Extending Kafka Streams for Complex Event Recognition

Anonymous Authors

***Abstract*—Streaming Analytics (SA) and Complex Event Recognition (CER) are of paramount importance in searching for an ultimate Big Data solution that can simultaneously address Data Velocity, Variety, and Volume. Indeed, the growing popularity of streaming data has pushed the boundaries of existing data systems, fostering the rise of Stream Processing Engines (SPE). However, Data Velocity never appears isolated. Streams are huge, heterogeneous, and noisy as they come from multiple sources. Horizontally-scalable SPEs like Flink and KSQL-DB allow continuous stream analytics using SQL-like languages. On the other hand, CER engines like OracleCEP and DroolFusion use regular languages for (parallel) pattern detection over heterogeneous streams. This paper takes a first step towards a unifying solution. To this extent, we present KELPr, an in-memory distributed CER engine designed extending the Dual Streaming Model and implemented on top of Kafka Streams.**

***Index Terms*—stream processing, complex event recognition, complex event processing, stream reasoning**

I. Introduction

In recent years, we are witnessing an acceleration in the way data are generated. The growing popularity of sensors data has pushed the boundaries of existing data systems, stressing the need for processing data as soon as they are available and before they are no longer useful. However, data velocity rarely appears in isolation. Data streams are often vast and should be joined with huge datasets (Volume). Moreover, most of the information needs to span across a multitude of heterogeneous data sources (Variety) [1].

Stream Processing is the research area that investigates how to address the challenges above. The related literature distinguishes between languages for Stream Analytics (SA) and Complex Event Recognition (CER) [2]. Languages for SA address the infinite nature of data streams by the means of operators called windows, which chunk the stream into finite portions and enable stateful computations [2]. Stream Processing languages for Big Data like Spark Structured Streaming [3] and KSQL-DB belong to the SA family [4], [5]. Notably, they simultaneously address data velocity and volume, but they neglect variety. Indeed, data are supposed to be homogeneous and often in relational form.

Solutions for CER use pattern-based languages, often enriched with probabilistic features [6] to detect correlations over input streams. CER languages work at a higher level of abstraction than SA ones. They allow composing events hierarchically decoupling from the possibly various raw data formats. In practice, CER languages address Data Variety and Velocity at the same time. However, the expressiveness of the languages often limits the scalability of the approaches.

In searching for an ultimate Big Data solution that can simultaneously address the Variety, Velocity, and Volume, SA and CER are complementary. Both the approaches endorse the declarative paradigm, allowing to optimize user-defined solutions. However, a unifying solution is still missing. In particular, the integration of CEP and SA features requires to investigate the trade-off between the language expressiveness and the scalability of the system.

Among the aforementioned languages for Big Streaming Data processing, Flink SQL and KSQL-DB look as promising starting points for the integration of CER features. Indeed, their underlying query engines expose extensible domain-specific languages (DSL) and a streaming-first execution paradigm. In fact, FlinkCEP Library is a notable attempt to embed CER functionality in Flink's SQL (using Match_Recognize Syntax). However, its scalability is limited if compared with the rest of Flink's operators. The main reason is due to Flink's deployment model and the known issues with distributed CER [7].

On the other hand, KSQL-DB uses Kafka Streams as an execution engine, which in turn has a simpler deployment model, closer to the one of parallel CER engines. Moreover, Kafka Streams comes with a formalized model called Dual Stream Model (DSM) that explains its operational semantics and fosters extensions [8].

In this paper, we present KELPr, i.e., an in-memory distributed CER engine built on top of Kafka Streams. KELPr execution semantics is defined as an extension of the Dual Streaming Model (DSM). Section II presents the necessary background knowledge about the DSM. Section III introduces the details of KEPLr's Model, a DSM extension for complex event recognition on top of record streams. Moreover, Section IV discusses the design of a proof of concept implementation and the results of its preliminary evaluation using synthetic data. Finally, Section VII draws conclusions and presents future works.

II. Background

In this section, we present the background knowledge on Stream Processing. In particular, we focus on the Stream Duality Model on which we built our contribution.

Stream Processing engines (SPE) reverse the interaction model of traditional Data Base Management Systems (DBMS). While DBMSs perform one-time queries and transformations over stored data, SPEs perform evaluations over information flows on the fly. Existing SPEs inherit their data model and their query model from Relational DBMSs. Nevertheless, they are designed according to the notion of *continuous semantics*: they produce an infinite output from an infinite input [9].

In these regards, the seminal work of Arasu et al. [9] represents the starting point of existing declarative stream processing languages [5]. The Continuous Query Language (CQL) extends relational algebra to process streams of tuples according to the discrete notion of time and three families of operators [9] depicted in Figure 1.

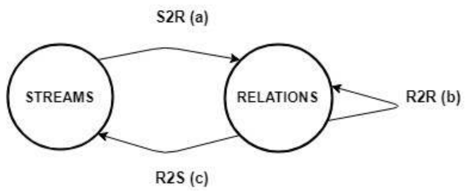


Fig. 1: CQL's interaction between streams and relations.

Stream-to-Relation 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 operators that produce a stream from a relation, e.g., RStream, which timestamps the result of the R2R operator.

Sax et al. [8] introduce the Dual Streaming Model (DSM) to reason about physical and logical order in data stream processing. The model builds upon the concepts presented by Arasu et al. [9]. In particular, it extends the notion of continuous semantics to consider out-order events. At the core of the model, there is the duality between *stream* and *table*, outlined through a data model and a set of related operators. In turn, these two concepts rely on the concept of *data record*.

Definition 1. The DSM defines a *data stream* as an unbounded, append-only, sequence of immutable records.

A record r is a tuple $\langle k, v, o, t\rangle \in \mathbb{D}_K \times \mathbb{D}_V \times \mathbb{N}_0 \times \mathbb{N}_0$, where (i) k is the key of the record and $\mathbb{D}_K$ is the keyspace, (ii) v is the value of the record and $\mathbb{D}_V$ is the value space, (iii) o is a unique offset expressed as a non-negative integer ($\mathbb{N}_0$), (iv) t is the timestamp of the record assigned by the record source expressed as a non-negative integer ($\mathbb{N}_0$).

Notably, the record offset is assigned incrementally and determines a total **ordering** of the stream. On the other hand, the record timestamp is assigned by the record source, and it determines a **partial ordering** of the stream. Intuitively, the latter allows out-of-order elements in the stream.

When records are interpreted as facts we call the stream *record stream*. On the other hand, when we interpret the records as updates we talk about *changelog streams*. The key semantics distinguishes between record and changelog stream more formally. In the former, all the keys are considered different. In the latter, a more recent record overrides a previous record with the same key.

Also, record stream processing requires continuous semantics [9]. Operators compute their results according to either physical or logical order, resulting in either a relation or a

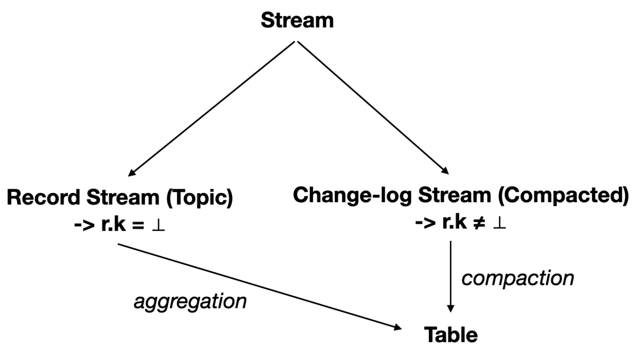


Fig. 2: The Stream Duality *Data* Model [8].

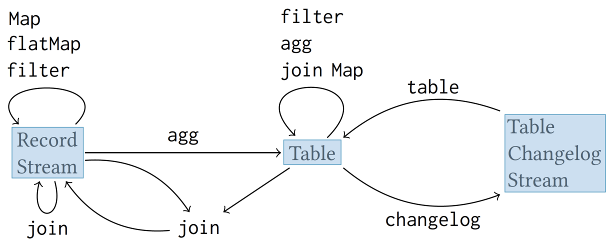


Fig. 3: The Stream Duality *Processing* Model [8].

stream. Instead, changelog stream processing requires an additional abstraction, i.e., the concept of *table*. More precisely, DSM defines a *table* as a collection of multiple table versions[1]. Each version represents the state of the table up to a certain point in time. In particular, we denote as T_t the table version of a certain time instant t.

Tables and changelog streams are two interpretations of the same abstraction. Thus, one can seamlessly switch from one to the other. A changelog stream can be reduced to a table through *compaction*[2] over record keys [10]. Instead, a record stream can be reduced to a table through an *aggregation*.

Besides compaction and aggregation, *Sax et al.* introduce additional ways of manipulating data in the form of *Stateless* and *Stateful* operators (see Figure 3).

Operator	1st Input	2nd Input	Output
`filter, mapValue`	KStream		KStream
	KTable		KTable
`map, flatMap`	KStream		KStream
`groupBy` $\rightarrow$ `agg`	KStream		KTable
	KTable		KTable
`groupBy + windowBy` $\rightarrow$ `agg`	KStream		KTable
`inner-/left-/outer-join`	KStream	KStream	KStream
`inner-/left-/outer-join`	KTable	KTable	KTable
`inner-/left-join`	KStream	KTable	KStream

TABLE I: The various operators of the Kafka Stream's DSL, with related inputs and outputs.

Stateless operators are applied to record streams. In particular, they are applied to each record of the input stream

[1]Presented in CQL as time-varying relation [9]

[2]Retaining the last events for each key

according to the physical order, producing an order-preserving output stream of results. This family of operators can be exemplified by the higher-order functions `filter`, `map`, and `flatMap`.

Stateful operators need to maintain the state of the computation. A relevant example is the `join` operation. It can be performed in three ways: a table-table join, a stream-table join, and a stream-stream join. All three operations exploit table versioning and stream windowing to perform temporal joins across the two representations.

As mentioned in Section I, Kafka Streams, i.e., a stream processing library built on top of Apache Kafka, is based on the Dual Streaming Model. In particular, there are two first-class entities in Kafka Streams: *KStream* and *KTable* correspond to record stream and table/changelog stream, respectively.

The Kafka Streams DSL implements the Dual Streaming Model and its operators, here depicted in Table I with their inputs and outputs.

III. KEPLr Model

In the following, we introduce the KEPLr's Data Model and Processing Model, which are the CER extension for the Dual Streaming Model [8]. By extending the DSM, we can also rely on the formal guarantees regarding out-of-order computation, as well as the theorems and related proofs exploiting stream equivalence and operators' correctness.

A. Data Model

As described in Section I, most of the CER languages are strongly typed, and the event type is used as a foundation for event detection. In particular, the event type is used to identify the stream and perform processing operations.

In the Dual Streaming Model, the concepts of event and event type are missing. However, DSM presents the concept of *schema* as a way to introduce data streams (see Definition 1) without providing an extensive analysis. A Schema is simply defined as an auxiliary structure for characterizing a data stream, which is seen as a set of homogeneous records. Events make this relation explicit, filling the gap between actual data and metadata, and enabling the streams to contain heterogeneous elements. Therefore, we propose the following definitions.

Definition 2. Let $\mathbb{A}$ be the set of all possible attribute names, a type τ is a set of attributes, i.e., $\tau \in \mathcal{P}(\mathbb{A})$, where $\mathcal{P}(\mathbb{A})$ is the power set of $\mathbb{A}$.

Definition 3. Let $\mathbb{T}$ be the set of all types, an *event* is a tuple $\langle k, \tau, o, i\rangle \in \mathbb{D}_K \times \mathbb{T} \times \mathbb{N}_0 \times \mathbb{N}_0$, where τ is the type of the event, while the other elements are defined like in Definition 1. Given an event e, we will indicate its type as $e.\tau$.

An *event stream* is an unbounded, append-only sequence of events $(e_1, e_2, ...)$, where, $\forall i \; e_i.\tau \in \mathbb{T}$.

Moreover, events are typically organized in hierarchies [11]. Thus, we provide a practical definition for type inheritance related to event attributes.

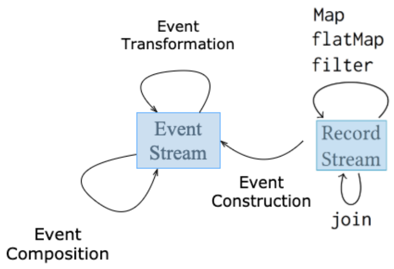


Fig. 4: The KEPLr's Model.

Definition 4. Given two types τ_1, τ_2, we say that τ_1 *inherits from* τ_2 if and only if $\tau_1 \subset \tau_2$.

Notably, from Definition 2 and 4 we can derive a special event-type $\top = \emptyset$, which corresponds to the empty set of attributes, and from which all the event-types inherit.

B. Processing Model

The DSM's main goal is to reason about the semantics of Kafka Streams operators. Due to such a high level of abstraction, the record's internal structure is kept as simple as possible. However, in real-world scenarios, the data model is more complex. For instance, Kafka Streams and Kafka use nested key and value pairs. Therefore, it makes sense to assume that records have an internal structure, e.g., a value can be a JSON or a tuple. To link event construction with the record's inner structure, we bind the set of event type names $\mathbb{T}$ (cf Definition 3) with the set of possible record values (cf Definition 1).

Definition 5. We define the *type mapping* as a function

$$type :: \mathbb{D}_V \to \mathbb{T}$$

which, given a record value, returns its type.

In the following, we present the main operations of the KEPLr Model. They operate over event streams, working at the type level. In particular, they focus on type manipulation in terms of *construction, transformation,* and *composition*. Figure 4 shows how the operations interact with the different abstractions.

Event Construction allows changing the interpretation from a record stream to an event stream. It represents the link between the world of DSM records and the world of KEPLr events. To better explain the semantics behind the operation, we provide the following definition.

Definition 6. The *event construction* is an operation that creates an event e out of a record r. We will denote the related correlation between e and r as $r \vdash e$. In particular,

$$r \vdash e \implies type(r.v) = e.\tau$$

In practice, this operation is implemented through a specific operator, defined as a function

$$\texttt{match} :: (\mathbb{D}_K \times \mathbb{D}_V \times \mathbb{N}_0 \times \mathbb{N}_0) \to (\mathbb{D}_K \times \mathbb{T} \times \mathbb{N}_0 \times \mathbb{N}_0)$$

In particular, its evaluation at time t' over a record stream RS is defined as

$$\{e \mid \exists r \in RS \; s.t. \; r \vdash e \; \wedge \; e.\tau = T \; \wedge \; e.t = r.t = t'\}$$

The resulting operation is both logical and physical order-preserving. Additionally, the reuse of the key extends this property also to the partitioning strategy, avoiding the consequent re-partitioning.

Event Transformation allows moving from one event type to another mirroring the semantics of DSM stateless operators. Therefore, `filter, map,` and `flatMap` are valid example of type transformations. Nevertheless, KEPLr's operators work at the type level.

Definition 7 enlists the possible cases of event transformations: (a) describes the case where the output event type (τ_2) is a subset of the input event type (τ_1), which corresponds to going up in the event hierarchy. (b) describes the case where the input event type (τ_1) is a subset of the output event type (τ_2), which corresponds to going down in the event hierarchy. Finally, (c) describes the case where there is no overlap between the input (τ_1) and output event types (τ_2). In practices, this corresponds to the *type negation* operation, as the input event type (τ_1) cannot be re-derived from the output one (τ_2).

Definition 7. The *type transformation* between two types $\tau_1 \rightarrow \tau_2$ is derived by one of the following cases

$$\begin{cases} \tau_2 \subset \tau_1 \wedge \tau_1 \nsubseteq \tau_2 & (a) \\ \tau_1 \subset \tau_2 \wedge \tau_2 \nsubseteq \tau_1 & (b) \\ \tau_1 \cap \tau_2 = \emptyset \wedge \tau_1, \tau_2 \neq \top & (c) \end{cases}$$

In practice, (a) can be implemented using a `Map` operation that takes just a subset of input's attributes; (b) can be implemented using a `flatMap` that enriches the event type in all possible ways. Finally, (c) can be implemented using a `filter` operation that checks type-related conditions on the event stream, abstracting mostly known patterns like `not` or `instanceof`.

Event Composition allows joining event streams over a certain condition. Consequently, event composition generates *dynamic types*, i.e., types that are defined at runtime by the operator [11]. In particular, the type of output records is the result of the type composition between the input types ($T_{out} = T_1 \otimes T_2$) defined as follows by leveraging the set-oriented nature of event types:

Definition 8. The *type composition* between two types τ_1, τ_2 is a type $\tau_p = \tau_1 \otimes \tau_2$ derived by one of the following cases

$$\begin{cases} \tau_p = \tau_1 \cup \tau_2 \text{ iff } \tau_1 \vee \tau_2 \\ \tau_p = \{\tau_1, \tau_2\} \text{ iff } \tau_1 \wedge \tau_2 \end{cases}$$

Additionally, we can distinguish between two different cases:

- Composition of *Heterogeneous* events that joins two different event streams defined by as many types.
- Composition of *Homogeneous* events: that joins the same stream with itself, resulting in a self-join.

Moreover, although the join condition is generic, CER languages typically focus on time-related ones.

C. Operator Semantics

In the following, we specify the denotational semantics of the KEPLr's operators. To this extent, we define the following evaluation function

$$[\![\cdot]\!]_t$$

which indicates the evaluation of a KEPLr expression over the input event stream at time t.

The `not` operator selects events of all types but the one specified in the operator. In particular, we provide the evaluation at a specific time t'

$$[\![\texttt{not } T]\!]_{t'} = \{e \mid e.\tau \neq T, e.t = t'\}$$

Regarding heterogeneous joins, we decided to start with `followedBy`, also known as SEQ or sequence, that is a fundamental operator in many CER languages [12]. `FollowedBy` is a variant of heterogeneous join that allows detecting subsequent events in a stream. In KEPLr, the result of a `followedBy` operation performed at time t' is specified as

$$[\![\varphi_1 \texttt{ followedBy } \varphi_2]\!]_{t'} = \bigcup_{t''<t'} \{e \mid \exists e_1 \in [\![\varphi_1]\!]_{t''},$$
$$e_2 \in [\![\varphi_2]\!]_{t'} \; s.t. \; e.\tau = e_1.\tau \otimes e_2.\tau\}$$

Last but not least, regarding homogeneous join, we focus on the `iteration` operator. `Iteration` allows the gathering of an arbitrary number of events of the same type. It is a specific case of homogeneous join, where homogeneous successive events are merged to form a single, complex event. Since the number of events that can be accumulated is finite, we can see this operation as an operational chain of `followedBy`.

IV. Implementation

The Kafka Streams data model is based on KStream and KTable. To implement KEPLr, we introduce a new entity, i.e., the *KTStream*, that represents the *event stream* concept (see Definition 3). According to KEPLr's model, Event Stream's operators work at the type level. However, neither KStream nor KTable includes the concept of type. Thus, we introduce the *EType* class, which implements the concept of type presented in Definition 2. EType is implemented as an abstract, generic class. Additionally, two EType instances can be joined together through *type composition*, as presented in Definition 8. In practice, the KTStream implementation contains an EType instance, mirroring what is described in Definition 3. Operating at type-level would be extremely unpractical for KEPLr users, as they would require going back and forth between KStream and KTStream to alternate CER and SA operations. Thus, we implement the KTStream as a *wrapper* of the KStream.

Our implementation is based on Kafka Streams Processor APIs. In order to address the intrinsic statefulness of CEP operators, we designed a Finite State Machine within a Kafka in-memory state store to maintain the match progress. This approach enables computational distribution across hash-based partitions. Consequently, the matching process is partition-based: we are able to match events that belong to the same partition.

We opted to implement four operators, whose semantics has already been described in Section III. These operators are depicted in Table II with the related input and output types.

Operator	1st Input	2nd Input	Output
`match`	KStream		KTStream
`not`	KTStream		KTStream
`iteration`	KTStream	KTStream	KTStream
`followedBy`	KTStream	KTStream	KTStream

TABLE II: Table representing the new operators, with their input and output classes.

In general, the semantics of CER operators depends on which events should be considered over the stream. Since this can lead to ambiguities, Cugola and Margara introduce *selection* and *consumption* policies [13], which enrich the CER language to control and explain such differences.

The Event Processing Language (EPL) implements such policies using a higher-order operator called `every`, which controls what event should take part in the pattern recognition. The `every` gives us the possibility to integrate ad-hoc consumption and selection policies directly inside pattern expressions, following the example of EPL [14]. In practice, `every` forces the evaluation of the underlining pattern expression to restart whenever it terminates, iterating the event search. In practice, this is implemented through the spawning of a new recognition automaton in order to iterate the detection process of the underlining expression.

CER operators present different behaviors according to the multiple uses of the `every`. This is similar to what happens in the relational context, where join operations can be optionally defined as *left* or *right*.

In the following, we provide two examples, representing two of the cases from Table III, which presents all possible combinations of the `followedBy` operator and the `every` operator in EPL format.

Input	A1 B1 C1 B2 A2 D1 A3 B3 E1 A4 B4
Patterns	Match
`every ( A` $\rightarrow$ `B )`	{A1,B1};{A2,B3};{A4,B4}
`every A` $\rightarrow$ `B`	{A1,B1};{A2,B3};{A3,B3};{A4,B4}
`A` $\rightarrow$ `every B`	{A1,B1};{A1,B2};{A1 B3};{A1,B4}
`every A` $\rightarrow$ `every B`	{A1,B1};{A1,B2};{A1,B3};{A2,B3}; {A3,B3};{A1,B4};{A2,B4};{A3,B4}; {A4,B4}

TABLE III: The possible combinations between an `every` and a `followed-by` ($\rightarrow$) in EPL format.

Besides the use of the `followedBy` and the `every`, we provide an additional parameter. This addition is motivated by a feasibility requirement. From a purely formal perspective, a pattern can indefinitely wait for the arrival of a specific event. This subsistence of a partial result can lead to storage saturation, which is not acceptable in a streaming context. For this reason, we introduce the *within* parameter, which limits the search for a specific event across a specified temporal span.

Before introducing the actual examples, we describe some preliminary operations, displayed in Listing 1. In particular, the initial KStream is created from a topic “A;B”, which contains events of both types A and B (Line 4). Then, the `match` operation is applied (Line 6), taking as input the set of all possible types (Line 1-2) and the initial KStream instance. Eventually, the `match` generates various KTStream instances, which can be used for pattern composition.

```
1   type1 = new ETypeAvro(”A.asvc”);
2   type2 = new ETypeAvro(”B.ascv”);
3   KStream<String,Record> stream =
4     builder.stream(”A;B”);
5   KTStream[] eventstreams =
6           KTStream.match(stream, type1,type2);
7   KTStream streamA = typedStreams[0];
8   KTStream streamB = typedStreams[1]
```

Listing 1: The preliminary operations for the examples.

Example 1 can be seen as a join over time between two streams, with the join windows being defined by the *within* time. In this case, we take every possible combination of an event A followed by event B. Listing 2 shows the actual implementation through the KEPLr’s DSL. In particular, we can see the *every()* method applied to both streams (Line 1 and 2).

```
1   streamA.every()
2     .followedBy(streamB.every(), 5000L)
3     .to(”out”);
```

Listing 2: Case 1 represented through KEPLr’s DSL.

Example 2 represents the case in which the search of the predecessor never stops. This continuous search results in groups of pairs having the same successor (see Figure 5).

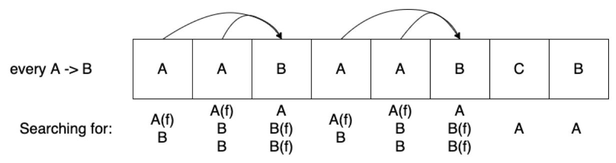
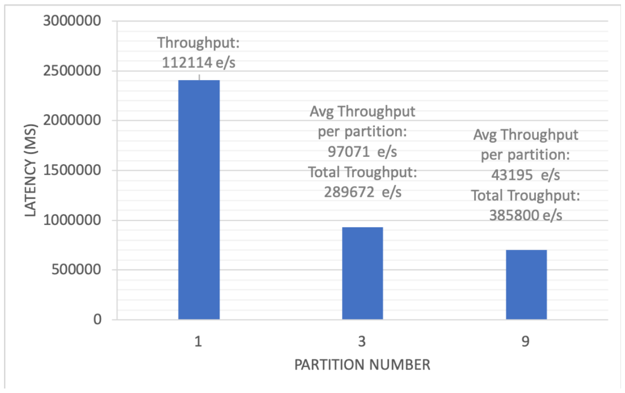


Fig. 5: In this case, matching pairs are grouped together according to their successor.

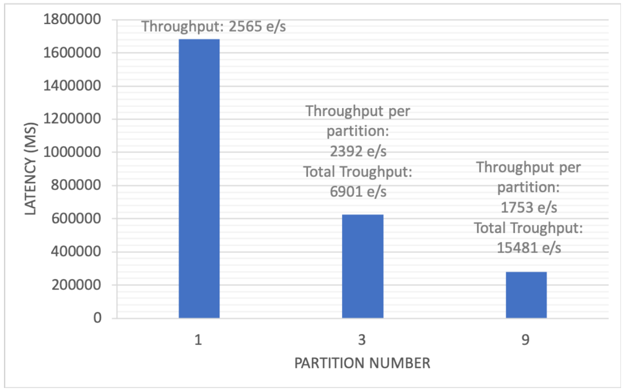
```
1   streamA.every()
2   .followedBy(streamB, 5000L)
3     .to(”out”);
```

Listing 3: Case 2 represented through KEPLr’s DSL.

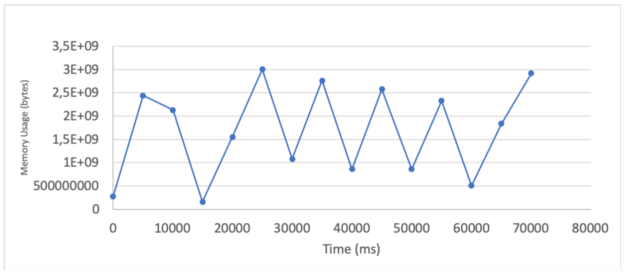
Listing 3 shows the actual implementation through the KEPLr’s DSL. In particular, we can see the *every()* method applied just to the predecessor stream (Line 1).



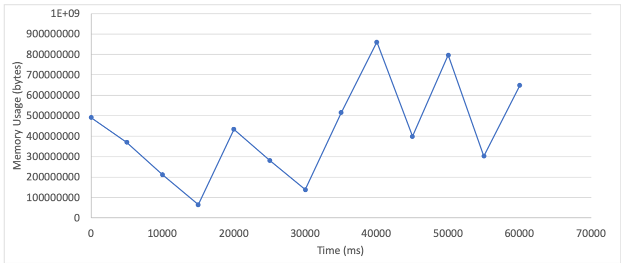
(a) Throughput of Example 1.



(b) Throughput of Example 2.



(c) A representation of W1 thread's memory usage.



(d) A representation of W2 thread's memory usage.

Fig. 6: The latency, related throughput, and memory usage of the examples.

V. Preliminary Evaluation

Last but not least, we couple each example with a performance assessment. In particular, we focus our analysis on both *correctness* and *throughput*. The former is measured comparing the actual cardinality of results with the expected one; the latter is measured varying the number of partitions. Although our implementation is distributed, we tested our system on a single physical machine, without considering fault tolerance. In particular, the experiments are performed on a Linux instance deployed on a server with a CPU of 20 cores @ 1.8Ghz, and a RAM DDR4 of 16GB 2133, provided by Cherry Servers[3].

Example 1 is the most complete in terms of results. It takes all combinations of an event A followed by an event B. From the throughput perspective, we notice a decrease of the latency from the single-partitioned case to the nine-partitioned case, due to computation scaling. The total throughput increases, but we notice a decrement of the per-partition throughput (see Figure 6a). The memory usage trend is related to the garbage collection of results. Thus, we have peaks of high memory usage with successive decreases, caused by the elimination of expired data (see Figure 6c).

Example 2 Increasing the number of partitions, in this case, results in a decrease of latency, and being the runs based on the same number of events, results in the throughput increase (see Figure 6b). Again, the memory usage is related to the garbage collection of results. Thus, it is still presenting a sawtooth trend (see Figure 6d). The frequency of the garbage collector is inferior with respect to Case 1. This is due to the inferior complexity of this query compared to the previous one. Indeed, in Example 1 we need to store much more partial matches since it looks for all possible combinations of an event with type A followed by an event with type B.

VI. Related Work

The idea of performing complex event processing on top of a distributed Big Data solution is explored also by other works. Giatrakos et. al. [7] specify how most of the CEP architecture is centralized since they have to rely on serial processing to uniquely identify the events' ordering. On the other hand, Big Data systems are based on a scalable architecture for taming Velocity and Volume. Thus, associating the two concepts is a complex task.

In order to tackle this problem, CEP computation needs to be implemented across a clustered architecture, in which the computation is **scaled-out** between multiple machines that work in parallel. Each machine generates a partial result, that is in the end combined to obtain the final aggregate. There are various ways of enabling parallelism, but two macro-categories can be identified: *Task-based parallelism* and *Data-based Parallelism*. The first relates to the parallelism between execution entities, e.g., operators, queries, nodes. In the second case, the parallelism is based on data, and on how data can be distributed among the computational units. In general, given

[3]Cherry Servers https://www.cherryservers.com/

the limitations that a single strategy possesses, it is possible to adopt hybrid approaches in order to compensate for the limitations of a model with another one's strengths. However, this can lead to other problems of the resulting hybrid system, and its compatibility with the operators.

Among the various solutions that try to perform CEP in a Big Data context, the most relevant is FlinkCEP [15], i.e., a complex event processing library built on top of Flink APIs. Similarly to our solution, the library provides a DSL for defining patterns, using non-deterministic automatons for complex event recognition. Patterns can be composed and nested within each other. FlinkCEP supports different categories of operators, spanning from sequences to logic operators, e.g., `followedBy` and `not`. Windowing is also supported for defining time-related boundaries to pattern evaluation, similarly to what happens in KEPLr. The use of consumption and selection policies is also allowed. In both cases, the policies are predefined, presenting less expressivity with respect to custom solutions like ours. The problem of predefined policies is related to the equivalence of the differences across different systems. Indeed, the semantics of FlinkCEP selection policies is not equivalent to one of the same construct in other state-of-the-art solutions [7]. Consequently, a formalization of the cited policies difficult to realize.

The main limitation of FlinkCEP is related to its scalability. Pattern recognition is performed on a single topology node, limiting the ability of the system to scale the computation across different tasks. Additionally, every FlinkCEP pattern is defined over a single DataStream instance. As consequence, pattern recognition is only possible over a single, homogeneous stream. The same problems are present in another work that tries to implement CEP on top of Kafka Streams [4]. Again, pattern recognition is performed in a single topology node, drastically limiting its scalability. Furthermore, the absence of a `followedBy` operator renders the library unable to correlate heterogeneous flows of data.

KEPLr's operators are implemented as isolated topology nodes, presenting the possibility to scale-out the computation with higher granularity. On the other hand, heterogeneous joins can address different incoming streams and their correlations.

VII. Conclusion

In this work, we presented our solution for performing CER on top of a distributed, scalable streaming platform like Kafka Streams. We addressed the problem by presenting an extension for the Dual Streaming Model. In particular, we introduced the concept of *type* and *events*, i.e., typed records. Then, we provided the definition of *event stream* together with a set of related operators. Finally, we presented a proof-of-concept implementation[5] based on the extended model. In particular, we introduced the *EType* and *KTStream* classes, respectively based on the *type* and *event stream* concepts.

[4]https://github.com/fhussonnois/kafkastreams-cep

[5]KEPLr https://github.com/semlanghi/kEPLr

Our future works on KEPLr include but are not limited to: (i) a systematic and more detailed evaluation performed through a comparison with Esper [14] using the COST model [16], (ii) a KEPLr extension based on Interval-Based Complex Event Recognition which includes Allen's Algebra's main operators, and a (iii) processing model extension including an inverse match operation, in order to tackle the isolation of the event stream entity compared to the rest of the model.

Both the second and third points are not trivial and may involve an extensive redesign of the proof-of-concept implementation of the system.

References

[1] D. Laney, "3d data management: Controlling data volume, velocity and variety," *META group research note*, vol. 6, no. 70, p. 1, 2001.

[2] G. Cugola and A. Margara, "Processing flows of information: From data stream to complex event processing," *ACM Computing Surveys (CSUR)*, vol. 44, no. 3, pp. 15:1–15:62, Jun. 2012.

[3] M. Armbrust, T. Das, J. Torres, B. Yavuz, S. Zhu, R. Xin, A. Ghodsi, I. Stoica, and M. Zaharia, "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*, G. Das, C. M. Jermaine, and P. A. Bernstein, Eds. ACM, 2018, pp. 601–613. [Online]. Available: https://doi.org/10.1145/3183713.3190664

[4] M. Hirzel, G. Baudart, A. Bonifati, E. Della Valle, S. Sakr, and A. Vlachou, "Stream processing languages in the big data era," *SIGMOD Rec.*, vol. 47, no. 2, pp. 29–40, 2018. [Online]. Available: https://doi.org/10.1145/3299887.3299892

[5] R. Tommasini, S. Sakr, E. Della Valle, and H. Jafarpour, "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*, A. Bonifati, Y. Zhou, M. A. V. Salles, A. Böhm, D. Olteanu, G. H. L. Fletcher, A. Khan, and B. Yang, Eds. OpenProceedings.org, 2020, pp. 643–646. [Online]. Available: https://doi.org/10.5441/002/edbt.2020.84

[6] E. Alevizos, A. Skarlatidis, A. Artikis, and G. Paliouras, "Probabilistic complex event recognition: A survey," *ACM Comput. Surv.*, vol. 50, no. 5, pp. 71:1–71:31, 2017. [Online]. Available: https://doi.org/10.1145/3117809

[7] N. Giatrakos, E. Alevizos, A. Artikis, A. Deligiannakis, and M. Garofalakis, "Complex event recognition in the Big Data era: a survey," *The VLDB Journal*, Jul. 2019.

[8] M. J. Sax, G. Wang, M. Weidlich, and J.-C. Freytag, "Streams and Tables: Two Sides of the Same Coin," in *Proceedings of the International Workshop on Real-Time Business Intelligence and Analytics - BIRTE '18*. Rio de Janeiro, Brazil: ACM Press, 2018, pp. 1–10.

[9] A. Arasu, S. Babu, and J. Widom, "The CQL continuous query language: semantic foundations and query execution," *The VLDB Journal*, vol. 15, no. 2, pp. 121–142, Jun. 2006.

[10] G. Wang, J. Koshy, S. Subramanian, K. Paramasivam, M. Zadeh, N. Narkhede, J. Rao, J. Kreps, and J. Stein, "Building a replicated logging system with apache kafka," *Proc. VLDB Endow.*, vol. 8, no. 12, pp. 1654–1655, 2015. [Online]. Available: http://www.vldb.org/pvldb/vol8/p1654-wang.pdf

[11] D. Luckham, "The power of events: An introduction to complex event processing in distributed enterprise systems," in *Rule Representation, Interchange and Reasoning on the Web, International Symposium, RuleML 2008, Orlando, FL, USA, October 30-31, 2008. Proceedings*, ser. Lecture Notes in Computer Science, N. Bassiliades, G. Governatori, and A. Paschke, Eds., vol. 5321. Springer, 2008, p. 3. [Online]. Available: https://doi.org/10.1007/978-3-540-88808-6_2

[12] A. Artikis, M. J. Sergot, and G. Paliouras, "An Event Calculus for Event Recognition," *IEEE Trans. Knowl. Data Eng.*, vol. 27, no. 4, pp. 895–908, 2015.

[13] G. Cugola and A. Margara, "TESLA: a formally defined event specification language," in *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems, DEBS 2010, Cambridge, United Kingdom, July 12-15, 2010*, J. Bacon, P. R.

Pietzuch, J. Sventek, and U. Çetintemel, Eds. ACM, 2010, pp. 50–61. [Online]. Available: https://doi.org/10.1145/1827418.1827427

[14] "Esper." [Online]. Available: http://www.espertech.com/esper/

[15] "FlinkCEP." [Online]. Available: https://ci.apache.org/projects/flink/flink-docs-stable/dev/libs/cep.html

[16] F. McSherry, M. Isard, and D. G. Murray, "Scalability! But at what COST?" in *15th Workshop on Hot Topics in Operating Systems, HotOS XV, Kartause Ittingen, Switzerland, May 18-20, 2015*, G. Candea, Ed. USENIX Association, 2015. [Online]. Available: https://www.usenix.org/conference/hotos15/workshop-program/presentation/mcsherry