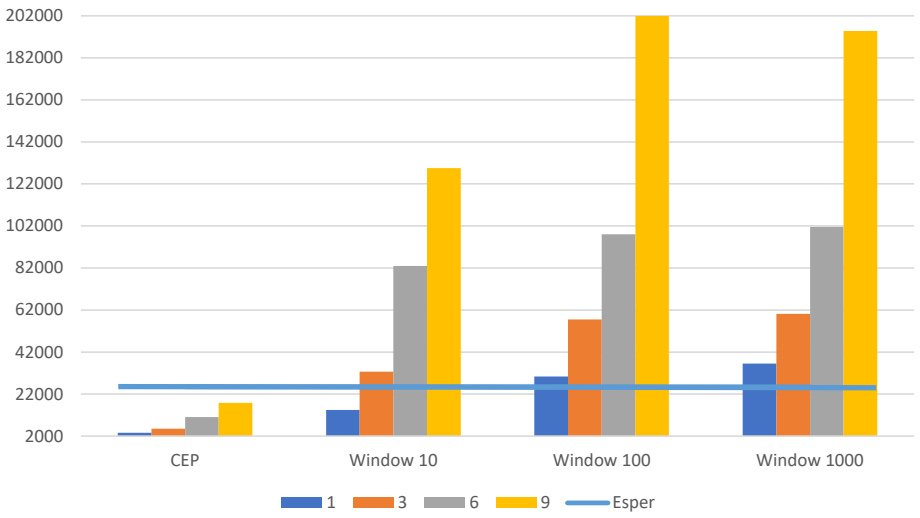


(b) Threshold interval with relative condition

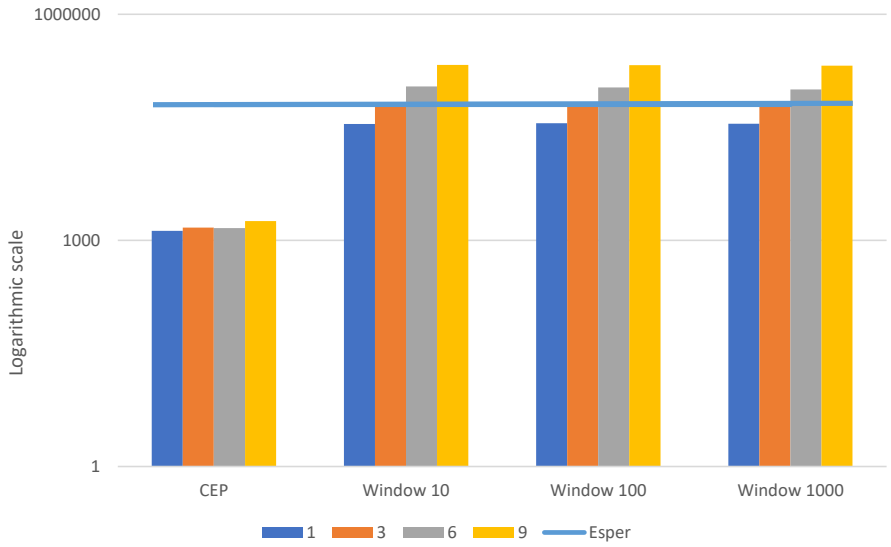
Figure 8: Throughput for the different queries with $1, 3, 6,$ *and* 9 workers

might not be clear on a logarithmic scale, for all window sizes. On the other hand,

The CEP implementation, in general, performs worse than Esper. Only



(c) Delta interval



(d) Aggregate interval

Figure 8: Throughput for the different queries with $1, 3, 6,$ *and* 9 workers (cont.)

for the interval with relative condition, the CEP implementation outperforms Esper starting from parallelism 6. We were expecting better performance for CEP on the interval with absolute condition, as the condition is checked on each

record individually. The actual results contradicts our expectations. By further investigations, we observed memory leakage within the CEP library of Flink as old matches and partial matches were not cleaned.

Scalability-wise, the window-based implementation benefits more from increasing the level of parallelism than the CEP-based implementation does. This is inherently due to the underlying window and CEP operators respectively.

6. Related Work

In this section, we present the work related to D^{2}IA considering the state of the art in complex event processing. Table 1 summarizes the comparison of D^{2}IA with related work. It shows that D^{2}IA's implementations on top of Flink supports all operators from Figure 3, while in the state of the art, only EPL supports all the operators but only in a centralized scenario. Moreover, in the following we briefly describe other CEP engines that are worth mentioning due to their expressiveness.

Feature /System	EPL	TP-Stream	ISEQ	ETALIS	CEDR	Flink	**Flink +D2IA**
Operator							
S2R	+	−	−	−	−	+	+
R2R	+	−	−	−	−	+	+
R2S	+	−	−	−	−	+	+
S2S	+	−	+	+	+	+	+
S2I	+	+	−	−	−	−	+
I2S	+	+	+	+	−	−	+
I2I	+	−	−	−	−	−	+
Scalability	−	*N/A*	−	−	−	+	+

Table 1: Operators coverage and scalability comparison

TPStream [18] introduced a stand alone operator that finds temporal relationships among intervals. TPStream allows defining homogeneous intervals with absolute conditions only. D^{2}IA interval generation operators cover both homogeneous with absolute and relative conditions and heterogeneous intervals.

ISEQ [19] 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 them. 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 [7] 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 [3] 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.

7. Discussion

The implementations used for $\texttt{D}^2\texttt{IA}$ validation makes use of the Flink's low-level streaming APIs, state management, and CEP. Despite being declarative, these APIs are embedded into programming languages and, thus, they are not completely portable to alternative systems, e.g., Spark or Kafka Streams.

Nevertheless, the support for SQL-based specification of data processing pipelines has been gaining attention by distributed data processing systems [5,

23, 17]. A declarative approach to specify processing logic may reach a wider range of users who are able to specify jobs without the need to be programmers. Moreover, SQL abstractions would foster a better unification between processing static data and streaming data [8].

Among the BigSPE systems, Flink was one of the early providers of SQL on streams through another Apache project, i.e., `Apache Calcite`, which can enforce general optimizations on the processing logic that are hard to be envisioned by the developer at job specification time. Interestingly, Apache Flink supports pattern matching over data streams since version 1.8.0, implementing the standard `Match_Recognize` SQL Clause using its CEP library.

Although Flink's Match_Recognize is still in its infancy within Flink SQL, it is worth to discuss how it relates with $\texttt{D}^2\texttt{IA}$, as it seems they are trying to answer the same research question: how to increase the expressiveness of BigSPE combining CEP and DSMS.

MR	$\texttt{D}^2\texttt{IA}$
Time Based Windows	Within, Frames
Measures	Value
Partition	KeyBy
Regular Language	Regular Language + Negation
Point Based Time	Interval Based Time

Table 2: Comparison of language clauses between Match_Recognize and $\texttt{D}^2\texttt{IA}$

Table 2 provides an high-level features comparison between Match_Recognize as it appears in Flink SQL and $\texttt{D}^2\texttt{IA}$. Both of the approaches provide the means for windowing but while $\texttt{D}^2\texttt{IA}$ users have access to expressive data-driven windows (i.e., Frames), Match_Recognize is limited to traditional temporal windows. For anything that is related to aggregations, both Match_Recognize and $\texttt{D}^2\texttt{IA}$ expose flexible selection operations (i.e., Value and Measures respectively). Moreover, both of the approaches take parallelism into account. For the case of event pattern detection, both $\texttt{D}^2\texttt{IA}$ and Match_Recognize use a simple regu-

lar language that consists of *next*, *Kleen's star*. Additionally, D^{2}IA allows the detection of absent event using negation for heterogeneous interval events, and Allen's algebra. Finally, a main difference between the two approaches regards the time model. Match_Recognize keeps Flink's single timestamp approach. On the other hand, D^{2}IA employs an interval-based temporal model.

Listing 7.1 shows a general template for the translation of a homogeneous interval specification, while heterogeneous ones cannot be implemented due to the absence of negation.

Listing 7.1: Flink SQL Match Recognize template for the homogeneous interval generation

```
1  Select Key, sts, ets, val from tbl
2  Match_Recognize(
3      Partition By Key
4      Order By timestamp
5      Measures A.Key as Key, First(A.timestamp) as sts,
6      Last(A.timestamp) as ets,
7      valOperator(A.value) as val
8      After Match skipStrategy
9      Pattern (A{minOccurs, maxOccurs} B)
10     Define
11         A as condition,
12         B as true)
```

The term *tbl* in Listing 7.1 refers to the continuous table wrapping the source event stream, cf. Definition 3. The elements of the generated interval are the start timestamp (sts), the end timestamp (ets) and the value computed over the elements of the interval (val). The pattern part enforces the number of occurrences required by the interval specification. The pattern term B is redundant and is just added to fulfill the current limitation in Flink's implementation of the Match_Recognize for greedy patterns. *valOperator* in line 7 refers to the function used to compute the value of the interval. The *condition* term in line 11 reflects the condition imposed on the elements being matched, either absolute or

relative conditions. Moreover, the *within* property of the interval specification is encoded as part of the condition. This is imposed by checking the difference of the timestamps of the successive events.

8. 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 a single source stream by means of aggregations over timestamped events (homogeneous), and event intervals derived from two or more sources (heterogeneous). In addition to our former work, we extensively evaluated two $\texttt{D}^2\texttt{IA}$ implementations based on alternative abstractions on top of Apache Flink. We evaluated and compared the two implementations systematically using the Linear Road Benchmark to assess their scalability. Last but not least, we discuss the relation between $\texttt{D}^2\texttt{IA}$ and SQL Match_Recognize within the context of the upcoming research trend on declarative stream processing using SQL.

Acknowledgement

The work of Ahmed Awad, Samuele Langhi, Riccardo Tommasini, Mahmoud Kamel and Sherif Sakr is funded by the European Regional Development Funds via the Mobilitas Plus programme (grant MOBTT75).

References

[1] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. Efficient pattern matching over event streams. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 147–160, New York, NY, USA, 2008. ACM.

[2] J. F. Allen. Maintaining knowledge about temporal intervals. *Commun. ACM*, 26(11):832–843, Nov. 1983.

[3] D. Anicic, S. Rudolph, P. Fodor, and N. Stojanovic. Stream reasoning and complex event processing in ETALIS. *Semantic Web*, 3(4):397–407, 2012.

[4] A. Arasu, S. Babu, and J. Widom. The cql continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, 2006.

[5] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1383–1394, New York, NY, USA, 2015. ACM.

[6] A. Awad, R. Tommasini, M. Kamel, E. D. Valle, and S. Sakr. D ˆ2 2 IA: stream analytics on user-defined event intervals. In P. Giorgini and B. Weber, editors, *Advanced Information Systems Engineering - 31st International Conference, CAiSE 2019, Rome, Italy, June 3-7, 2019, Proceedings*, volume 11483 of *Lecture Notes in Computer Science*, pages 346–361. Springer, 2019.

[7] R. S. Barga, J. Goldstein, M. H. Ali, and M. Hong. Consistent streaming through time: A vision for event stream processing. In *CIDR*, pages 363–374, 2007.

[8] E. Begoli, T. Akidau, F. Hueske, J. Hyde, K. Knight, and K. Knowles. One sql to rule them all - an efficient and syntactically idiomatic approach to management of streams and tables. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, pages 1757–1772, New York, NY, USA, 2019. ACM.

[9] E. F. Codd. A database sublanguage founded on the relational calculus. In *Proceedings of the ACM-SIGFIDET Workshops*, 1971.

[10] G. Cugola and A. Margara. Low latency complex event processing on parallel hardware. *J. Parallel Distrib. Comput.*, 72(2):205–218, 2012.

[11] N. Dindar et al. Modeling the execution semantics of stream processing engines with secret. *VLDB J.*, 22(4), 2013.

[12] O. Etzion and P. Niblett. *Event Processing in Action*. Manning, 2010.

[13] B. Gedik. Generic windowing support for extensible stream processing systems. *Software: Practice and Experience*, 44(9):1105–1128, 2013.

[14] K. Georgala, M. A. Sherif, and A. N. Ngomo. An efficient approach for the generation of allen relations. In *ECAI*, pages 948–956, 2016.

[15] M. Grossniklaus, D. Maier, J. Miller, S. Moorthy, and K. Tufte. Frames: Data-driven windows. In *DEBS*, pages 13–24. ACM, 2016.

[16] M. Hirzel, G. Baudart, A. Bonifati, E. D. Valle, S. Sakr, and A. Vlachou. Stream processing languages in the big data era. *SIGMOD Record*, 47(2):29, 2018.

[17] H. Jafarpour and R. Desai. KSQL: streaming SQL engine for apache kafka. In *Advances in Database Technology - 22nd International Conference on Extending Database Technology, EDBT 2019, Lisbon, Portugal, March 26-29, 2019*, pages 524–533. OpenProceedings.org, 2019.

[18] M. Körber, N. Glombiewski, and B. Seeger. TPStream: Low-Latency Temporal Pattern Matching on Event Streams. In *EDBT*, pages 313–324, 2018.

[19] M. Li, M. Mani, E. A. Rundensteiner, and T. Lin. Complex event pattern detection over streams with interval-based temporal semantics. In *DEBS*, 2011.

[20] F. McSherry, M. Isard, and D. G. Murray. Scalability! but at what {COST}? In *15th Workshop on Hot Topics in Operating Systems (HotOS {XV})*, 2015.

[21] B. Nebel and H. Bürckert. Reasoning about temporal relations: A maximal tractable subclass of allen's interval algebra. *J. ACM*, 42(1):43–66, 1995.

[22] A. Paschke. Eca-ruleml: An approach combining ECA rules with temporal interval-based KR event/action logics and transactional update logics. *CoRR*, 2006.

[23] M. Pathirage, J. Hyde, Y. Pan, and B. Plale. Samzasql: Scalable fast data management with streaming sql. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1627–1636, May 2016.

[24] M. B. Vilain and H. A. Kautz. Constraint propagation algorithms for temporal reasoning. In *Proceedings of the 5th National Conference on Artificial Intelligence.*, pages 377–382, 1986.

Appendix A. EPL Translations for Evaluation Queries

Listing Appendix A.1: EPL Translation for Q1

```
create context SegmentedByKey
partition by key from SpeedEvent;
@name('ThresholdAbsolute')
context SegmentedByKey
select * from SpeedEvent
match_recognize (
partition by key
measures first(A.timestamp) as first_ts,
last(A.timestamp) as last_ts, max(A.value) as value,
count(A.value) as count, first(A.key) as key
after match skip past last row
pattern (A{2,})
interval 10 seconds or terminated
define
    A as A.value >= 50
);
```

Listing Appendix A.2: EPL Translation for Q2

```
create context SegmentedByKey
partition by key from SpeedEvent;
@name('ThresholdRelative')
context SegmentedByKey
select * from SpeedEvent
match_recognize (
partition by key
measures max(max(A.value),C.value) as value,
first(A.key) as key, last(A.timestamp) as last_ts,
C.timestamp as first_ts, count(A.value) as counter
after match skip past last row
pattern (C A+)
interval 5000 seconds or terminated
define
      C as C.value >= 30,
      A as (A.value > C.value * 0.1)
      and (A.value > prev(A.value,1) +
            (prev(A.value,1)*0.1))
)
```

Listing Appendix A.3: EPL Translation for Q3

```
create schema ResultStream (value double,
first_ts long, last_ts long, key string)
starttimestamp first_ts endtimestamp last_ts;
create context SegmentedByKey
partition by key from SpeedEvent,
key from ResultStream;
context SegmentedByKey
create variable long last_time = 0L;
context SegmentedByKey insert into ResultStream
```

```
10 select * from SpeedEvent#expr(timestamp>last_time)
11 match_recognize (
12 partition by key
13 measures
14 (sum(A.value)+C.value)/(count(A.value)+1) as value,
15 C.timestamp as first_ts, last(A.timestamp) as last_ts,
16 first(A.key) as key
17 after match skip past last row
18 pattern (C A+)
19 interval 5000 seconds or terminated
20 define
21       A as (Math.abs(A.value - C.value) >= 5)
22 )
23 context SegmentedByKey
24 on ResultStream(first_ts>last_time)
25 set last_time = last_ts;
26 @name('Delta') select * from ResultStream;
```

Listing Appendix A.4: EPL Translation for Q4

```
1 create context SegmentedByKey
2 partition by key from SpeedEvent;
3 context SegmentedByKey
4 insert into ResultStream
5 select avg(value) as value,
6 first(timestamp) as first_ts,
7 last(timestamp) as last_ts,
8 key as key, count(*) as counter
9 from SpeedEvent#expr_batch(avg(value) < 67.0,false);
10 @name('Aggregate') select value,
11 first_ts, last_ts, key
12 from ResultStream(value>=67.0 and counter>=2);
```