EPL: The Event Processing Language for Streaming Data (Technical Report)

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

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

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

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

1 MATCH RECOGNIZE & ARRAYS

1.1 EPL Data Model

EPL has an object-oriented data model that revolves around the concept of *event*. An Event is defined as a set of attribute-value mappings, where values may be of arbitrary types. Without loss of generality, we consider the following four: strings, numbers, arrays, and nested events. In the JSON data model [1], JSON Objects are structurally equivalent to EPL events. However, JSON Objects are schemaless and time agnostic. Thus, we extend the JSON data model with temporal annotations and we constrain it with a schema. In Table 1, we report the mappings between the two data models, highlighting the correspondences between events, objects, types, schema, and features.

JSON Values and Objects. In the JSON Data Model, JSON values are inductively defined as either *strings*, *numbers*, *JSON objects*, and *JSON arrays*. More formally, given the set of all unicode characters Σ, a string s is an element of Σ^*. Consequently, the set of real numbers $\mathbb{R}$ is a subset of Σ^*. Given a set of distinct strings $s_1, ..., s_n$ and a set of generic JSON values $v_1, ..., v_n$, we call *JSON object* the 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. Last but not least, empty JSON objects are represented as $\{\}$.

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 o nested events, numerical values, or strings. and *group attributes*, which we map to arrays.

Definition 1.1. 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,

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

Additionally, two schemas R_1, R_2 are said *compatible*, if they are disjointed, 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. 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, $events(\epsilon)$ identifies the set of all possible events, since for any given R, $\epsilon \subset R$. On top of this, the empty event $\{\}$ does not adhere to any schema except ϵ.

Another major aspect of the EPL 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*

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

EPL supports event contemporaneity. Indeed, the timestamp function may map two events to the same timestamp.

Definition 1.2. 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 *stream* the partially ordered set $S = (\mathbb{E}, \leq_t)$.

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

EPL Data Model		JSON Data Model [1]
Events	$\subset$	JSON Objects
Strings	$\equiv$	Strings
Arrays	$\equiv$	JSON Arrays
Numbers	$\subset$	Numbers
Event schemas	nd	nd
Time Awareness	nd	nd

Table 1: Mappings between EPL and JSON.

timestamps.

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

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 $CE \subset MN$). Last but not least, all events e_1, e_2, e_3 adhere to 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.

1.2 Syntax & Semantics

In this section, we present EPL syntax and semantics. EPL contains multiple constructs combined according to the grammar in Table 2. 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
(6) *Regexp* are expressed using Match-Recognize, they are evaluated against a stream of events and a set of attribute-condition mappings, and return a set of matches, i.e., ordered 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[1]. All the constructs and queries presented in this paper can be executed on a custom Jupyter environment available on GitHub[2].

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.

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

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. If the attribute is a group attribute in the **MATCH_RECOGNIZE**, the aggregation is performed on the mapped array. In Table 4, we reported all the aggregation functions used in this work. For instance, $avg(att)$ calculates the average of either all the values mapped to attribute att in stream $S = (\{e_1, e_2, ..., e_n\}, \leq_t)$, or all values in the array mapped to att in a reference event e.

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

Conditions are collections of EPL predicates, i.e., relations between terms (and aggregates) based on mathematical operators (=, <, > ...). 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 evaluation of a simple condition *cond* against an event e.

$$[\![P(\delta_1, ..., \delta_k)]\!]_e = \begin{cases} true & P([\![\delta_1]\!]_e, ..., [\![\delta_k]\!]_e)\ holds \\ false & otherwise \end{cases}$$
$$[\![cond_1 \ \texttt{AND}\ cond_2]\!]_e = [\![cond_1]\!]_e \wedge [\![cond_2]\!]_e$$
$$[\![cond_1 \ \texttt{OR}\ cond_2]\!]_e = [\![cond_1]\!]_e \vee [\![cond_2]\!]_e$$
$$[\![\texttt{NOT}\ cond]\!]_e = \neg [\![cond]\!]_e$$

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**).

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

MATCH_RECOGNIZE (MR) is a construct part of the SQL-standard. It is used to recognize patterns of data "rows". In EPL, these rows are the events, and the patterns are defined through some Regular Expression (regex). Table 5 shows the semantics of all MR constructs. The evaluation of MR constructs is always performed against a stream of events with a given schema R (S_R). This schema can either be a simple or a more generic, *variant* schema. In this work, we consider only simple schemas, leaving variant schemas for a future extension. The stream can be optionally partitioned through the **PARTITION BY** construct, isolating matching patterns according to a set of grouping attributes. We call the results of these evaluations *matches*, i.e., events with totally ordered mappings. Consequently, the constructs' semantics defined over events can also be used for matches.

Matches are the results of *MR Patterns* evaluations. These patterns are defined through regexes, which are composed of a set of attributes. Each of the attributes is associated with an aggregate condition $acond_i$ through the **DEFINE** syntax. As a

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

γ : α **FROM** β | α **FROM** β **GROUP BY** Δ
| α **FROM** β **GROUP BY** Δ **HAVING** *acond*
α : **SELECT** (Δ, AGG)
β : ι **WHERE** *cond* | ι
| R **MATCH_RECOGNIZE** (μ)
cond : $P(\delta_1, ..., \delta_k)$ | *cond* **AND** *cond*
| *cond* **OR** *cond* | **NOT** *cond*
ι : ω | $\omega\omega$ | ω **JOIN** ω **ON** *cond* | ξ
ω : ξ**#win:**f | ξ**#win:**f **AS** x
ξ : R | $R(cond)$ | **PATTERN** $[\rho]$

ρ : ρ''' | $\rho' \rightarrow \rho'$ | ρ' **AND** ρ'
| **EVERY** $(\rho''' \rightarrow \rho''')$ | **EVERY** $(\rho'''$ **AND** $\rho''')$
ρ' : ρ'' **AND NOT** R | ρ'' **where timer:within**(n) | ρ''
ρ'' : **EVERY** ρ''' | ρ'''
ρ''' : x=R | x=$R(cond)$ | ρ''' **OR** ρ'''
μ : **PARTITION BY** $\Delta\ \zeta$ | ζ
ζ : **MEASURES** (Δ, AGG)
AFTER MATCH SKIP *sel_policy*
PATTERN η **INTERVAL** n
DEFINE x_1 **AS** $acond_1, ...x_n$ **AS** $acond_n$

Table 2: The EPL Grammar.

Name	Notation	Definition
-	$n[S$	$\{e \mid e \in S \wedge t(e) > n\}$
-	$S]n$	$\{e \mid e \in S \wedge t(e) < n\}$
-	S_R	$events(R) \cap S$
support	$supp(\rho, S)$	$\{n \mid \exists e \in [\![\rho]\!]_S, \wedge t(e) = n\}$
merge	$S_1 \bowtie S_2$	$\{e_1 \cup e_2 \mid e_1 \in S_1,\ e_2 \in S_2\ compat. \wedge t(e_1 \cup e_2) = \max(t(e_1), t(e_2))\}$
first	$first(S)$	$\arg\min_{e \in S} t(e)$
last	$last(S)$	$\arg\max_{e \in S} t(e)$

Table 3: Summary of used notations.

Aggregation	Description
$[\![avg(att)]\!]_{S,e}$	It returns the average of the value mapped to attribute *att*
$[\![prev(lb.att)]\!]_{S,m}$	It returns the previously matched element within a match m
$[\![prev(lb.att, n)]\!]_{S,m}$	It returns the n-th previosly matched element within a match m

Table 4: Summary of Aggregation Functions.

consequence, the regex evaluation needs to take into account this set of mappings that we call $\mathbb{D}$. In the following, we inductively define the semantics of regex patterns in terms of result matches.

The basic regex is a single attribute (*lb*). The regex's results are matches of single events that satisfy the attribute's related condition. These matches can be extended in multiple ways. *Concatenation* returns a match of two *contiguous* events, i.e., an event that immediately follows another event. *Alternation* (|) matches either one of two sub-regex. It is used to provide an alternative recognition of two different patterns of events. Its semantics is similar to the **OR** in EPL Patterns. *Optional* (?) matches either nothing (empty match) or a given label. It is used to provide optional matches which may or may not be included in the final result. *Kleene-Star (∗), Plus (+), and Cardinality ([n])* are successive iterations of Concatenation. However, they gather the matching events in an array. Each event is indexed according to its position in the sequence. Kleene-Star and Plus create an array of arbitrary size, while Cardinality returns an array of fixed size.

Selection Policies are policies that validate how overlapping matches are treated, introduced by **AFTER MATCH SKIP**. We define three policies on a pair of matches (m_1, m_2), based on the auxiliary functions *start* and *end*, and *events*, all reported in Table 6

INTERVAL limits the extent of the pattern match. To do so, it filters all matches according to the maximum temporal distance across the matched events.

$$checkInterval(m, n) = \begin{cases} 1 & t(end(m)) - t(start(m)) < n \\ 0 & otherwise \end{cases}$$

MEASURES selects specific match values through the match's labels. The operation is similar to the **SELECT**, but here it does not operate on a stream of events but simply on a set of matches. Additionally, we provide the definition of some special aggregation functions. The *prev* function can be evaluated in both **MEASURES** and **DEFINE** syntaxes. Given a reference event and an offset n, it is able to return the n-th preceding event.

$$[\![prev(lb.att)]\!]_{S,m} = e_{i-1}.att \text{ where}$$
$$\exists lb,\ s.t.\ m = ..., lb : [..., e_{i-1}, e_i, ...], ...$$
$$[\![prev(lb.att, n)]\!]_{S,m} = \begin{cases} prev(prev(lb.att), n-1) & if\ n > 1 \\ prev(lb.att) & otherwise \end{cases}$$

Example. *The Match-Recognize is used to detect "bull runs", rapid increases in the stock price followed by a sudden drop.*

```
SELECT * FROM Crypto
MATCH_RECOGNIZE (
PARTITION BY Crypto.id
MEASURES B.id AS id, max(A.v)
 AS M, min(A.v) AS m,
((A.firstOf().ts) − (A.lastOf().ts))/86400 AS d
AFTER MATCH SKIP pastLast
PATTERN (A* B)
DEFINE A AS
A.amount >= 1.20 ∗ prev(A.amount)
DEFINE B AS
B.amount < 0.80 ∗ A.lastOf().amount))
```
(Q1)

$[\![R \ \texttt{MATCH\_RECOGNIZE} \ (\mu)\ ]\!]_{S,} = [\![\mu]\!]_{S_R,}$

$[\![\texttt{PARTITION BY} \ \delta_1, ..., \delta_n \ \zeta]\!]_{S,} = \bigcup_{v_1 \in \mathbb{V}, ... v_n \in \mathbb{V}} [\![\zeta]\!]_{S'},$

where $S' = [\![R(\delta_1 = v_1 \ \texttt{AND} \ ... \delta_n = v_n)]\!]_{S_R,}$

$[\![\texttt{MEASURES} \ \delta_1 \ \texttt{AS} \ "s_1", ..., agg_1 \ \texttt{AS} \ "s_{n+1}", ... \ \zeta]\!]_{S_R,} =$

$\bigcup_{m \in M} ("s_1" : [\![\delta_1]\!]_m, ..., "s_{n+1}" : [\![agg_1]\!]_m, ...)$ with $M = [\![\zeta]\!]_{S_R,}$

$[\![\texttt{AFTER MATCH SKIP} \ sel\_policy \ \zeta]\!]_{S_R,} =$

$\{m_1, m_2 \mid m_1 \in [\![\zeta]\!]_{S_R,} \wedge m_2 \in [\![\zeta]\!]_{S_R,}, \ sel\_policy(m_1, m_2) \wedge$
$\forall m_2' \ t(end(m_1)) < t(start(m_2)) \leq t(start(m_2'))\}$

$[\![\texttt{PATTERN} \ \eta \ \texttt{INTERVAL} \ n \ \texttt{DEFINE} \ x_1 \ \texttt{AS} \ acond_1, ...]\!]_{S_R,} =$

$\{m \mid m \in [\![\eta]\!]_{S_R,\mathbb{D}} \wedge checkInterval(m, n) \wedge \mathbb{D} = \bigcup_i (x_i, acond_i)\}$

$[\![lb]\!]_{S_R,\mathbb{D}} = \{lb : e \cdot \mid lb \in \mathbb{L} \wedge e \in S_R \wedge t(e) \in \wedge$
$\exists (lb, acond) \in \mathbb{D} \ s.t. \ [\![acond]\!]_{S,e} = true\}$

$[\![\rho_1 \ \rho_2]\!]_{S_R,\mathbb{D}} = \{m_1, m_2 \mid m_1 \in [\![\rho]\!]_{S_R} \wedge m_2 \in [\![\rho_2]\!]_{S_R}$
$\wedge \forall m_2' \in [\![\rho_2]\!]_{S_R,\mathbb{D}} \ (start(m_2) = start(m_2') \vee$
$(t(end(m_1)) < t(start(m_2)) \leq t(start(m_2'))))\}$

$[\![\rho_1 \mid \rho_2]\!]_{S_R,\mathbb{D}} = [\![\rho_1]\!]_{S_R,\mathbb{D}} \cup [\![\rho_2]\!]_{S_R,\mathbb{D}}$

$[\![\rho?]\!]_{S_R,\mathbb{D}} = \{\} \cup [\![\rho]\!]_{S_R,\mathbb{D}}$

$[\![lb[n]]\!]_{S_R,\mathbb{D}} = \{lb : [E] \mid \forall i \ s.t. \ 0 < i < n,$
$(lb' : v[i-1], lb'' : v[i]) \subseteq E \rightarrow$
$(lb' : v[i-1], lb'' : v[i]) \in [\![lb' \ lb'']\!]_{S_R,\mathbb{D}}\}$

$[\![lb+]\!]_{S_R,\mathbb{D}} = \bigcup_{i=1}^{\infty} [\![lb[i]]\!]_{S_R,\mathbb{D}}$

$[\![lb*]\!]_{S_R,\mathbb{D}} = \bigcup_{i=0}^{\infty} [\![lb[i]]\!]_{S_R,\mathbb{D}}$

Table 5: The Match-Recognize construct's Semantics.

The query evaluation is partitioned according to the stock name. It searches for arbitrarily long sequence of events labeled A, followed by an event labelled B. The two labels are mapped to specific conditions, resulting in the following set of mappings $\mathbb{D}$

$$\mathbb{D} = \{(A, A.amount >= 1.20 * prev(A.amount)),$$
$$(B, B.amount < 0.80 * A.lastOf().amount))\}$$

Each evaluations searches for n contiguous A events, i.e., events that strictly follow each other, and no event is matched between them, leading to the evaluation of the following pattern

$[\![A[0] \ A[1]...]\!]_{S_R,\mathbb{D}} = \{A[0] : e_0, A[1] : e_1, ...\} = m$ such that

$\forall i [\![A[i].amount >= 1.20 * A[i-1].amount]\!]_{S,m} = true$

Ultimately, all the matching events are gathered into an array, which concatenates with a B event. The resulting match is then filtered through the the selection policy: if all events satisfy the pastLast predicate, the match is validated, resulting in only subsequent matches. The matches are then projected on the maximum and minimum of the events gathered in the array.

REFERENCES

[1] 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

Predicate/Function	Formal Definition	Description
$start(m)$	$start(m) = e_1 \leftrightarrow \exists lb,\ m = lb : e_1, \ldots \vee m = lb : [e_1, \ldots], \ldots$	It returns the first event of a given match m
$end(m)$	$end(m) = e_n \leftrightarrow \exists lb,\ m = \ldots, lb : e_n \vee m = \ldots, lb : [\ldots, e_n]$	It returns the last event of a given match m
$events(m)$	$\{e \mid \exists lb\ s.t.\ (lb : e) \in m\}$	It returns all the events within a given match m
$pastLast(m_1, m_2)$	$events(m_1) \cap events(m_2) = \emptyset \wedge t(start(m_1)) < t(start(m_2))$	It considers only subsequent matches, with no events in common
$toNext(m_1, m_2)$	$m_1 = lb_1 : start(m_1), lb_2 : start(m_2), \ldots \wedge m_1 = lb'_1 : start(m_2), \ldots \vee pastLast(m_1, m_2)$	It considers those matches with at least one event in common
$toCurrent(m_1, m_2)$	$start(m_1) = start(m_2) \vee toNext(m_1, m_2)$	It considers events that start on the same event

Table 6: Summary of selection policies and their auxiliary functions.