The Event Processing Language for Streaming Data

Samuele Langhi
Lyon 1 University
Lyon, France
samuele.langhi@univ-lyon1.fr

Riccardo Tommasini
INSA Lyon
Lyon, France
riccardo.tommasini@insa-lyon.fr

Angela Bonifati
Lyon 1 University
Lyon, France
angela.bonifati@univ-lyon1.fr

Thomas Bernhardt
EsperTech
New Jersey, USA
tom@espertech.com

ABSTRACT

The Event Processing Language (EPL) is one of the oldest and most established languages in the stream processing landscape, leveraging SQL as a backbone. In addition, EPL is very expressive, combining features for analytical continuous queries and complex event recognition. Although the language has multiple implementations and is used in various application domains, the absence of underlying formal semantics leads to erroneous behaviour and ambiguity of the language constructs. This paper focuses on formalizing a self-contained subset of the EPL's Data Definition Language and Data Modification Language. The DDL allows specifying event schema hierarchies, while the DML supports continuous query writing through windowing and EPL Patterns. The role of our formalization is twofold: it leads to addressing the semantic ambiguity of the language, and it allows identifying possible optimizations across the various constructs. Our work will benefit users, practitioners, and researchers interested in standardization efforts for this relevant class of streaming query languages.

KEYWORDS

epl, esper, streaming, cep, json

1 INTRODUCTION

The Event Processing Language (EPL) is one of the oldest and most mature query language in stream processing (SP), i.e., the area of databases that aims at analyzing data streams in real time. While SP has been around for two decades [5, 12], EPL's development started in 2004, leading to the first language release published in 2006. In almost 20 years of stable development, EPL has recently got new attention given the growing popularity of declarative languages for SP [19, 30].

The language and its reference implementation Esper were first created to monitor financial trades. Still, their use spread across various domains, such as transportation, logistics, flight monitoring, telecommunication, and security, to name a few [1]. Moreover, the project was open-sourced from the start of the development, which fostered the growth of the language alongside its user base.

In the SP landscape, EPL provides a unique combination of features, and it is the only language combining CEP and streaming analytics with SQL-like syntax. EPL includes various features beyond standard SQL constructs that are convenient in several use cases, i.e., reporting control, context partitioning, and advanced selection policies. In practice, the language exhibits high expressiveness yet it lacks a formal semantics.

Our work addresses the formalization of the core fragment of EPL. Together with Espertech, we identified the most relevant subset of the language having practical interest. To do so, we studied the documentation, selected EPL query logs containing hundreds of anonymized queries[2], and over 40 research papers adopting EPL/Esper as subjects of study. More specifically, we focus on the following EPL's core features: (i) EPL's Data Definition Language (DDL), which allows specifying event schemas and their hierarchies; (ii) EPL's Data Manipulation Language (DML), which simultaneously supports window-based continuous queries and complex event processing. Our formalization efforts have several benefits in the development of EPL: (i) they lead to resolving semantic ambiguities of the language: For example, the **NOT** operator, which is meant to specify guard expressions based on a specific event schema, does not have a well-defined semantics when used outside such context, and may lead to missing answers . (ii) They allow the definition of equivalences in EPL continuous queries and, thus, open the doors to additional query optimization. For example, under certain conditions, the two alternatives for expressing selections (i.e., **WHERE** and patterns filters) are equivalent. (iii) They support the identification of potentially harmful constructs. For instance, an event pattern abusing of the **EVERY** construct can lead to major redundancies in the resulting stream with a memory overload side effect. To summarize, our contribution is two-fold:

- the specification of EPL's data model which consists of hierarchical, object-based entities;
- the syntax of EPL operators and their formal semantics, covering EPL expressions and patterns.

Outline. Section 2 introduces EPL and presents a running example that will help us alongside the entire manuscript. Section 3 provides the formal underpinnings of the EPL data model. Section 4 introduces the details of the formalization. Section 5 presents the related work. Finally, Section 6 draws conclusions discussing future work.

2 THE EVENT PROCESSING LANGUAGE

This section presents an overview of EPL's language features: EPL's Data Definition Language (DDL) allows specifying events schemas that, in turn, constrain the structure of the data streams; EPL's Data Manipulation Language (DML) allows writing continuous queries using window-based aggregation and temporal

[1]https://www.espertech.com/customers/

[2]https://raw.githubusercontent.com/semlanghi/kEPLrbook/master/query-logs-analysis.png

Schema	TR	TR	TR	MN	TR	MN	MN	TR	SM	TR	MN	TR	E	...	TR	TR
`therm`	"R1"	"R2"	"R1"		"R2"			"R1"		"R2"		"R1"		...	"R1"	"R2"
`camera`				"R2"		"R2"	"R1"				"R2"		"R2"	...		
`sensor`									"R2"					...		
`xCoor`				3		3	3				3			...		
`yCoor`				2		2	2				2			...		
`temp`	21	20	21		21			21		23		21		...	21	45
`humid`	40	42	39		38			42		35		40		...	40	16
`log`													"500"	...		
Time (seconds)	0	10	20	25	30	31	32	40	45	50	53	60	65	...	280	290

Table 1: Sample stream S, with events originating from a set of cameras, thermometers, and smoke detectors connected to a smart home ecosystem. [T]herm[R]ead, [M]oveme[N]t, [E]error, [SM]oke

operators. To better explain EPL's nuances, we introduce a running example: we consider a stream S, an example of which is shown in Table 1, containing the events coming from multiple devices connected to a smart-home ecosystem, i.e., cameras, thermometers, and smoke sensors. These sensors are placed in two house rooms labeled "R1" and "R2".

EPL is an Object-Oriented language that borrows most of its syntax from SQL. To define *events*, EPL's DDL borrows the notion of *schema* from the relational word and adds the *stream* concept. A stream is an infinite sequence of partially ordered events with the same schema delineating its structure. Event schemas are declared through the **CREATE SCHEMA** notation, which is followed by a list of attribute-types mappings. The schema references all the incoming events with that specific schema. Therefore, it acts as a filter on the heterogeneous input stream. EPL's DDL supports **polymorphism**, i.e., an event can adhere to multiple schemas. Indeed, a schema can *inherit* another schema. Consequently, all the events that comply with the child schema also adhere to the parent schema. In practice, the child schema retains all the attributes of the parent schema and can extend it by adding other attributes. Listing 1 shows the definition of the event schemas related to the running example. The camera can detect movements inside its range of vision (`Movement`), while the thermometer registers the temperature and humidity in the room every 20 seconds (`ThermRead`). Additionally, smoke sensors send a notification once they sense smoke (`Smoke`). Two examples of polymorphism are shown in lines 3 and 5.

```
1 CREATE SCHEMA CameraEvent (camera string);
2 CREATE SCHEMA Movement (xCoor int, yCoor int)
3     INHERITS CameraEvent;
4 CREATE SCHEMA Error (log string)
5     INHERITS CameraEvent;
6 CREATE SCHEMA Smoke (sensor string);
7 CREATE SCHEMA ThermRead (therm string,
8     temp double, humid double);
```

Listing 1: EPL DDL: Event Schema Creation.

EPL's DML allows writing continuous queries to evaluate over input streams [5]. EPL queries reference the input streams via their schema name. Events are processed in the order of arrival, and each is mapped to a given processing time by the system. EPL's DML syntax is similar to SQL and includes classic aggregates, e.g., count. However, unlike relational databases, the result of an EPL's DML query is updated as new events arrive. Listing 2 shows a straightforward query counting all the events from the `Error` stream.

```
SELECT COUNT(*) FROM Error
```

Listing 2: EPL DML: Counting Errors

By default, a query operates over the whole stream, considering all the events received so far. However, this may pose significant performance drawbacks given the infinite nature of the streams. EPL's DML allows limiting the query scope using *window operators*, which divide the stream into finite portions, over which other operators can compute aggregations [12]. The finite portion of the stream can be treated as a table allowing stateful aggregation and joins, as shown in the listing below. EPL supports multiple windowing policies, although in this work we consider only time windowing for lack of space. Listing 3 shows a query for detecting possible fire occurrences. Since the first sign of fire is the smoke, followed by a significant rise in the room's temperature, the query joins smoke detections and high-temperature registrations in the last 5 minutes. If evaluated against the sample stream S, this continuous query returns a stream of a single event, reported in Table 2. The resulting event includes all the attributes from the two joined events and has the earliest event's timestamp.

```
SELECT *
FROM Smoke#win:time(5 min),
     ThermRead#win:time(5 min)
WHERE sensor=therm AND temp>40 AND humid<20;
```

Listing 3: EPL DML: Analytic Query for detecting possible fires within a room.

A unique feature of EPL's DML is its pattern sub-language. **EPL patterns** leverage time-aware operators for combining multiple schemas, and the results are new, composite events. Such events can then be used outside the pattern definition for projection, filtering, and joins.

```
SELECT *
FROM PATTERN
    [EVERY (x=Movement OR y=Smoke) →
            z=ThermRead(temp>40 AND humid<20)
            where timer:within(5 minutes)]
WHERE x.camera=z.therm OR y.sensor=z.therm;
```

Listing 4: The simplest EPL query version 2.

EPL Patterns are embedded in EPL's DML using the **FROM PATTERN** clause. They present multiple ways of defining events' correlations. The *followed-by* operator (⟶) detects an event followed by another with respect to time. Logical operators like **OR** and **AND** are truth-based operators, they treat the underlining patterns as logic expressions that turn true once they are

Schema	`therm`	`sensor`	`temp`	`humid`	Time
SM ∪ TE	"R2"	"R2"	45	16	290

Table 2: The result of the query in Listing 3

Schema	x:MN,z:TR	x:MN,z:TR	y:SM,z:TR	x:MN,z:TR
x	⟨"R2",3,2⟩	⟨"R2",3,2⟩		⟨"R2",3,2⟩
y			⟨"R2"⟩	
z	⟨"R2",45,16⟩	⟨"R2",45,16⟩	⟨"R2",45,16⟩	⟨"R2",45,16⟩
Time	290	290	290	290

Table 3: The result of the query in Listing 4.

completed. Moreover, the **NOT** checks the non-occurrence of an event. All these operators can be combined with the **EVERY** to generate multiple variants of the same pattern.

EPL patterns further raise the expressivity of stream processing queries. For instance, the query in Listing 3, although capable of detecting possible fires, it does not consider a dangerous corner case: if the fire does not produce smoke immediately (e.g., from a gas leak), the detection does not happen immediately. In practice, if we remove the smoke event at time 45 from stream S, the fire detection event is not produced[3]. To prevent this scenario, we can exploit the presence of cameras and optionally consider movements in the camera's field of vision as potential signs of fire. The query in Listing 4 uses EPL Patterns to account for this scenario, detecting either movements *or* smoke followed by a significant temperature event.

In this case, the schema name is used in combination with a filter expression, whose semantics are slightly different than the **WHERE**, as we describe in further sections. Additionally, the query outputs multiple events. This redundancy is caused by the presence of the **EVERY** and **OR**. Their combination generates multiple searching instances of the pattern for each movement or smoke detection. All these instances are completed when a significant temperature reading arrives. In Table 3, we show the result of the CEP query. The resulting complex events have a nested format since the original events are wrapped within predefined variables, i.e., x, y, and z.

Like other EPL queries, by default, EPL patterns search for events across the entire input stream. This behaviour can be limited using guards, e.g., **time:within**. Variables are mapped to matched events as the pattern continues its search until completion. The pattern maps each variable to only one event, i.e., the first to arrive since the pattern activation. This first-oriented approach reiterated through the **EVERY**, represents the base recognition step.

3 EPL DATA MODEL

EPL's data model revolves around the concept of *event*, i.e., a set of key-value pairs, where values may be of arbitrary types, including nested events. Such a model resembles those of object-oriented and document databases. Considering the recent formalisation by Bourhis et al [8], JSON Objects are structurally equivalent to EPL's events. However, *JSON Objects are schema-less and time agnostic.* In the following, we present the EPL data model formalisation that we built on top of the JSON model formalisation [8] introducing both the notion of event schema and temporal awareness. Table 4 summarises the mappings between the two data models, highlighting the correspondences between events, objects, types, schema, and features.

In the JSON Data Model, JSON values are inductively defined as either *strings*, *numbers*, *JSON objects*, and *JSON arrays*. Without loss of generality, we can consider the same types for EPL events. More formally, given the set of all Unicode characters Σ, a string

[3]https://github.com/semlanghi/kEPLrbook/blob/master/jupyter_notebooks/Listing_3_without_smoke.ipynb

EPL Data Model		JSON Data Model [8]
Events	$\subset$	JSON Objects
Strings	$\equiv$	Strings
Arrays	$\equiv$	JSON Arrays
Numbers	$\subset$	Numbers
Event schemas	nd	nd
Time Awareness	nd	nd

Table 4: Mappings between EPL and JSON.

s is an element of Σ^*. Given a set of distinct strings $s_1, ..., s_n$ and a set of JSON values $v_1, ..., v_n$, we call *JSON object* a one-to-one mapping of the form $\{s_1 : v_1, ...s_n : v_n\}$, where each element $s_i : v_i$ of the object is a *key-value pair.* On the other hand, a JSON array $[v_1, v_2, ..., v_n]$ is an ordered sequence of JSON values indexed on their position. Being themselves JSON values, objects and arrays can be arbitrarily nested within each other. We indicate with $\mathbb{V}$ the set of all JSON values.

Given the presence of arbitrary nesting, we introduce a notation for accessing values inside objects and arrays. In particular, given an object o, $o.key$ represents the value v such that $(key : v) \in o$. Moreover, given an array a, $a[i]$ stands for i-th value of the array v_i, i.e., $a = [..., v_i, ...]$. The main advantage of this notation is the capacity of accessing values across different nesting levels. For instance, for accessing v_i in the object $o = \{..., key : [..., v_i, ...]\}$, it is sufficient to write $o.key[i]$.

Extended Event-based Semantics. Events are JSON Objects that adhere to a given *event schema.* Since JSON objects are themselves JSON values, EPL events are considered JSON values too. An event schema is a mapping between string and sets of JSON values. The mapped strings are called *attributes*, while the sets of JSON values represent the values' domains for events that adhere to that schema. Moreover, we distinguish between *simple attributes*, which are mapped to nested events, numerical values, or strings, and *group attributes*, which we map to arrays.

Definition 3.1. Given the sets of JSON values $\mathbb{V}_1, \mathbb{V}_2, ...\mathbb{V}_n$ and the distinct attributes $s_1, ..., s_n$, an event schema is the mapping defined as $R = \{s_1 : \mathbb{V}_1, ..., s_n : \mathbb{V}_n\}$. Moreover, we define as $events(R)$ all the set of events that adhere to R. More formally,

$$e \in events(R) \leftrightarrow e \supseteq \{s_1 : v_1, ...s_n : v_n, ..., s_m : v_m\} \text{ where } v_i \in \mathbb{V}_i$$

Additionally, two schemas R_1, R_2 (and the adhering events) are said *compatible*, if they are disjoint, i.e. $R_1 \cap R_2 = \emptyset$.

We define the adherence of an event e to a given schema R through the subset relation ($\subseteq$) to allow event polymorphism (vid. Section 2). Indeed, given two schemas R_1, R_2, we say that R_1 *inherits* R_2 iff $R_1 \subset R_2$. In this context, we consider the empty schema ϵ, which corresponds to an empty mapping. Notably, $events(\epsilon)$ is the set of all events, since for any R, $\epsilon \subset R$.

Another major aspect of the EPL data model is the concept of time and order. In order to abstract the concept of time assigned by the system, we introduce the *timestamp function*, which given an event of a generic type returns a timestamp, i.e., a natural number.

$$t : events(\epsilon) \longrightarrow \mathbb{N}$$

EPL supports event contemporaneity. Indeed, the timestamp function may map two events to the same timestamp. Moreover, the timestamp function is globally defined on the set of events, which is necessary for defining time-dependent EPL Patterns across multiple types.

$$
\begin{aligned}
\gamma &: \alpha\ \texttt{FROM}\ \beta \ \mid\ \alpha\ \texttt{FROM}\ \beta\ \texttt{GROUP BY}\ \Delta \\
&\mid\ \alpha\ \texttt{FROM}\ \beta\ \texttt{GROUP BY}\ \Delta\ \texttt{HAVING}\ acond \\
\alpha &: \texttt{SELECT}\ (\Delta, AGG) \\
\beta &: \iota\ \texttt{WHERE}\ cond\ \mid\ \iota \\
cond &: P(\delta_1, ..., \delta_k)\ \mid\ cond\ \texttt{AND}\ cond \\
&\mid\ cond\ \texttt{OR}\ cond\ \mid\ \texttt{NOT}\ cond \\
acond &: P(\delta_1, ..., agg_1, ...)\ \mid\ acond\ \texttt{AND}\ acond \\
&\mid\ acond\ \texttt{OR}\ acond\ \mid\ \texttt{NOT}\ acond \\
\iota &: \omega\ \mid\ \omega\texttt{,}\omega\ \mid \omega\ \texttt{JOIN}\ \omega\ \texttt{ON}\ cond \mid\ \xi \\
\omega &: \xi\texttt{\#win:}f\ \mid \xi\texttt{\#win:}f\ \texttt{AS}\ x \\
\xi &: R\ \mid\ R(cond)\ \mid\ \texttt{PATTERN}\ [\rho] \\
\rho &: \rho''' \mid \rho' \texttt{->} \rho'\ \mid\ \rho'\ \texttt{AND}\ \rho' \\
&\mid \texttt{EVERY}\ (\rho''' \texttt{->} \rho''') \mid \texttt{EVERY}\ (\rho'''\ \texttt{AND}\ \rho''') \\
\rho' &: \rho''\ \texttt{AND NOT}\ R \mid\ \rho''\ \texttt{where timer:within}(n) \mid \rho'' \\
\rho'' &: \texttt{EVERY}\ \rho''' \mid\ \rho''' \\
\rho''' &: x{=}R\ \mid x{=}R(cond)\ \mid\ \rho'''\ \texttt{OR}\ \rho'''
\end{aligned}
$$

Figure 1: The EPL Grammar.

Definition 3.2. The *time-ordering relation* $\leq_t$ is a partial ordering relation based on a given timestamp function t, where

$$a \leq_t b \leftrightarrow t(a) \leq t(b)$$

Given a set of events $\mathbb{E} \subset events(\epsilon)$, a timestamp function t, and the relative time ordering relation $\leq_t$, we call a *stream* a partially ordered set $S = (\mathbb{E}, \leq_t)$.

Example. *If we try to formalize the running example from Listing 1, we obtain the following events and event schemas with the related assigned timestamps.*

$$
\begin{aligned}
&CE = \{camera : \Sigma^*\} \quad SM = \{sensor : \Sigma^*\} \\
&MN = \{camera : \Sigma^*, xCoor : \mathbb{R}, yCoor : \mathbb{R}\} \\
&E = \{camera : \Sigma^*, log : \Sigma^*\} \\
&TR = \{therm : \Sigma^*, temp : \mathbb{R}, humid : \mathbb{R}\} \\
&e_1 = \{camera : "R2", xCoor : 3, yCoor : 2\} \\
&e_2 = \{therm : "R1", temp : 21, humid : 40\} \\
&e_3 = \{camera : "R2", log : "500"\} \quad e_4 = \{sensor : "R2"\} \\
&t(e_1) = 25 \quad t(e_2) = 0 \quad t(e_3) = 65 \quad t(e_4) = 45
\end{aligned}
$$

Notably, e_1 adheres to MN, and e_3 adheres to E. Additionally, both e_1 and e_3 have schema CE ($e_1, e_3 \in events(CE)$) since schema CE is inherited by E and MN ($CE \subset E$ and $CE \subset MN$). All events e_1, e_2, e_3 adhere to the empty schema ϵ ($e_1, e_2, e_3 \in events(\epsilon)$). Last but not least, the set $S = \{e_2, e_1, e_4, e_3\}$ is a stream since it is ordered according to t.

Name	Notation	Definition
-	$\mathbb{ET}_{>n}$	$\{n' \mid n' \in \mathbb{ET} \wedge n' > n\}$
-	$\mathbb{ET}_{<n}$	$\{n' \mid n' \in \mathbb{ET} \wedge n' < n\}$
-	S_R	$events(R) \cap S$
support	$supp^{S,\mathbb{ET}}(\rho)$	$\{n \mid \exists e \in [\![\rho]\!]_{S,\mathbb{ET}}, \wedge t(e) = n\}$
merge	$S_1 \uplus S_2$	$\{e_1 \cup e_2 \mid e_1 \in S_1,\ e_2 \in S_2\ compat.\}$

Table 5: Summary of used notations.

4 EPL SYNTAX & SEMANTICS

In this section, we present EPL syntax and semantics. EPL contains multiple constructs combined according to the grammar in Figure 1. The grammar limits the construct combination, representing a safe guideline for writing EPL queries that avoids major ambiguities and problems, e.g., arbitrary nesting of **EVERY** or misuse of the **NOT**. These constructs are categorized according to their evaluation: (1) *Terms* are either strings, numbers, or math expressions, they are evaluated against an event and may return simple values, objects, or arrays (2) *Aggregates* are functions evaluated against a stream and an event, returning an aggregated value (3) *Conditions* are logic-based constructs, they are evaluated against an event and a stream, returning a boolean value, i.e., *true* or *false* (4) *Expressions* deal with the manipulation of events, they are evaluated against a stream and return another stream (5) *Patterns* deal with patterns recognition, they are evaluated on a stream of events and return another stream of events

In EPL, a *query* is a composition of language constructs of the form **SELECT-FROM**. The query evaluation is abstracted by the evaluation function, denoted as $[\![\,]\!]$, which is recursively applied across the query's syntax tree. In the remainder of the section, we will walk through the constructs and their evaluation results[4]. All the constructs and queries presented in this paper can be executed on a custom Jupyter environment available on GitHub[5].

Terms are used for selecting and comparing events' values. They are expressed in the form of either JSON values ($v \in \mathbb{V}$), event attributes, or mathematical expressions ($y : \mathbb{N}^n \to \mathbb{N}$). The evaluation of a term δ against an event e ($[\![\cdot]\!]_e$) returns a value, i.e., a numeric value, a JSON object, or an array. Like SQL, the $\star$ is a special term that instructs the query to return the entire event. In the rest of the paper, we refer to Δ as a list of terms δ_i.

$$
[\![\delta]\!]_e = \begin{cases}
e & \delta = \star \\
v & (\delta = x \wedge e.x = v) \vee (\delta = x[n] \wedge e.x[n] = v) \\
 & \vee\ (\delta = x.lastOf() \wedge e.x = [..., v]) \\
 & \vee\ (\delta = x.firstOf() \wedge e.x = [v, ...]) \\
 & \vee\ (\delta = \delta_1.\delta_2 \wedge [\![\delta_2]\!]_{e'} \text{ where } e' = [\![\delta_1]\!]_e) \\
\delta & \delta \in \mathbb{N} \vee (\delta = "x" \wedge x \in \Sigma^*) \\
y([\![\Delta]\!]_e) & \delta = f(\Delta) \\
\emptyset & otherwise
\end{cases}
$$

Aggregations are functions that, given a stream S and an event, return a value ($[\![\cdot]\!]_{S,e}$). Aggregations are based on an attribute and a set of parameters. If the attribute is a simple attribute, the aggregation function is evaluated against the entire stream.

$$
[\![avg(att)]\!]_{S,e} = \begin{cases}
(e.att[1], ..., e.att[m])/n & if\ e.att \text{ is an array} \\
(e_1.att + ... + e_n.att)/n & otherwise
\end{cases}
$$

Conditions are collections of EPL predicates, i.e., relations between terms (and aggregates) based on mathematical operators ($=, <, > \ldots$). There can be two types of conditions. Simple conditions (*cond*) can only be composed of terms and are evaluated against a single event ($[\![\cdot]\!]_e$). In the following, we define the

[4]For the sake of clarity, we will indicate just the events inside the resulting stream. The default ordering relation is $\leq_t$.
[5]https://github.com/semlanghi/kEPLrbook

evaluation of a simple condition *cond* against an event *e*.

$$[\![P(\delta_1, ..., \delta_k)]\!]_e = \begin{cases} true & P([\![\delta_1]\!]_e, ..., [\![\delta_k]\!]_e) \; holds \\ false & otherwise \end{cases}$$

$$[\![cond_1 \;\; \textbf{AND} \;\; cond_2]\!]_e = [\![cond_1]\!]_e \wedge [\![cond_2]\!]_e$$
$$[\![cond_1 \;\; \textbf{OR} \;\; cond_2]\!]_e = [\![cond_1]\!]_e \vee [\![cond_2]\!]_e$$
$$[\![\textbf{NOT} \;\; cond]\!]_e = \neg[\![cond]\!]_e$$

On the other hand, aggregate conditions (*acond*) may contain aggregations. Their evaluation takes as input an additional stream S ($[\![\cdot]\!]_{S,e}$). In the following, we only show the evaluation of aggregate conditions' predicates since it remains equivalent to simple conditions for logical operators (**AND, OR, NOT**).

$$[\![P(\delta_1, ..., agg_1, ...)]\!]_{S,e} = \begin{cases} true & if \; P([\![\delta_1]\!]_e, ..., [\![agg_1]\!]_{S,e}, ...) \\ false & otherwise \end{cases}$$

Expressions abstract the semantics of EPL relational operators. Thus, they include *selection*, *projection*, *grouping*, *joins* and *windows*. Their semantics of the operators are reported in Table 6 and are highly dependent on *when* the evaluation happens. Thus, we introduce the set of evaluation time instants $\mathbb{ET} \subset \mathbb{N}$, representing when the evaluation is performed. An expression is evaluated against $\mathbb{ET}$ and a stream $S = (\mathbb{E}, \leq_t)$ ($[\![\cdot]\!]_{S,\mathbb{ET}}$). Additionally, we provide in Table 5 a set of preliminary notations to improve readability.

We start our description with the **FROM** construct, which iterates over the stream *evaluating the right-side expression* incrementally, feeding the **SELECT** with each result. This approach enables the replication of real-time execution. The **SELECT** *projects* a list of n terms $\Delta = (\delta_1, ..., \delta_n)$ against the last event arrived, and a list of m aggregations $AGG = (agg_1, ..., agg_m)$ against the optionally windowed stream.

On the other hand, the **FROM**'s right-side expression defines the query's input stream. The expression can be a schema name, an EPL pattern, a named window, or a table name. This work concentrates only on the former two cases, leaving the other two as future works. A schema name selects events adhering to the referenced schema, while an EPL pattern composes an input stream of *complex events*, i.e., events that contain other events. Moreover, **WHERE** represents a *selection* operation based on a condition, which determines if an input event is valid or not. If present, the **GROUP BY** enables grouped aggregation over the matched events according to a list of attributes, which is validated by the **HAVING**, similarly to SQL. The input stream may also be windowed according to a policy. The policy is abstracted by a *windowing function* f introduced by a special notation (**#win:**). A windowing function is a function that, given a stream S and timestamp *tau*, returns a stream S' such that $S' \subset S$. There can be different windowing policies [10], but in this work, we focus on time-based windows with size σ and hop step h:

$$time^{\sigma,h}(S, \tau) = \begin{cases} \{e \mid e \in S \;\wedge\; t(e) \geq (\tau - \sigma)\} & if \; \tau \bmod h = 0 \\ \emptyset & otherwise \end{cases}$$

Joins are performed between windowed streams. The operator's semantics is based on the *merge* operation (vid. Table 5), which merges two compatible events. To prevent naming conflicts and avoid event incompatibility (vid. Definition 3.1), the **AS** can assign a label to the events that are part of a window.

Example. *We rewrite the query from Listing 3 according to our formal notation (Q1). In particular, we consider the stream S from Table 1 and the following set of evaluation time instants, each one related to an event arrival:*

$$\mathbb{ET} = \{0, 10, 20, 25, 30, 31, 32, 40, 45, 50, 53, 60, 65, ..., 280, 290\}$$

$$\begin{aligned} [\![&\textbf{SELECT} * \textbf{FROM} \; SM\textbf{\#win:}time^{300,1}, \\ &TR\textbf{\#win:}time^{300,1} \; \textbf{WHERE} \; sensor = therm \\ &\textbf{AND} \; temp > 40 \; \textbf{AND} \; humid < 20]\!]_{S,\mathbb{ET}} \end{aligned} \tag{Q1}$$

*The query's semantics is based on the **FROM**, which enables an incremental evaluation over the set of time instants $\mathbb{ET}$. At each i-th iteration, it evaluates the underlining join operation on the set of time instants $\mathbb{ET}_{<n}$, with n being i-th oldest timestamp of set $\mathbb{ET}$ (cf. Table 5). As a result, the first iteration is based on $\mathbb{ET}_{\leq 0} = \{0\}$, the second on $\mathbb{ET}_{\leq 10} = \{0, 10\}$, and so on. At each iteration, both window functions are computed over $\max(\mathbb{ET}_{<n})$, resulting in a sliding movement that returns only events of the last 360 seconds (5 minutes). The first window (SM**#win:**$time^{300,1}$) gathers events that complies to schema SM ($e \in S_{SM}$). The second one (TR**#win:**$time^{300,1}$) returns events from schema TR ($e \in S_{TR}$). For instance, when we evaluate the windows on $\mathbb{ET}_{\leq 290}$, we obtain the following events*

$$\begin{aligned} &[\![SM\textbf{\#win:}time^{300,1}]\!]_{S,\mathbb{ET}_{\leq 290}} = \{sensor: ”R2”\} \\ &[\![TM\textbf{\#win:}time^{300,1}]\!]_{S,\mathbb{ET}_{\leq 290}} = \\ &\quad \{therm: ”R1”, temp: 21, humid: 40\}, \\ &\quad \{therm: ”R2”, temp: 20, humid: 42\}, ..., \\ &\quad \{therm: ”R2”, temp: 45, humid: 16\} \end{aligned}$$

Such events are then joined together. The streaming join can be seen as a cartesian product that merges ($\uplus$) events in both windows. The resulting events are the following

$$\begin{aligned} &[\![SM\textbf{\#win:}time^{300,1}, TM\textbf{\#win:}time^{300,1}]\!]_{S,\mathbb{ET}_{\leq 290}} = \\ &\quad \{sensor: ”R2”, therm: ”R1”, temp: 21, humid: 40\}, ..., \\ &\quad \{sensor: ”R2”, therm: ”R2”, temp: 45, humid: 16\} \end{aligned}$$

*Finally, events are filtered according to the **WHERE** condition, which in this case is satisfied by the last event, i.e., the result of the evaluation and the related query, as shown in Table 2.*

EPL Patterns deal with the composition and recognition of event patterns. Their semantics are shown in Table 6. The basic pattern x=R matches the first occurrence of an event of schema R, and maps it to a variable x, similarly to what the **AS** does. The **EVERY** iterates the subpattern's evaluation by consecutively cutting $\mathbb{ET}$ and restarting the evaluation from the last non-empty result of the sub-pattern, i.e., the maximum M of its support (vid. Table 5). Eventually, this recursive behavior of the operator creates a search-and-cut process in which (i) we search for the last event matched by the sub-pattern; then, (ii) we cut $\mathbb{ET}$ up to the found result ($\mathbb{ET}_{>M}$), and reiterate the evaluation. As the recursion progresses, the evaluation is performed on more recent time scopes due to the first-oriented approach of the recognition process. The recursion stops as soon as the underlining pattern returns an empty result.

The *Followed-By* (**→**) *merges* (vid. Table 5) each event returned by the left-side pattern with all the events returned by the right-side pattern, evaluated from the left-side result's timestamp. *Or* returns the first result from either one of the two sub-patterns as soon as they are triggered. *And* detects when both sub-patterns are satisfied regardless of the order of arrival. Thus, it is equivalent to the conjunction (**OR**) of two Followed-By patterns.

$$[\![\alpha\ \texttt{FROM}\ \beta]\!]_{S,\mathbb{ET}} = \bigcup_{n\in\mathbb{ET}} [\![\alpha]\!]_{S',\mathbb{ET}_{\le n}} \text{ s.t. } S' = [\![\beta]\!]_{S,\mathbb{ET}_{\le n}}$$

$$[\![\texttt{SELECT}\ (\Delta, AGG)]\!]_{S,\mathbb{ET}} = \{e \cup e_{AGG} \mid$$
$$\exists e' \in S, t(e') = max(\mathbb{ET}) \land e = e' |_{\Delta} \land$$
$$e_{AGG} = \bigcup_{1\le i\le m} (\text{“}agg_i(a_i)\text{”} : [\![agg_i(a_i)]\!]_{S,e'})\}$$

$$[\![\iota\ \texttt{WHERE}\ cond]\!]_{S,\mathbb{ET}} = \{e \mid e \in [\![\iota]\!]_{S,\mathbb{ET}} \land [\![cond]\!]_e = true\}$$

$$[\![\alpha\ \texttt{FROM}\ \beta\ \texttt{GROUP BY}\ \Delta]\!]_{S,\mathbb{ET}} = \bigcup_{n\in\mathbb{ET}, v_i\in\mathbb{V}} [\![\alpha]\!]_{S',\mathbb{ET}_{\le n}}$$
$$\text{where } S' = [\![\beta\ \texttt{WHERE}\ \bigwedge_{\delta_i\in\Delta} \delta_i = v_i]\!]_{S,\mathbb{ET}}$$

$$[\![\alpha\ \texttt{FROM}\ \beta\ \texttt{GROUP BY}\ \Delta\ \texttt{HAVING}\ acond]\!]_{S,\mathbb{ET}} = \bigcup_{n\in\mathbb{ET}, v_i\in\mathbb{V}} [\![\alpha]\!]_{S',\mathbb{ET}_{\le n}}$$
$$\text{where } S' = [\![\beta\ \texttt{WHERE}\ \bigwedge_{\delta_i\in\Delta} \delta_i = v_i]\!]_{S,\mathbb{ET}}$$
$$\land [\![acond]\!]_{S',e'} \land t(e') = max(\mathbb{ET})$$

$$[\![R]\!]_{S,\mathbb{ET}} = \{e \mid e \in S_R \land t(e) \in \mathbb{ET}\}$$

$$[\![R(cond)]\!]_{S,\mathbb{ET}} = [\![R\ \texttt{WHERE}\ cond]\!]_{S,\mathbb{ET}}$$

$$[\![\texttt{PATTERN}\ [\rho]]\!]_{S,\mathbb{ET}} = [\![\rho]\!]_{S,\mathbb{ET}}$$

$$[\![\xi\texttt{\#win:}f]\!]_{S,\mathbb{ET}} = f([\![\xi]\!]_{S,\mathbb{ET}}, \max(\mathbb{ET}))$$

$$[\![\xi\texttt{\#win:}f\ \texttt{AS}\ x]\!]_{S,\mathbb{ET}} = \{\{x : e\} \mid \forall e \in f([\![\xi]\!]_{S,\mathbb{ET}}, \max(\mathbb{ET}))\}$$

$$[\![\omega_1, \omega_2]\!]_{S,\mathbb{ET}} = [\![\omega_1]\!]_{S,\mathbb{ET}} \bowtie [\![\omega_2]\!]_{S,\mathbb{ET}}$$

$$[\![\omega_1\ \texttt{JOIN}\ \omega_2\ \texttt{ON}\ cond]\!]_{S,\mathbb{ET}} = [\![\omega_1, \omega_2\ \texttt{WHERE}\ cond]\!]_{S,\mathbb{ET}}$$

$$[\![x{=}R]\!]_{S,\mathbb{ET}} = \{\{x : e\} \mid \forall e \in \operatorname*{arg\,min}_{e\in S_R\land t(e)\in\mathbb{ET}} t(e)\}$$

$$[\![x{=}R(cond)]\!]_{S,\mathbb{ET}} = \{\{x : e\} \mid \forall e \in \operatorname*{arg\,min}_{e\in S_R\land t(e)\in\mathbb{ET}\land[\![cond]\!]_e = true} t(e)\}$$

$$[\![\texttt{EVERY}\ \rho]\!]_{S,\mathbb{ET}} = \begin{cases} [\![\rho]\!]_{S,\mathbb{ET}} \cup [\![\texttt{EVERY}\ \rho]\!]_{S,\mathbb{ET}_{>M}} & [\![\rho]\!]_{S,\mathbb{ET}} \neq \emptyset \\ \emptyset & otherwise \end{cases}$$

$$\text{where } M = \max(supp^{S,\mathbb{ET}}(\rho))$$

$$[\![\rho_1 \rightarrow \rho_2]\!]_{S,\mathbb{ET}} = \bigcup_{e\in[\![\rho_1]\!]_{S,\mathbb{ET}}} \{\{e\} \bowtie [\![\rho_2]\!]_{S,\mathbb{ET}_{>t(e)}}\}$$

$$[\![\rho_1\ \texttt{AND}\ \rho_2]\!]_{S,\mathbb{ET}} = [\![(\rho_1 \rightarrow \rho_2)\ \texttt{OR}\ (\rho_2 \rightarrow \rho_1)]\!]_{S,\mathbb{ET}}$$

$$[\![\rho_1\ \texttt{OR}\ \rho_2]\!]_{S,\mathbb{ET}} = \{c \mid \exists c \in \operatorname*{arg\,min}_{c'\in[\![\rho_1]\!]_{S,\mathbb{ET}}\cup[\![\rho_2]\!]_{S,\mathbb{ET}}} t(c')\}$$

$$[\![\rho\ \texttt{where timer:within}(n)]\!]_{S,\mathbb{ET}} = [\![\rho]\!]_{S,\mathbb{ET}_{\le m}}$$

$$[\![\rho_1\ \texttt{AND NOT}\ \rho_2]\!]_{S,\mathbb{ET}} = [\![\rho]\!]_{S,\mathbb{ET}_{\le m}} \text{ where } m = \min(supp^{S,\mathbb{ET}}(\rho_2))$$

Table 6: The semantics of both Expressions and Patterns.

Guards limit the pattern evaluation according to a specific condition. In particular, the time-based guard **timer:within** limits the evaluation to a subset of $\mathbb{ET}$. In practice, only the results coming within m time units from the start of the stream are retained. **AND NOT** is another *guard* construct that stops the evaluation as soon as an event with an event schema arrives.

In EPL, we can generate multiple variants of the same operator by adopting the **EVERY**. Here, we show the effect on the Followed-By, since it presents the most complex semantics out of all EPL patterns.

Example. *In Q2, we report a variant of the Sequence operator by rewriting the query from Listing 4. The* ***EVERY*** *finds all events from MN and SM followed by the first event TR that satisfies a set of criteria. Again, we consider the stream S from Table 1 and the following set of evaluation time instants, each one related to an event arrival:*

$$\mathbb{ET} = \{0, 10, 20, 25, 30, 31, 32, 40, 45, 50, 53, 60, 65, ..., 280, 290\}$$

$$\begin{aligned}
&[\![\texttt{SELECT} * \texttt{FROM PATTERN}\\
&[\texttt{EVERY}(x = MN\ \texttt{OR}\ y = SM) \rightarrow\\
&z = TR(temp > 40\ \texttt{AND}\ humid < 20)\\
&\texttt{where timer:within}(5\ mins)]\\
&\texttt{WHERE}\ x.camera = z.therm\\
&\texttt{OR}\ y.sensor = z.therm]\!]_{S,\mathbb{ET}}
\end{aligned} \tag{Q2}$$

In this case, the ***FROM*** *triggers an underlining pattern. The pattern is evaluated on incremental portions of* $\mathbb{ET}$. *For the sake of clarity, in the following evaluations, we will consider one incremental portion of* $\mathbb{ET}$, *which caps at time 290, and we will name it* $\mathbb{ET}'$ *(*$\mathbb{ET}' = \mathbb{ET}_{<290}$*)*

The pattern is based on the followed-by construct ($\rightarrow$*), which performs an iterative merge between results coming from two patterns, that we reference as left-side and right-side patterns, respectively. The followed-by evaluates the left-side pattern, and for each result, it evaluates the right-side pattern on future timestamps. In this case, the left-side pattern is an* ***EVERY****, whose evaluation is recursive and composed of two main steps:*

(1) *we search for the first movement or smoke detection over the current evaluation set, mapping it to either variable x or y, respectively. Considering the* ***FROM*** *right-side expression's evaluation on evaluation set* $\mathbb{ET}'$*, we obtain the following*

$$[\![x = MN\ \ \texttt{OR}\ \ y = SM]\!]_{S,\mathbb{ET}'} =$$
$$\{x : \{camera : \text{"}R2\text{"}, xCoor : 3, yCoor : 2\}\}$$

(2) *we evaluate recursively the* ***EVERY*** *over a subset of the original evaluation set that starts from the maximum time instant of the support of the underlining pattern, i.e., the timestamp of the matched events. Thus, since*

$$supp^{S,\mathbb{ET}'}(x = MN\ \ \texttt{OR}\ \ y = SM) = 25$$

the recursive evaluation of the ***EVERY*** *is performed starting from time instant 25 (excluded):*

$$[\![\texttt{EVERY}\ \ x = MN\ \ \texttt{OR}\ \ y = SM]\!]_{S,\mathbb{ET}'_{>25}}$$

This two-step process is recursively iterated until step 1 does not return an empty set, resulting in the following events:

$$\{x : \{camera : \text{"}R2\text{"}, xCoor : 3, yCoor : 2\}\}$$
$$\{x : \{camera : \text{"}R2\text{"}, xCoor : 3, yCoor : 2\}\}$$
$$\{x : \{camera : \text{"}R1\text{"}, xCoor : 3, yCoor : 2\}\}$$
$$\{y : \{sensor : \text{"}R2\text{"}\}\}$$
$$\{x : \{camera : \text{"}R2\text{"}, xCoor : 3, yCoor : 2\}\}$$

In practice, the **EVERY** *gathers all events arrived of type MN or SM. All these events are then evaluated into the followed by* ($\rightarrow$)*. For each event e that part of the left-side pattern's result, the followed-by evaluates* ρ_2 *based on timestamps greater than* $t(e)$*, i.e.,* $\mathbb{ET}_{>t(e)}$*. The results of these evaluations are merged with the related left-side pattern result:*

$$\{x : \{camera : "R2", xCoor : 3, yCoor : 2\}\} \uplus [\![\rho_2]\!]_{S,\mathbb{ET}'_{>25}}$$
$$\{x : \{camera : "R2", xCoor : 3, yCoor : 2\}\} \uplus [\![\rho_2]\!]_{S,\mathbb{ET}'_{>31}}$$
$$\{x : \{camera : "R1", xCoor : 3, yCoor : 2\}\} \uplus [\![\rho_2]\!]_{S,\mathbb{ET}'_{>32}}$$
$$\{y : \{sensor : "R2"\}\} \uplus [\![\rho_2]\!]_{S,\mathbb{ET}'_{>45}}$$
$$\{x : \{camera : "R2", xCoor : 3, yCoor : 2\}\} \uplus [\![\rho_2]\!]_{S,\mathbb{ET}'_{>53}}$$

The right-side pattern ρ_2 *searches for the first TR event* ($\arg\min t(e)$) *that satisfies a filter condition* ($temp > 40$ **AND** $humid < 50$)*. On top of this, the* **within** *construct limits the evaluation set with a maximum threshold. In this specific case, this limitation does not influence the evaluation result on stream S. In summary, the following evaluations are performed (cond = temp>40* **AND** *humid<20)*

$$[\![z{=}TR(cond)]\!]_{S,\mathbb{ET}'_{>25}} = [\![z{=}TR(cond)]\!]_{S,\mathbb{ET}'_{>31}} = \ldots$$
$$= \{z : \{therm : "R2", temp : 45, humid : 16\}\} = e_{right-side}$$

As a consequence, $e_{right-side}$ *is merged with all the events coming out of the left-side pattern. Finally, one last filtering step is performed, assuring the equality of room identifiers* ($x.camera = z.therm$ **OR** $y.sensor = z.therm$)*, resulting in the following merged events, reported in Table 3.*

$$\{x : \{camera : "R2", xCoor : 3, yCoor : 2\}\} \uplus e_{right-side}$$
$$\{x : \{camera : "R2", xCoor : 3, yCoor : 2\}\} \uplus e_{right-side}$$
$$\{y : \{sensor : "R2"\}\} \uplus e_{right-side}$$
$$\{x : \{camera : "R2", xCoor : 3, yCoor : 2\}\} \uplus e_{right-side}$$

5 RELATED WORK

This section positions our work in the Stream Processing state-of-the-art focusing, in particular, on declarative languages [19]. Notably, the declarative paradigm has emerged to simplify stream processing tasks. However, without broadly accepted standards, the resulting landscape is fragmented [19, 30].

Stream and Event Processing. Stream and event processing techniques have been around for decades [5, 12]. The most acknowledged organization of the state of the art builds on the distinction between Data Stream Management Systems (DSMS), and Complex Event Recognition Engines (CERE). Several formal frameworks have been presented by academia and industry for both [1, 13, 15, 16, 20, 27]. On the DSMS side, the seminal work of Arasu et al. is a commonly accepted formalization for DSMS languages. Continuous Query Language (CQL) extends relational algebra with streams and operators to cope with their unboundedness [1]. On the CERE side, Chakravarthy et al. [11] designed SNOOP, a specification language that allows active databases to detect events.

Language Calculi. Within event stream processing, they aim at abstracting a unified semantics that capture different languages and systems. Among these solutions, Brooklet [27] is a stream calculus capable of modeling multiple stream processing languages, e.g., CQL, StreamIt [29], and Sawzall [17]. A Brooklet program defines a graph of operators and can capture non-determinism by avoiding the queues micro-management, covering all the possible outcomes. Another significant contribution is the Event Calculus (EC), a powerful logic programming formalism for reasoning about events. EC adopts functional statements and the special *holdsAt* predicate to assign temporality to them. It can be used for multiple purposes, e.g., complex event recognition [4].

Streaming SQL. The initial vision for a Streaming SQL standard dates back around 2008 [23, 28]. Stonebraker et al. [28] discussed the requirements analysis for an industrial-grade stream processing engine to satisfy. Among other timely proposals, the adoption of SQL as a language for real-time data management is highly relevant. On a similar note, Jain et al. combined and discuss different proposals from the industry that may converge into a so-called standard. Recently, the discussion reopened thanks to a renovated proposal: Bengoli et al. [6] present the opportunities and challenges for a streaming-first SQL standard that builds upon the lessons learned from the rise of Big Data SP systems. In these regards, we summarize the most prominent ones in the literature. For a more comprehensive survey that extends to generic declarative languages, we point the reader to [19].

Apache Calcite [7] is a complete query processing system that provides many standard functionalities for query parsing, optimization, and execution. Calcite supports streaming operators based on `WINDOW`s as a form of advanced grouping condition. Calcite is a popular choice for streaming systems too. Projects such as Apache Apex [18], Flink, Apache Samza [9], and Storm [21], and BEAM [24] have chosen to implement their streaming SQL extensions using Calcite components. In particular, it is worth noticing Flink's support for advanced windows (e.g., Sessions) and `MATCH RECOGNIZE`, whose semantics relies on regular expressions and is part of the SQL standard. Both features are also supported in EPL, although not included in our analysis.

KSQL-DB [22] is a streaming SQL engine implemented on top of the Kafka Streams API [26]. In KSQL, a `TABLE` is the result of the compaction of a key-based append-only log (Topic). A Table is opposed to the `STREAM` abstraction, which directly maps into Kafka topics. To move between these abstractions, KSQL extends SQL with windowed joins, aggregations, and filters.

SparkSQL [3] is a Spark module for structured data processing. It was recently extended to perform relational operations over unbounded Dataframes [2]. The SQL extension is embedded into Scala code and supports projections, selection, and window-based joins and aggregations.

6 CONCLUSIONS

The Event Processing Language (EPL) stands out as a well-known stream processing language adopted in numerous applications since 2004 [14, 25, 31]. Moreover, EPL features constructs touching upon more recent versions of SQL and beyond, e.g., complex event recognition and report controlling. EsperTech maintains the language's documentation and open-source implementation since 2004, which considers the evolving user requests.

Our work addresses the formalization of a core fragment of EPL identified thanks to real-world anonymized query logs courtesy of EsperTech. As future work, we plan to extend our formal analysis across other subsets of EPL, e.g., match-recognize, contexts, tables, and the related delete/update commands. Our analysis paves the way to detect semantic ambiguities that can guide users and practitioners through optimizations and performance improvements for future versions of the language.

REFERENCES

[1] Arvind Arasu, Shivnath Babu, and Jennifer Widom. 2006. The CQL continuous query language: semantic foundations and query execution. *VLDB J.* 15, 2

(2006), 121–142. https://doi.org/10.1007/s00778-004-0147-z

[2] Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, and Matei Zaharia. 2018. Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 601–613. https://doi.org/10.1145/3183713.3190664

[3] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives (Eds.). ACM, 1383–1394. https://doi.org/10.1145/2723372.2742797

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

[5] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. 2002. Models and Issues in Data Stream Systems. In *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 3-5, Madison, Wisconsin, USA*, Lucian Popa, Serge Abiteboul, and Phokion G. Kolaitis (Eds.). ACM, 1–16. https://doi.org/10.1145/543613.543615

[6] Edmon Begoli, Tyler Akidau, Fabian Hueske, Julian Hyde, Kathryn Knight, and Kenneth L. Knowles. 2019. 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 Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 1757–1772. https://doi.org/10.1145/3299869.3314040

[7] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire. 2018. Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 221–230. https://doi.org/10.1145/3183713.3190662

[8] Pierre Bourhis, Juan L. Reutter, Fernando Suárez, and Domagoj Vrgoc. 2017. JSON: Data model, Query languages and Schema specification. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017, Chicago, IL, USA, May 14-19, 2017*, Emanuel Sallinger, Jan Van den Bussche, and Floris Geerts (Eds.). ACM, 123–135. https://doi.org/10.1145/3034786.3056120

[9] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink™: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.* 38, 4 (2015), 28–38. http://sites.computer.org/debull/A15dec/p28.pdf

[10] Paris Carbone, Jonas Traub, Asterios Katsifodimos, Seif Haridi, and Volker Markl. 2016. Cutty: Aggregate Sharing for User-Defined Windows. In *Proceedings of the 25th ACM International Conference on Information and Knowledge Management, CIKM 2016, Indianapolis, IN, USA, October 24-28, 2016*, Snehasis Mukhopadhyay, ChengXiang Zhai, Elisa Bertino, Fabio Crestani, Javed Mostafa, Jie Tang, Luo Si, Xiaofang Zhou, Yi Chang, Yunyao Li, and Parikshit Sondhi (Eds.). ACM, 1201–1210. https://doi.org/10.1145/2983323.2983807

[11] Sharma Chakravarthy and D. Mishra. 1994. Snoop: An Expressive Event Specification Language for Active Databases. *Data Knowl. Eng.* 14, 1 (1994), 1–26. https://doi.org/10.1016/0169-023X(94)90006-X

[12] Gianpaolo Cugola and Alessandro Margara. 2012. Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.* 44, 3 (2012), 15:1–15:62. https://doi.org/10.1145/2187671.2187677

[13] Nihal Dindar, Nesime Tatbul, Renée J. Miller, Laura M. Haas, and Irina Botan. 2013. Modeling the execution semantics of stream processing engines with SECRET. *VLDB J.* 22, 4 (2013), 421–446. https://doi.org/10.1007/s00778-012-0297-3

[14] EsperTech. 2006. *Esper EPL.* https://www.espertech.com/esper/

[15] Alejandro Grez, Cristian Riveros, Martín Ugarte, and Stijn Vansummeren. 2020. On the Expressiveness of Languages for Complex Event Recognition. In *23rd International Conference on Database Theory, ICDT 2020, March 30-April 2, 2020, Copenhagen, Denmark (LIPIcs, Vol. 155)*, Carsten Lutz and Jean Christoph Jung (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 15:1–15:17. https://doi.org/10.4230/LIPIcs.ICDT.2020.15

[16] Alejandro Grez, Cristian Riveros, Martín Ugarte, and Stijn Vansummeren. 2021. A Formal Framework for Complex Event Recognition. *ACM Trans. Database Syst.* 46, 4 (2021), 16:1–16:49. https://doi.org/10.1145/3485463

[17] Robert Griesemer. 2008. Parallelism by design: data analysis with sawzall. In *Sixth International Symposium on Code Generation and Optimization (CGO 2008), April 5-9, 2008, Boston, MA, USA*, Mary Lou Soffa and Evelyn Duesterwald (Eds.). ACM, 3. https://doi.org/10.1145/1356058.1356089

[18] Ananth Gundabattula and Thomas Weise. 2019. Apache Apex. In *Encyclopedia of Big Data Technologies*, Sherif Sakr and Albert Y. Zomaya (Eds.). Springer. https://doi.org/10.1007/978-3-319-63962-8_316-1

[19] Martin Hirzel, Guillaume Baudart, Angela Bonifati, Emanuele Della Valle, Sherif Sakr, and Akrivi Vlachou. 2018. Stream Processing Languages in the Big Data Era. *SIGMOD Rec.* 47, 2 (2018), 29–40. https://doi.org/10.1145/3299887.3299892

[20] James N. Hughes, Matthew D. Zimmerman, Christopher N. Eichelberger, and Anthony D. Fox. 2016. A survey of techniques and open-source tools for processing streams of spatio-temporal events. In *Proceedings of the 7th ACM SIGSPATIAL International Workshop on GeoStreaming, IWGS@SIGSPATIAL 2016, Burlingame, California, USA, October 31 - November 3, 2016*, Farnoush Banaei Kashani, Chengyang Zhang, and Abdeltawab M. Hendawi (Eds.). ACM, 6:1–6:4. https://doi.org/10.1145/3003421.3003432

[21] Muhammad Hussain Iqbal, Tariq Rahim Soomro, et al. 2015. Big data analysis: Apache storm perspective. *International journal of computer trends and technology* 19, 1 (2015), 9–14.

[22] Hojjat Jafarpour and Rohan Desai. 2019. 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*, Melanie Herschel, Helena Galhardas, Berthold Reinwald, Irini Fundulaki, Carsten Binnig, and Zoi Kaoudi (Eds.). OpenProceedings.org, 524–533. https://doi.org/10.5441/002/edbt.2019.48

[23] Namit Jain, Shailendra Mishra, Anand Srinivasan, Johannes Gehrke, Jennifer Widom, Hari Balakrishnan, Ugur Çetintemel, Mitch Cherniack, Richard Tibbetts, and Stanley B. Zdonik. 2008. Towards a streaming SQL standard. *Proc. VLDB Endow.* 1, 2 (2008), 1379–1390. https://doi.org/10.14778/1454159.1454179

[24] Shen Li, Paul Gerver, John Macmillan, Daniel Debrunner, William Marshall, and Kun-Lung Wu. 2018. Challenges and Experiences in Building an Efficient Apache Beam Runner For IBM Streams. *Proc. VLDB Endow.* 11, 12 (2018), 1742–1754. https://doi.org/10.14778/3229863.3229864

[25] Oracle. 2012. *OracleCEP.* https://docs.oracle.com/cd/E13213_01/wlevs/docs30/

[26] Matthias J. Sax, Guozhang Wang, Matthias Weidlich, and Johann-Christoph Freytag. 2018. Streams and Tables: Two Sides of the Same Coin. In *Proceedings of the International Workshop on Real-Time Business Intelligence and Analytics, BIRTE 2018, Rio de Janeiro, Brazil, August 27, 2018*, Malú Castellanos, Panos K. Chrysanthis, Badrish Chandramouli, and Shimin Chen (Eds.). ACM, 1:1–1:10. https://doi.org/10.1145/3242153.3242155

[27] Robert Soulé, Martin Hirzel, Robert Grimm, Bugra Gedik, Henrique Andrade, Vibhore Kumar, and Kun-Lung Wu. 2010. A Universal Calculus for Stream Processing Languages. In *Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6012)*, Andrew D. Gordon (Ed.). Springer, 507–528. https://doi.org/10.1007/978-3-642-11957-6_27

[28] Michael Stonebraker, Ugur Çetintemel, and Stanley B. Zdonik. 2005. The 8 requirements of real-time stream processing. *SIGMOD Rec.* 34, 4 (2005), 42–47. https://doi.org/10.1145/1107499.1107504

[29] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. 2002. StreamIt: A Language for Streaming Applications. In *Compiler Construction, 11th International Conference, CC 2002, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings (Lecture Notes in Computer Science, Vol. 2304)*, R. Nigel Horspool (Ed.). Springer, 179–196. https://doi.org/10.1007/3-540-45937-5_14

[30] Riccardo Tommasini, Sherif Sakr, Emanuele Della Valle, and Hojjat Jafarpour. 2020. Declarative Languages for Big Streaming Data. In *Proceedings of the 23rd International Conference on Extending Database Technology, EDBT 2020, Copenhagen, Denmark, March 30 - April 02, 2020*, Angela Bonifati, Yongluan Zhou, Marcos Antonio Vaz Salles, Alexander Böhm, Dan Olteanu, George H. L. Fletcher, Arijit Khan, and Bin Yang (Eds.). OpenProceedings.org, 643–646. https://doi.org/10.5441/002/edbt.2020.84

[31] WSO2. 2022. *WSO2 Stream Processor.* https://docs.wso2.com/display/SP400/Introducing+Stream+Processor