Noname manuscript No.
(will be inserted by the editor)

Complex Event Recognition in the Big Data Era: A Survey

Nikos Giatrakos · Elias Alevizos · Alexander Artikis · Antonios Deligiannakis · Minos Garofalakis

Received: date / Accepted: date

Abstract

The concept of event processing is established as a generic computational paradigm in various application fields. Events report on state changes of a system and its environment. Complex Event Recognition (CER) refers to the identification of composite events of interest, which are collections of simple, derived events that satisfy some pattern, thereby providing the opportunity for reactive and proactive measures. Examples include the recognition of anomalies in maritime surveillance, electronic fraud, cardiac arrhythmias and epidemic spread. This survey elaborates on the whole pipeline from the time CER queries are expressed in the most prominent languages, to algorithmic toolkits for scaling-out CER to clustered and geo-distributed architectural settings. We also highlight future research directions.

1 Introduction

CER systems accept as input a stream of time-stamped 'simple, derived events' (SDEs). These are the result of applying a computational derivation process to some other event, such as a measurement coming from a sensor. Using SDEs as input, CER systems identify complex events (CEs) of interest, which are collections of events that satisfy some pattern [35, 72, 73]. The pattern of a CE imposes temporal and, possibly, atemporal constraints on its sub-events. Consider, for example, maritime surveillance, where CER allows for expressing patterns that attach meaning to fused, streaming event tuples for the real-time detection of suspicious or potentially dangerous situations that may have a serious impact on the environment and safe navigation at sea [130]. Simple position signals are continuously collected from hundreds of thousands of ships worldwide. Vessels report their position at different time scales, while the messages are often noisy or even contradictory. Vessel movement is combined with static geographical information while, for effective vessel identification and tracking, additional data sources are taken into account, such as weather reports and satellite images.

CER significantly differentiates itself from traditional streaming conceptualizations [77]. Classical stream querying leaves to the clients the responsibility of attaching meaning to the result set. On the contrary, CER languages allow querying for complex patterns that match incoming events on the basis of their content, sequencing and ordering relationships as well as other spatio-temporal constraints. Additionally, CER systems incorporate strategies for selecting (and consuming) events to derive composite ones [73].

This survey is built on three pillars. The first pillar (Section 2) concerns CER languages. There are various CER systems and languages that have been proposed in the literature. These systems have a common goal, but differ in their architectures, data models, pattern languages and processing mechanisms. Their comparative assessment is further hindered by the fact that many of the techniques have been developed in different communities, each bringing in their own terminology and view on the problem. The current survey presents a unified view of CER languages. Our focus is

N. Giatrakos, A. Deligiannakis, M. Garofalakis
Athena Research and Innovation Center, Greece
Technical University of Crete, Greece
E-mail: {ngiatrakos,adeli,minos}@imis.athena-innovation.gr, {ngiatrakos,adeli,minos}@softnet.tuc.gr

Elias Alevizos,
National and Kapodistrian University of Athens, Greece
National Centre for Scientific Research Demokritos, Greece
E-mail: ilalev@di.uoa.gr, alevizos.elias@iit.demokritos.gr

Alexander Artikis,
University of Piraeus, Greece
National Centre for Scientific Research Demokritos, Greece
E-mail: a.artikis@unipi.gr, a.artikis@iit.demokritos.gr

on the formal methods for CER developed in the database, distributed systems and artificial intelligence communities.

The recent development of Big Data platforms, such as Apache Storm [8], Spark [7] and Flink [1], has made it simpler to design and build distributed processing pipelines. However, these platforms are not sufficient on their own for enabling CER. We will show how CER languages may be used with Big Data platforms. The second pillar of our survey (Sections 3-4) is thus on scaling-out CER, moving from centralized architectures to clustered settings prevalent in private data centers or public and hybrid clouds. Therefore, we will present parallel CER approaches for dealing with the volume of Big event Data and also focus on elastic (adaptive) CER to handle volatile velocity and event data distributions.

The third pillar overviews CER applications at vast scale, which operate over a set of *geo-distributed* sites (clusters, sensors, IoT and other smart devices) each accumulating event streams which later need to be efficiently synthesized to provide holistic answers to continuous queries. In this case, communication scalability issues are of additional essence. On the one hand, *in-situ processing* places local filters in the event sources to reduce communication. On the other hand, *in-network processing* pushes the evaluation of network-wide query operations to sites that are near the relevant event sources so that relevant events are synthesized early, extracting only compact aggregated information to be further forwarded in the network.

The first key survey of CER systems was presented by Cugola and Margara [61]. The second one focused on the lack of Veracity of Big Data [27], where systems that can handle uncertainty were presented. For example, Alevizos et al. [27] identified two major classes of methods for probabilistic CER: automata-based systems (e.g., [102], [150], [161], [138]) and logic-based ones (e.g., [145], [43], [151], [63]). Other examples are [112], using fuzzy set theory, and [149], using Petri Nets for recovery of missing events. In this survey we do not elaborate on uncertainty handling in CER. Our survey is thus complementary to [61] and [27]. We focus on efficiently handling the volume, velocity and geographic distribution of Big Data in CER, while at the same discussing the expressivity of formal CER languages.

2 Languages for Complex Event Recognition

Numerous CER systems and languages have been proposed in the literature. These systems have a common goal, but differ in their architectures, data models, pattern languages, and processing mechanisms. For example, many CER systems provide users with a pattern language that is later compiled into some form of automaton. The automaton model is generally used to provide the semantics of the language and/or as an execution framework for pattern matching. Apart from automata, some CER systems employ tree-based models. Again, tree-based formalisms are used for both modeling and recognition, i.e., they may describe the complex event patterns to be recognized as well as the applied recognition algorithm. Recently, logic-based approaches to CER have been attracting considerable attention, since they exhibit a formal, declarative semantics, and at the same time have been proven efficient enough for Big Data applications. Our goal in this section is to provide an overview of the various languages that have been proposed, determine the regions of their convergence and divergence, and establish requirements that they should satisfy in terms of their expressive power (for a short tutorial on CER languages see also [34]). Note that it is not our intention to present a survey of CER systems, since this is covered elsewhere [61]. Our focus is on identifying the classes of languages typically used in CER in order to determine their expressive power and limits, and to initiate a discussion about how expressive power may interact with the performance exhibited by a CER system. This discussion regarding performance is taken up again fully in Sections 3 - 5. In this section, we begin by presenting a set of language features and notions usually encountered in CER systems, as well as a set of extra requirements that we deem should be satisfied by such systems but have attracted less attention thus far. We then present the three classes of CER systems that we have identified: automata-based systems, logic-based ones and those that employ trees. Note that there also exist hybrid systems, albeit these constitute a minority. We briefly mention those as well.

2.1 Setting the scene

Due to the great variety of existing CER languages and systems, there is a lack of a common ground for comparing them, and extracting a set of common operators is far from being a trivial task. Notwithstanding this disparity, it seems that it is indeed possible to identify some basic features that should be present in every CER language; in fact, as of late there have appeared attempts targeting such a unification and homogenization [88, 85]. Therefore, before delving into the presentation of the languages themselves, we begin by presenting these common features and establishing a set of requirements. In Section 2.1.1 we discuss the basic operators that constitute the building blocks of a CER language, borrowing from [27]. Our contributions may be found in Sections 2.1.2 – 2.1.5, where we discuss some extra features that are common in CER systems and conclude with a discussion on a set of functionality requirements that are less frequently satisfied.

2.1.1 Abstract Event Algebra

A CER system takes as input a stream of events, also called simple derived events (SDEs), along with a set of patterns,

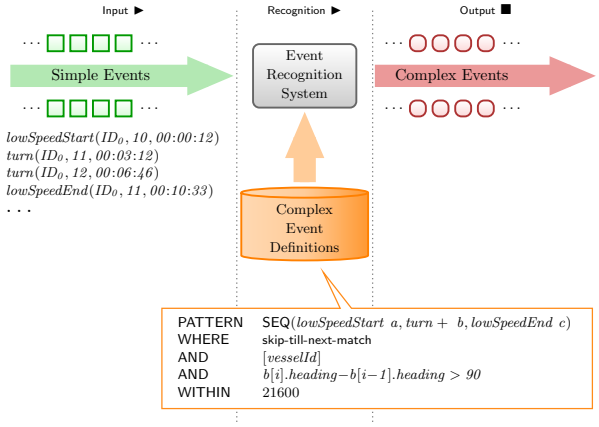


Fig. 1: High-level view of a CER system, using the maritime domain as an example. The simple event stream consists of *turn*, *lowSpeedStart* and *lowSpeedEnd* events from a vessel with id ID_0. The pattern attempts to capture a sequence of events, where the first indicates that a vessel starts moving at a low speed, then the vessel performs one or more turns and finally ends by a single event indicating the end of the slow movement.

defining relations among the SDEs, and detects instances of pattern satisfaction, thus producing an output stream of complex events [117,73]. Since time is of critical importance for CER, a temporal formalism is used in order to define the patterns to be detected. Such a pattern imposes temporal (and possibly atemporal) constraints on the input events, which, if satisfied, lead to the detection of a CE.

Typically, an event has the structure of a tuple of values which might be numerical or categorical. The most important attributes which are always to be found in an event are those of *Event Type* and *timestamp*. The timestamp may be a single timepoint, indicating the occurrence time of the event, or an interval, in cases where events may be durative. These two basic attributes may be accompanied by any number of extra attributes. As an example, consider the domain of maritime monitoring where the input stream consists of events emitted from vessels sailing at sea and relaying information about their kinematic behavior, e.g., location, speed, heading, etc [130]. In the terminology of the maritime domain, these are called AIS (Automatic Identification System) messages. Each such SDE may contain an event type referring to the type of movement executed by a vessel (e.g., turning, sailing, accelerating) and a timestamp. Additionally, it may contain a number uniquely identifying each vessel (an identifier) along with attributes for its longitude, latitude, speed, etc. Figure 1 depicts a high-level view of a CER system, using the maritime domain as an example.

In order to define the CEs to be detected upon the stream of SDEs, we need a language that is expressive enough for the needs of CER. The most basic operator is that of *selection* according to a set of predicates. This set of selection predicates are applied to every SDE and those SDEs that do not satisfy the predicates are filtered out. An example of a selection operator on the AIS messages could be one that checks the speed of each vessel and retains only those messages with a speed above 0.1 knots in order to keep only those vessels that are actually on the move. As far as the temporal operators are concerned, the most basic is the *sequence* operator, usually denoted by a semicolon. The implied constraint in this case is that the events connected through a sequence operator must succeed one another temporally. These operators are sufficient to define simple patterns, but for more complex patterns, we need to incorporate some more operators. With the help of the theory of descriptive complexity, recent work has identified those constructs of an event algebra which strike a balance between expressive power and complexity [162]. For other event formalisms, see also [76,85,105,49,28,29].

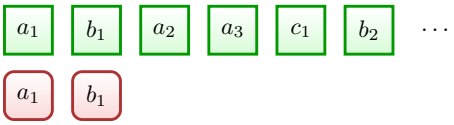
These constructs may be summarized as follows:

- *Sequence*: Two events following each other in time.
- *Disjunction*: Either of two events occurring, regardless of their temporal relation.
- *Iteration*: An event occurring N times in sequence, where $N \geq 0$.
- *Conjunction*: Both events occurring, regardless of their temporal relation.
- *Negation*: Absence of event occurrence.
- *Selection*: Select those events whose attributes satisfy a set of predicates/relations, temporal or otherwise.
- *Projection*: Return an event whose attribute values are a possibly transformed subset of the attribute values of its sub-events.
- *Windowing*: The event pattern must occur within a specified time window.

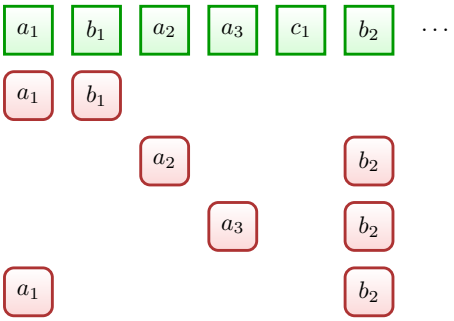
The above list can be presented in the form of a simple event algebra, as presented below [27]

$$
\begin{aligned}
ce ::= \; & sde && \mid \text{Base case} \\
& ce_1 \; ; \; ce_2 && \mid \text{Sequence} \\
& ce_1 \vee ce_2 && \mid \text{Disjunction} \\
& ce^* && \mid \text{Iteration} \\
& ce_1 \wedge ce_2 && \mid \text{Conjunction} \\
& \neg \, ce && \mid \text{Negation} \\
& \sigma_\theta(ce) && \mid \text{Selection} \\
& \pi_m(ce) && \mid \text{Projection} \\
& [ce]_{T_1}^{T_2} && \mid \text{Windowing (from } T_1 \text{ to } T_2\text{)}
\end{aligned}
\tag{1}
$$

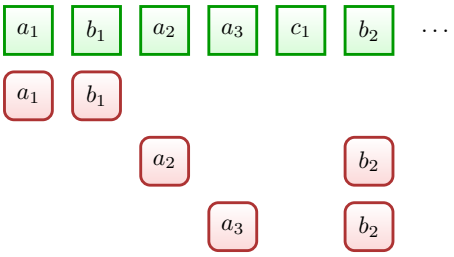
where $\sigma_\theta(ce(v_1, \ldots, v_n))$ *selects* those ce whose variables v_i satisfy the set of predicates θ and $\pi_m(ce(a_1, \ldots, a_n))$ *returns* a ce whose attribute values are a possibly transformed subset of the attribute values of a_i of the initial ce, according to a set of mapping expressions m. Note that *conjunction*



(a) Matches under strict-contiguity.



(b) Matches under skip-till-any-match.



(c) Matches under skip-till-next-match.

Fig. 2: Example of selecting input events for the pattern $R = a; b$ under different selection policies. The top stream (green rectangles) represents the input stream. The bottom streams (red, rounded rectangles) represent the matches produced, one per row.

may also be written by combining sequence and disjunction, as: $ce ::= (ce_1; ce_2) \vee (ce_2; ce_1)$. Please note that, compared to the event algebra presented in [27], we have explicitly added the conjunction operator, as it is not only important, but it may also require special handling and cannot always be derived from the other operators when there is no support for disjunction. This inductive definition showcases an important feature of CER languages: their compositionality, i.e., the ability to define hierarchies of events, where SDEs may be used to define some CEs and these may again be used to define other, higher-level CEs.

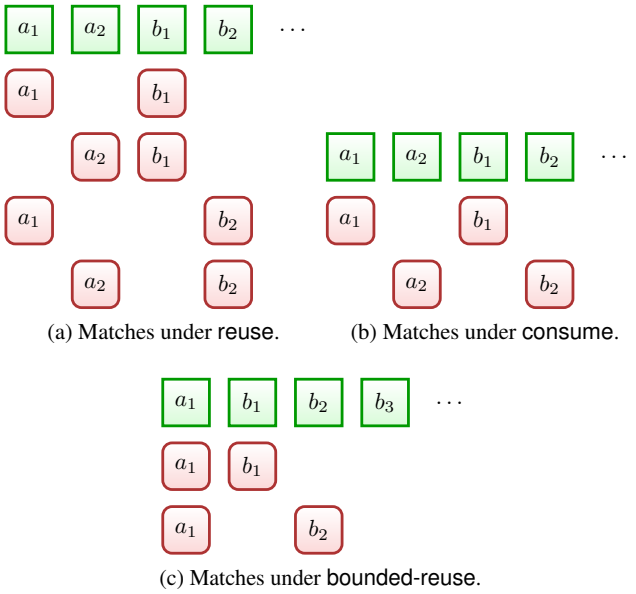
2.1.2 Selection Policies

The first three temporal operators in the event algebra presented above (Eq. 1), namely sequence, disjunction and iteration, resemble the three operators of standard regular expressions: concatenation, union and Kleene star respectively. This similarity is not a coincidence, as automata have frequently been used as computational models in CER. Since SDEs are not symbols but tuples, the automata variations employed in CER typically have predicates on their transitions: the predicates of the selection operators. By following the semantics of regular expressions, one would then expect that the SDEs involved in a match of a CER pattern should occur contiguously in the input stream. As an example, consider the case where we have a stream with event types a, b or c (for simplicity we ignore all other attributes) and we define a simple pattern as $a; b$ (an event of type a followed by one of type b). Figure 2a shows an example of such a stream along with the match that would be detected.

However, in CER it is often the case that we are also interested in matches where the involved SDEs need not be contiguous. This is where the notion of *selection policies* enters the scene [61, 162, 163]. As its name suggests, a selection policy determines which SDEs may be allowed to enter a match by establishing conditions about whether we are allowed to skip any events, deemed as "irrelevant", or not. The single match of Figure 2a is the result of our pattern under one such policy, called strict-contiguity, due to its requirement that all events in a match must be contiguous in the input stream. This is indeed the strictest policy in the sense that it produces the fewest matches. On the other end of the strictness spectrum is the so called skip-till-any-match policy. In this case, any combination of events that satisfy the succession constraints of the pattern, regardless of whether they are contiguous or not, is considered a match. This policy is closer in spirit to logic programming where running a query/goal returns all results that satisfy it. Figure 2b depicts the matches of our example stream for the pattern $a; b$ under the skip-till-any-match policy.

In between these two extremes, there exists the skip-till-next-match policy. In this case, it is still possible to skip irrelevant events, e.g., a c event occurring between an a and a b, but only the immediately next relevant event in the stream is selected, thus restricting the number of matches with respect to the skip-till-any-match policy. Figure 2c shows the matches produced in our example under skip-till-next-match. The match $\{a_1, b_2\}$ of skip-till-any-match is no longer a match, since the partial match that started with a_1 selected b_1 and then capitulated.

Another common policy is the so-called partition-contiguity policy, where the stream is first partitioned into substreams according to a predicate and then strict-contiguity is applied to each substream separately. For example, for the maritime domain this could be useful in order to partition the stream according to the vessel identifier so that each vessel has its own stream and a pattern may be applied to each individual vessel. The same partitioning technique can also be applied to any of the previous three selection policies. In the field of runtime verification a similar technique is used, under the name of "parametric trace slicing" [53]. Within the field of CER itself, such partitioning schemes may be subsumed under the notion



(a) Matches under reuse. (b) Matches under consume.

(c) Matches under bounded-reuse.

Fig. 3: Example of consuming input events for $R = a; b$ under different consumption policies, with skip-till-any-match as the selection policy. The top stream (green rectangles) represents the input stream. The bottom streams (red, rounded rectangles) represent the matches produced, one per row. Note that for Figure 3c the input stream is slightly different.

of *contexts* [163,73]. A context may be defined as a specification of conditions that groups events together for purposes of common processing where each event is assigned to one or more context partitions [73]. As a result, partition-contiguity may not necessarily be viewed as a separate selection policy, but as a combination of a selection policy (strict-contiguity) with a partition/slicing/context scheme.

As a closing remark to this section, we should note that there is no universal consensus about the semantics of selection policies. The discussion of this section borrows the terminology and semantics of the SASE CER engine [21, 162]. FlinkCEP [2], a CER engine built on top of the Flink distributed processing engine [1], uses very similar notions for selecting events, but with different semantics in some cases. We will discuss this issue in more detail in the following sections.

2.1.3 Consumption Policies

There is yet another notion for determining which events may participate in a match: that of *consumption policies* [89,73, 163,61]. A consumption policy determines whether an event that has participated in match of a pattern R is allowed to participate again in other matches of R. In what follows, we adopt the terminology of [73].

The most relaxed consumption policy is called the reuse policy. As its name suggests, under this policy events may be used without any restrictions to any number of matches, provided that they satisfy the constraints of the pattern and of the selection policy. Figure 3a shows an example of the matches produced for the pattern $a; b$ under the reuse consumption policy and the skip-till-any-match selection policy.

The strictest consumption policy is called consume. Under consume, whenever an event becomes member of a match it is no longer allowed to be included in any future matches. Figure 3b shows an example of this policy. Upon the arrival of b_1, the candidate matches are $\{a_1, b_1\}$ and $\{a_2, b_1\}$. Assuming that the production of the matches starts from the one whose initiator (a) is temporally first (for other options, see [163]), then $\{a_1, b_1\}$ is produced. This has two effects: a) b_1 becomes no longer available and $\{a_2, b_1\}$ is disqualified as a match; b) a_1 also becomes invalidated, thus disqualifying $\{a_1, b_2\}$ as a match when b_2 arrives at the next timepoint. We are therefore left with two matches after b_2: $\{a_1, b_1\}$ and $\{a_2, b_2\}$.

An intermediate policy is the one called the bounded-reuse consumption policy. The goal of this policy is to allow the reuse of events, but to impose an upper bound on the number of matches in which an event may participate. Figure 3c shows the matches produced under this policy when at most 2 matches are allowed. Assuming again the production of matches follows a temporal order, the last match that would normally be produced under reuse, $\{a_1, b_3\}$, is dropped since 2 matches with a_1 have already been produced.

2.1.4 Windows

CER systems are not expected to detect CEs by considering at every timepoint all SDEs that have occurred in the past. In order to limit their search space, which can quickly become unmanageable, especially when relaxed selection policies and consumption policies are used, they typically incorporate a special operator, that of *windowing* [73,61]. Windows are usually applied on a per pattern basis and their function is to restrict, up to a certain point in the past, the SDEs to be considered. Although they can in principle be subsumed as a constraint built from the standard operators of an event algebra (by restricting the time difference between the last and first events in a match), they are usually defined as an extra operator, due to their importance and their effect on the complexity of pattern evaluation.

The most typical window constraint to be found in a pattern is of the form $within(W)$, usually appended at the end. A key distinction between window types is the one between time-based and count-based (also called tuple-based) windows. Time-based windows impose a constraint on the size of the time interval into which a match can extend. The constraint imposed is that a (temporally ordered) candidate match $M = \{e_1, \ldots, e_n\}$ is indeed a valid match if $e_n.timestamp - e_1.timestamp < W$. This basically constitutes a sliding window of length W whose step (slide)

is equal to the temporal resolution of the CER system. On the other hand, tuple-based windows impose an explicit constraint. In this case, a constraint like $within(W)$ cannot be directly expressed through the operators of an event algebra; it implies that only the last W SDEs that have arrived at the system are to be considered, regardless of their timestamps. Some CER systems also include other window types, like tumbling windows [50].

From an implementation point of view, we may also distinguish between actual and logical windowing mechanisms. With actual windows, events belonging to a window are buffered and their processing begins as soon as the window's timer has expired, for time-based windows, or the count limit has been reached for count-based ones. When logical windowing is used, the SDEs are not buffered, but are processed as soon as they arrive.

2.1.5 Requirements

Our discussion thus far has focused on a basic core of operators and features for a CER language. We conclude this section by adducing a set of extra requirements for CER, extracted from the limitations of the core features.

Support for both instantaneous and durative events: The majority of CER languages make the assumption that the timestamp of each event, either SDE or CE, is a single timepoint. However, there are domains where it is more natural to express events as having a temporal duration. For example, in human activity recognition, the activity of a person walking is durative. The same holds in the maritime domain for several types of vessel behavior, such as fishing. Sometimes, the interval of an activity may be open, in the sense that it is still ongoing. For example, we should not wait until the end of fishing before we report it. Additionally, there are some subtle issues with respect to the semantics of instantaneous events. When single timepoints are used as timestamps, it is possible that some unintended semantics might be introduced [76, 128] (see also Sections 2.4 and 2.6). Note also that formalisms for reasoning on durative events have appeared in the past, such as the Event Calculus [105, 49] and Allen's Interval Algebra [28, 29], and have been used for defining event algebras (e.g., [129, 35]).

Support for relational events: By relational events we mean CEs whose detection depends on multiple entities of the domain under study. In the maritime domain, detecting a possible collision requires to relate the activity of at least two vessels. It is possible to detect such relational CEs by partitioning the input stream according to its entities and attempt to join these substreams. Such joins raise new issues with respect to the runtime complexity of CE patterns. For instance, relational events are not easy to be expressed and captured with simple computational models, like simple automata or even extensions of automata often used in CER, which often assume that only a single stream exists. Therefore, more expressive models are required, like quantified event automata, used for runtime verification [40].

Support for concurrency constraints: Sequence is one of the basic operators in CER and this is the reason why automata are so popular as computational models for CER. On the other hand, there exist patterns that require a mechanism for detecting concurrent events, especially in the case of relational CEs. The above mentioned example of collision detection is such a case, since a pattern for it would need to relate the behavior of two vessels at the same time.

Support for patterns without windows: As already mentioned, windows are essential in CER since they significantly reduce the search space for pattern matching. Although they might also be useful from a conceptual point of view as well (there are indeed patterns which are meaningless if they extend beyond a time interval), it is also the case that some long-term relationships are important to capture as a CE, without knowing beforehand their maximum temporal duration. For example, a pattern for detecting when a vessel approaches a port cannot be meaningfully constrained in terms of its duration, since different vessel types exhibit different movement patterns (or the window would have to be so large in order to include all cases, rendering it essentially meaningless). Although it is conceptually and semantically possible to get rid of the window constraint, this would incur a heavy performance cost, considering that the size of the window is one of the main factors affecting runtime complexity (the larger the window the more partial matches that need to be maintained) [162].

Support for event hierarchies: We have already mentioned that it is important to be able to define CEs hierarchically, i.e., using lower-level CEs to define other CEs at higher levels. Hierarchies allow for structured, succinct representations, and thus code maintenance. We repeat this requirement here, since it is not always satisfied. Event hierarchies raise issues both with respect to the semantics of CER languages and the performance of CER systems. For example, CER systems that are based on automata resemble in certain respects register automata [98], i.e., automata that are equipped with registers in order to store past elements of a stream and later be able to retrieve them for comparison purposes. However, register automata are not closed under complement, which implies that it is not obvious how the negation operator is to be properly used in a CER language that uses automata as its underlying model. With respect to performance, a hierarchy of CEs might exhibit a structure where CEs might participate in the definition of multiple other CEs. Treating such hierarchies in a naive manner would result in redundant computations. For hierarchies to be a viable feature, careful optimizations should be employed [114, 35].

Support for background knowledge and non-temporal reasoning: Most CER systems focus on temporal reasoning

and their selection predicates are usually (in)equalities on event attributes or aggregate functions, like *averaging* of a certain event attribute, when iteration is present. Useful as this kind of reasoning might be, there are also cases where we need to take into account information that is not present in the SDEs themselves. For example, we might need to know whether it is prohibited to fish within a specific area at sea in order to detect vessels that violate this restriction. Therefore, it is important for a CER system to be able to incorporate background knowledge (e.g., areas where fishing is prohibited) and to perform non-temporal reasoning as well (e.g., that a position lies within a given area polygon).

2.2 Automata–based Systems

Since the temporal operators of CER languages (sequence, disjunction, iteration) resemble those of regular expressions, it is no surprise that automata have been quite popular as computational models of CER systems. In such systems, patterns are usually defined in a language similar to SQL, with the addition of operators for the regular part of the pattern. This regular part typically appears first, followed by a set of predicates that the events appearing in the regular part must satisfy. After a pattern has been defined, it goes through a compiler that transforms it to an automaton (often non-deterministic). The automaton is then fed with the stream of SDEs, changing states according to whether predicates on the outgoing transitions of its current state are satisfied. Upon the triggering of a transition, the responsible SDE may be ignored if it is irrelevant or stored if it is relevant. Whenever it reaches a final state, we say that a full match has been completed and a CE is detected. The user is then informed of the occurrence of the CE, along with the SDEs participating in the match. When the automaton is in a non-final state, we say that its stored SDEs constitute a "partial match" that may or may not eventually lead to a full match.

Automata-based CER systems started as academic projects (e.g., Cayuga [66,67], SASE [157,87,21,162] and its derivatives, like SASE+ [68], NextCEP [143], DistCED [133]), inspired to some extent by previous work on Data Stream Management Systems, such as TelegraphCQ [52], CQL [32] and CQL's commercial derivative, Oracle CQL [16]. Work along these lines still continues to this date [26], with effort being also devoted to providing solid foundations [85]. Automata-based CER has recently reached a new level of maturity and found its way into systems, such as Flink, which provides a library for complex event recognition and processing, called FlinkCEP [2]. In what follows, we will focus mainly on the popular FlinkCEP and the highly cited SASE, which are both open source. The general ideas are very similar in all systems, though.

SASE is a CER engine which translates patterns into non-deterministic automata, acting as its computational model. As far as its input is concerned, it assumes that each SDE is represented by an event type, a timestamp in the form of a single timepoint and any other extra attributes. All SDEs are also assumed to be in a single stream (in the case of multiple streams, these must first be merged into a single stream) and this stream is temporally ordered. A CE pattern is defined through a SQL-like language, whose purpose is to detect SDEs occurring in the specified temporal order and satisfying any extra constraints (possibly atemporal) in the form of predicates. The pattern of Figure 1 is an example of a SASE pattern as applied to the maritime domain. The PATTERN clause (line 1) captures a sequence of events, where the first indicates that a vessel starts moving at a low speed, then executes one or more turns and finally ends by a single event indicating that the slow movement has ended. The WHERE clause (lines 2–4) does three things: 1) determines the selection policy as skip-till-next-match; 2) partitions the stream according to the id of the vessels so that the pattern is applied to each vessel individually; 3) imposes an atemporal constraint on all the turn events so that the heading of each turn event differs by more than 90 degrees from the heading of the previous turn event. Finally, the WITHIN constraint (line 5) requires that the pattern happens within 21600 seconds (6 hours). This could be a simple pattern detecting zig-zag maneuvers, typical for vessels while trawling.

Figure 4 shows the automaton that would be constructed from this pattern. The PATTERN clause is first used to build the structure of the automaton and subsequently the WHERE clause determines the predicates that must be placed on the transitions. This automaton is non-deterministic as multiple outgoing edges from a state may evaluate to TRUE at the same time (upon reading the same event). Non-determinism essentially implies that, when an automaton can follow more than one edges, it must be cloned, i.e., a new run must be created, with each run following a different "trajectory" from that point on. When a run of an automaton follows a transition denoted by a solid line – see Figure 4 – the triggering event is stored in a buffer as being relevant for a possible future complete match. When a transition denoted by a dashed line is followed, the triggering event is considered as irrelevant and is not stored. The multiple runs created due to non-determinism (which can be present even with strict contiguity) and the buffers storing the relevant events constitute the main runtime bottlenecks. These bottlenecks become more pronounced when skip-till-any-match is used and when the pattern includes iteration operators [162].

SASE can accommodate all operators (even negation) mentioned in Section 2.1.1 and all selection policies mentioned in Section 2.1.2. On the other hand, its operators are not fully compositional, e.g., nesting of iterations is not allowed. In theory, it also allows for event hierarchies through another clause, called RETURN, appended at the end of patterns and acting as a projection operator. However, its pub-

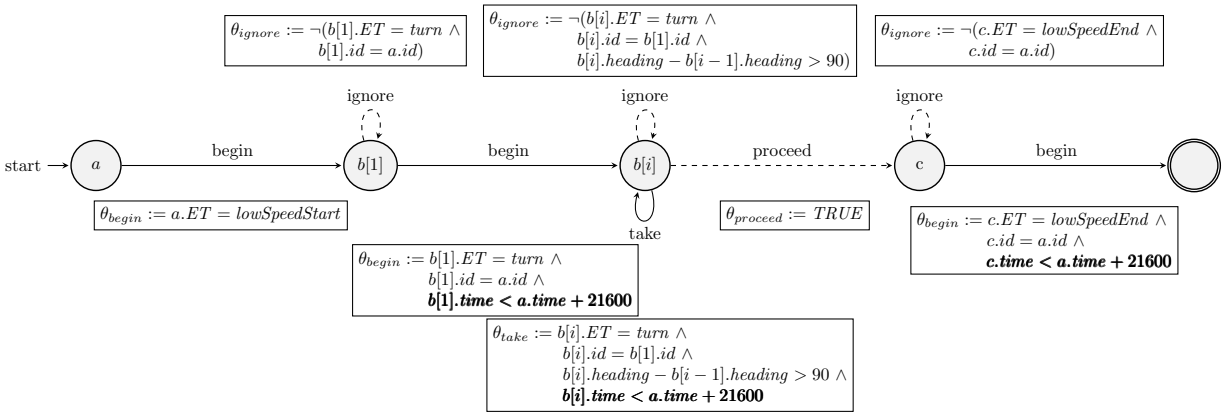


Fig. 4: A non-deterministic automaton (NFA), as constructed by SASE, for the pattern of Figure 1. *begin* edges move the NFA to a next state and store the SDE. *take* edges iterate over the same state, again storing the SDEs. *ignore* edges skip "irrelevant" events due to the use of skip-till-next-match. The *proceed* takes the NFA out of the iteration. Above or below each edge, its respective constraints are shown as a (negated) conjunction of predicates found in the pattern. Conjuncts shown in bold correspond to the window constraint. Note that the window constraint is placed on multiple edges in order to be able to "kill" an instance of the NFA as soon as possible if the constraint is violated, without waiting until it reaches the last non-final state (c). ET stands for $EventType$.

licly available source code does not include this functionality [17]. As far as the allowed consumption policies are concerned, only reuse is available. Finally, the windows in SASE are strictly time-based and constitute logical views.

FlinkCEP [2] is close in spirit to SASE. It also employs non-deterministic automata, equipped with predicates on their transitions. From a language perspective, one important difference to SASE is that, strictly speaking, it does not have a language for defining patterns. Instead, the user is required to write a pattern in Java or Scala, as shown in Listing 1, where we define a simple pattern with a vessel starting from an idle status (speed less than 0.1 knots) and then accelerates to more than 15 knots within less than 10 seconds. Writing patterns in such a way is cumbersome and error-prone. On the other hand, the advantage of FlinkCEP is that patterns are compositional. For example, the *idle* pattern in the Listing 1 can be used to define other patterns as well, besides the *abruptStart* pattern.

Listing 1: Example of Scala source code for defining patterns in FlinkCEP

```
// initial pattern to start the sequence
val idle: Pattern[Event, _] =
     Pattern.begin("idle").where(event => event.getSpeed
     < 0.1)

// strict contiguity
val highSpeed: Pattern[Event, _] =
     idle.next("highSpeed").where(event =>
     event.getSpeed > 15)

// relaxed contiguity
val highSpeed: Pattern[Event, _] =
     idle.followdBy("highSpeed").where(event =>
     event.getSpeed > 15)

// non-deterministic relaxed contiguity
val highSpeed: Pattern[Event, _] =
     idle.followedByAny("highSpeed").where(event =>
     event.getSpeed > 15)

// window
val abruptStart: Pattern[Event, _] =
     highSpeed.within(Time.seconds(10))
```

FlinkCEP also accommodates all selection policies (and supports some more, see [2]), albeit with a slightly different terminology: relaxed-contiguity is the equivalent of skip-till-next-match and non-deterministic-relaxed-contiguity the equivalent of skip-till-any-match. However, the semantics of its selection policies do not exactly correspond to those of SASE. For example, it seems that non-determinism is not applicable for strict-contiguity in FlinkCEP, which is at odds with the semantics of this policy in SASE. Consider the pattern $a; b^*; b$ and the simple stream a_1, b_1, b_2. Even with strict-contiguity, this pattern would require a non-deterministic automaton for SASE. After the arrival of b_1, SASE would have two NFA runs: one that would terminate the pattern, move the run to its final state and complete the match, producing $M_1 = \{a_1, b_1\}$; and one that would treat b_1 as belonging to the iteration operator and that would reach its final state after b_2, thus producing another match, $M_2 = \{a_1, b_1, b_2\}$. On the other hand, FlinkCEP would detect only M_1. Inferring the precise semantics of FlinkCEP's selection policies is not trivial, as its internals are not documented and the documentation provides only informal explanations. With respect to consumption policies, reuse and consume are supported (along with some variations), but not bounded-reuse. As with selection policies, there is a difference in terminology: NO_SKIP

is the equivalent of reuse and SKIP_PAST_LAST_EVENT the equivalent of consume.

FlinkCEP can use all operators of our algebra of Section 2.1.1. In fact, it includes several extensions of these operators. For example, quantifiers, as used in regular expressions, are also available, for imposing lower and/or upper bounds on the number of repetitions of an iteration operator. It is also quite flexible with respect to the windows that may be applied to a pattern. Both time-based and count-based windows are available, which can also be either sliding or tumbling. Finally, as in SASE, windows in FlinkCEP are also logical views, which means that SDEs are not buffered in batches (as, for example, in Spark streaming [5]), but they are directly forwarded to the operators of a pattern.

Siddhi [18] is a commercial CER engine with capabilities similar to those of FlinkCEP. It is also based on state machines for pattern matching and can support all operators of our algebra and most of the selection and consumption policies. Contrary to FlinkCEP, Siddhi offers a language for defining patterns.

Cayuga [66,67] is similar to SASE, but a bit earlier and relatively less expressive. It does not support windows, but supports iteration, although, like SASE, does not allow for nested iterations. Due to the absence of windows, avoiding unbounded storage in the presence of iteration is achieved by using an automaton model that stores only the attribute values of the most recent iteration. Cayuga uses skip-till-any-match as its selection and reuse as its consumption policy. As opposed to the systems presented thus far, in order to avoid semantical ambiguities arising when timepoints are used as timestamps [76], Cayuga treats events as durative, with instantaneous events also available as a special case. A sequence operator in Cayuga is satisfied if the involved events are not overlapping, i.e., the end timepoint of an event is smaller than the start timepoint of the next event in the sequence.

NextCEP [143] and DistCED [133] are two other automata-based systems. From a language perspective, they are very similar to Cayuga. They also treat events as durative in order to avoid the semantical issues mentioned above. Their focus is not so much on providing a language with more expressive power, but on optimizing the evaluation of automata for efficient, distributed processing, e.g., by query rewriting. These issues will be discussed in Sections 4 and 5.

2.3 Logic–based Systems

Besides automata-based systems, there exists another significant line of work where a CER system employs a logic-based temporal formalism (see [36] for a survey). In this case, patterns often have the form of a rule, with a head and a body defining the conditions which, if satisfied, lead to the detection of a CE. The semantics are those of the temporal formalism employed. The underlying mechanism for performing inference can vary: from Selective Linear Definite (SLD) resolution used in Prolog-based systems to directed graphs (resembling automata) constructed from the rules.

The Chronicle Recognition System (CRS) is an example of the latter case [71,79,70,72]. A chronicle in CRS terminology is essentially a CE, i.e., a set of events linked together by time constraints and whose occurrence may depend on the context. A chronicle definition resembles a logical rule, having a body and a head. Pattern (2) below presents a simplified definition of the $abruptStart$ pattern in the language of CRS.

$$
\begin{array}{ll}
1 & \text{chronicle } abruptStart[?VesselId](T_2)\ \{ \\
2 & \quad \text{event}(AISMessage[?VesselId, idle], T_1) \\
3 & \quad \text{event}(AISMessage[?VesselId, highSpeed], T_2) \\
4 & \quad T_2 - T_1 \text{ in } [1, 10] \\
5 & \}
\end{array} \tag{2}
$$

Variables start with upper case letters while predicates and constants start with lower case letters, as in logic programming. Prefixing a variable with ? denotes that it is an atemporal variable. A feature of CRS is that a CE, like $abruptStart$, may be defined through multiple rules, thus expressing disjunction. Another feature that CRS supports and is generally lacking in other CER systems is the fact that the subevents in the definition of a CE need not necessarily be totally ordered. For example, line 4 in Pattern (2) could be $T_2 - T_1$ in $[-4, 6]$, indicating that the $highSpeed$ event could precede the $idle$ by at most 4 timepoints. CRS also supports negation and iteration, although an iteration operator must have explicit lower and upper bounds on the number of repetitions. On the other hand, mathematical operators are not allowed as constraints, e.g., stating that the speed of a vessel should be above or below a certain threshold, e.g., as we did in the FlinkCEP pattern shown in Listing 1. Such information must be provided explicitly to the system through preprocessing. Notice, for example, that in lines 2–3 of Pattern 2, we assume that there already exists an attribute in each SDE concerning the speed status of a vessel (whether it is idle or has a high speed).

In order to be evaluated, a CRS pattern is compiled to a Temporal Constraint Network (TCN), i.e., a graph whose nodes correspond to events and edges encode the temporal constraints. This allows CRS to perform both consistency checking on the patterns and apply optimizations by propagating constraints in the graph or even removing them completely if it detects that they are redundant. It is also possible to provide semantics for the CRS language by using colored Petri Nets [55]. The runtime behavior of the CRS system is similar to that of automata-based systems, as instances of a TCN are continuously created and killed, according to whether future events can or cannot satisfy their constraints.

skip-till-any-match and reuse are the default selection policy and consumption policy during evaluation, a fact which can lead to a substantial number of TCN instances being created and maintained. Although CRS does not directly use the notion of selection and consumption policies, it enlists techniques for reducing the number of active TCN instances which are closely related. A pattern may be required to detect chronicles that are not overlapping, a requirement which is a stricter variation of skip-till-next-match. Moreover, two TCN instances of the same pattern may also be forbidden to share events, which essentially corresponds to the consume consumption policy. CRS includes various other optimization techniques, such as "temporal focusing", which re-orders the states of the TCN based on event frequency; such techniques are being used in various contemporary CER approaches [143, 104].

We conclude this section by presenting RTEC (Event Calculus for Run-Time reasoning) [35], a CER engine based on the Event Calculus [105], written in Prolog. The Event Calculus is a logic programming action language that allows for reasoning about events and their effects. RTEC is an implementation of the Event Calculus tailored to event streams, by incorporating a windowing mechanism along with caching and indexing techniques for efficient reasoning. Patterns in RTEC are (locally) stratified logic programs [134]. RTEC patterns are usually written through initiatedAt and terminatedAt rules which determine when a CE starts and ceases to hold respectively. Pattern (3) is an example of a RTEC rule defining the $withinArea$ CE, with which we want to detect intervals during which vessels are inside areas of a specific type, assuming that we also have SDEs (or lower level CEs) about the entrance and exit of vessels in and out of areas.

$$\begin{aligned}&\mathsf{initiatedAt}(withinArea(VesselId) = AreaType,\ T) \leftarrow\\ &\quad \mathsf{happensAt}(entersArea(VesselId, Area),\ T),\\ &\quad typeOf(Area, AreaType).\\ &\mathsf{terminatedAt}(withinArea(VesselId) = \_,\ T) \leftarrow\\ &\quad \mathsf{happensAt}(exitsArea(VesselId, Area),\ T).\end{aligned} \tag{3}$$

RTEC will find all timepoints within a window in which $withinArea$ is initiated, then compute the timepoints in which it is terminated, and finally calculate its maximal intervals by pairing initiating and terminating points. In other words, RTEC assumes that composite activities are subject to the law of inertia , i.e., a CE/fluent continues to hold unless explicitly terminated. Note also that RTEC makes no assumptions on the temporal distance between initiating and terminating points; these may be in different windows.

As is the case with CRS, a CE can have multiple definitions to indicate disjunction. It is also possible to define arbitrary temporal constraints among the timestamps of the subevents of a CE. Typically though, a global actual window (not a logical view) for all patterns is defined in order to restrict the search space of events. Contrary to CRS, besides temporal constraints, it is also possible to include mathematical operators. As a matter of fact, RTEC inherits the full expressive power of logic programming, and thus can naturally handle, among others, arbitrary constraints, relational CEs, and background knowledge. Another feature of RTEC is that it can handle both point-based and interval-based events and has constructs to manipulate time intervals, e.g., through union, intersection and complement.

RTEC does not include an explicit sequence operator. It also does not support iteration, either unbounded or bounded. As is usual with most other CER systems, reuse is the only supported consumption policy. With respect to selection policies, there is a divergence between instantaneous and durative CEs. Instantaneous CEs follow skip-till-any-match whereas durative CEs follow a (deterministic) version of skip-till-next-match, since RTEC has been designed to compute the *maximal* intervals in which a CE is said to take place.

We also need to mention that there are some other logic-based CER systems. For example, PADRES is a distributed publish/subscribe messaging system [110] with fault detection and load balancing capabilities. It supports composite subscriptions (i.e., CEs) by combining low-level events through temporal and logical constraints. PADRES is based on Jess, a rule-based matching engine [15].

2.4 Tree–based Systems

Another line of work on CER is the one that employs trees as a computational model, with ZStream being the prime example [123]. From a language perspective, it is interesting to note that ZStream is very similar to SASE, following the same syntax in most respects. For example, the fishing pattern of Figure 1 would be written in ZStream in an almost identical manner, with the exception of the clause for the selection policy, which would be absent. It is not easy to infer which selection policy ZStream follows, since this information is not directly reported, but we deduce from the provided examples that it must be skip-till-next-match. Thus, the main contribution of ZStream does not lie in offering more expressive power, but in using trees as the underlying computational model, which opens up avenues for optimizations.

ZStream differs from automata-based models in that it assumes that CEs are durative. A sequence operator among CEs is satisfied if the end timepoint of one CE is smaller than the start timepoint of the next CE in the sequence. This allows ZStream to avoid semantical ambiguities when hierarchies of events are present. For example, if $R_1 := a; b$ is a pattern with a window W_1, then each CE detected would typically acquire the timestamp of the last event, in this case of b. Now, if we define another pattern as $R_2 := R_1; c$ with a window W_2, then the intended semantics should be that R_2 is equivalent to $a; b; c$ with the extra constraints $b.timestamp - a.timestamp < W_1$ and $c.timestamp - a.timestamp <$

W_2. However, the expression $R_2 := R_1; c$ could be translated to an automaton violating the second window constraint, since R_1 would have the timestamp of b and not of a. By treating CEs of R_1 as durative, ZStream can avoid such issues.

ZStream translates patterns to trees, whose leaves store SDEs and internal nodes correspond to operators. At the same time, it does not eagerly evaluate a pattern's predicates. Instead, it first collects SDEs into batches and then starts the evaluation. The combination of trees and batch evaluation allows ZStream to follow various physical plans for a given pattern, according to the expected cost of its operators and predicates. Such optimizations could be applied even to the simplest of patterns. For example, for the pattern $a; b$, SASE would create a new NFA run for every a appearing in the stream, even if we knew that b is a rare event. On the other hand, ZStream can follow another plan, by waiting until a b event has arrived and then checking the previously arrived a events that fall within the specified window and simply discarding those that have "expired".

E-Cube [114] is another CER system which employs tree structures, albeit in a different manner than ZStream. With respect to its expressive power, E-Cube supports most of the common CER operators, with the exception of iteration. Its temporal model is based on intervals, with only the SDEs being instantaneous. Its selection and consumption policies cannot be unambiguously determined in the work presenting it, but, by the definition of the sequence operator, we deduce that it probably follows skip-till-any-match and reuse.

The main power of E-Cube lies in its capabilities of multi-query optimizations, i.e., in its ability to evaluate multiple patterns while avoiding redundant and duplicate computations when the patterns under evaluation share some common structure. It achieves this by constructing an event pattern query hierarchy that allows for sharing of subpatterns, thus eliminating redundancy. Additionally, it employs a cost-driven optimizer that can devise an optimal plan in the sense of finding the plan that allows for maximal re-use of intermediate results. Importantly, E-Cube also has elastic properties. It continuously gathers stream statistics and can adapt online to a new execution plan when a drift is detected.

2.5 Hybrid approaches

ZStream is a purely tree-based CER engine. Esper also uses trees for the core of its functionality, like filtering, windowing and aggregations [11]. However, Esper is not easy to classify, since it also uses non-deterministic automata for its "match-recognize" pattern matching functionality, i.e., for the regular part of a pattern. Allen's interval algebra is also used for some of its time methods. This mixture of trees, automata and logic makes Esper patterns quite expressive, but the consequences on the semantics, soundness and completeness are unclear.

The T-Rex system, using TESLA as an event specification language, is an example of a CER system combining logic-based rules with automata [58,59]. T-REX represents a transitional system from automata to logic, as it is not purely logic-based. Its patterns, written in TESLA, have a syntax similar to SASE and they are also translated to automata in order to be evaluated upon an event stream. However, TESLA patterns can be translated to TRIO formulas [80], i.e., to formulas of a first-order logical language that supports temporal operators and has clear semantics in terms of a metric temporal logic. Pattern (4) is an example of a TESLA pattern, capturing the FlinkCEP pattern presented in Listing 1, where we assume that the $Idle$ and $HighSpeed$ CEs have already been defined by other patterns. The sequence operator is defined using the WITHIN clause: it states that a $HighSpeed$ event must follow an $Idle$ event in no more than 10 seconds. Partitioning (by $VesselId$) is denoted through the use of the \$ operator.

```
1 DEFINE AbruptStart()
2 FROM Idle( VesselId = $x)
3 AND last HighSpeed( VesselId = $x)                    (4)
4 WITHIN 10sec from Idle
5 CONSUMING Idle
```

TESLA supports most CER operators except disjunction. It also supports hierarchies of events. With respect to selection policies, it can define patterns with skip-till-any-match, but also makes some other policies available, similar to skip-till-next-match. For example, if we had a stream with two $Idle$ events and one $HighSpeed$, then Pattern (4) would select only the second $Idle$, as denoted by the $last$ keyword in line 3. By using the $first$ keyword, it would select only the first $Idle$ event. Notice that TESLA is one of the few systems that can define different consumption policies. Line 5 in Pattern (4) imposes the consume policy on the pattern.

2.6 Open Issues & Critical Discussion

We conclude our treatment of CER languages with a critical discussion that attempts to pinpoint the weaknesses of the various approaches. For easier reference, we also provide an overview of the discussed systems in Table 1. A question mark in a cell indicates that there are not enough details in the paper(s) describing the relevant system to draw conclusions about the operator/functionality corresponding to the cell's column.

According to our discussion thus far, automata-based CER systems seem to satisfy many of the requirements described in Section 2. Especially solutions, such as FlinkCEP, offer a significant number of extra features, thus providing substantial flexibility for defining patterns. A closer look,

System	σ	π	$\vee$	$\wedge$	$\neg$	;	*	W	H	T.M.	B.K.	S.P.	C.P.	Remarks
Automata														
SASE	✔	✔	✘	✘	✔	✔	✘	Logical	✘	Points	✘	all	Re	$\vee$, $\wedge$ and hierarchies possible in principle but not available in source code.
SASE+	✔	✔	✘	✘	✔	✔	✔	Logical	✘	Points	✘	all	Re	Iteration cannot be nested. $\vee$, $\wedge$ and hierarchies possible in principle but not available in source code.
Cayuga	✔	✔	✔	?	✘	✔	✔	No windows	?	Intervals	✘	Stam	Re	
FlinkCEP	✔	✔	✔	?	✔	✔	✔	Logical	✔	Points	✘	all	Co,Re	Patterns in Java or Scala. Extra selection and consumption policies available. Quantified iteration available.
NextCEP	✔	✘	✔	✘	✔	✔	✔	Logical	✘	Intervals	✘	Stnm variant	Re	Durative events to handle associativity of sequence operators.
DistCED	✔	✘	✔	✔	✔	✔	✔	Logical	✘	Intervals	✘	?	?	Sequence and concatenation have different semantics. Sequence does not allow sub-events to be overlapping, permissible in concatenation.
Siddhi	✔	✔	✔	✔	✔	✔	✔	Logical	✔	Points	✘	Sc,Stam	Co,Re	Quantified iteration available.
Logic														
CRS	✔	✘	✔	?	✔	✔	✔	Logical	✔	Points	✘	Stam, Stnm variant	Co,Re	Only equality predicates (unification) for σ. Iteration explicitly bounded.
RTEC	✔	✔	✔	✔	✔	✔	✘	Actual	✔	Points + Intervals	✔	Stam for instantaneous CEs, Stnm variant for durative	Re	Sequence through explicit time constraints.
PADRES	✔	?	✔	✔	?	✔	✔	?	✔	Points	✘	?	?	Based on Jess [15]. Iteration explicitly bounded.
Trees														
ZStream	✔	✔	✔	?	✔	✔	✔	Actual	✔	Intervals	✘	Stnm variant	Re	Reordered execution and lazy evaluation of patterns.
E-Cube	✔	✔	✔	✔	✔	✔	✘	Logical	✔	Intervals	✘	Stam	Re	Pattern hierarchies to share intermediate results.
Hybrid														
TESLA	✔	✔	✘	?	✔	✔	✔	Logical	✔	Points	✘	Stam, Stnm variant	Co,Re	Rules translatable to temporal logical formulas, converted to automata for evaluation.
Esper	✔	✔	✔	?	✔	✔	✔	✔	✔	Points	✘	?	?	Mixture of trees, automata and Allen's interval algebra. Windows available, but type unknown.
System	σ	π	$\vee$	$\wedge$	$\neg$	;	*	W	H	T.M.	B.K.	S.P.	C.P.	Remarks

Table 1: Expressive capabilities of CER systems.
σ: selection, π: projection, $\wedge$: conjunction, $\vee$: disjunction, $\neg$: negation, ;: sequence, *: iteration, W: windowing, H: hierarchies, T.M.: temporal model, B.K.: background knowledge, S.P.: selection policies, C.P.: consumption policies, Stam : skip-till-any-match, Stnm : skip-till-next-match, Sc : strict-contiguity, Co : consume, Re : reuse.

however, reveals that there still exists a number of pending issues.

The existence of multiple systems, each with its own language and its own variation of automaton model, might be viewed as a sign of vigor for the field. On the other hand, this heterogeneity, where the semantics of a language have to be inferred from the operational semantics of the employed automata, can be a source of confusion. A discussion about the formal semantics of automata might seem like a purely theoretical endeavor, but the lack of such semantics can have important implications. Consider the requirement for event hierarchies and compositional patterns. It is a well-known fact that classical regular expressions and automata have nice closure properties, thus allowing for compositional definitions of expressions [95]. Interestingly, this is also the case for symbolic automata, i.e., automata that have predicates on their transitions and resemble the automaton models proposed for CER [65]. However, by adding memory to such automata, so that predicates relating more than one event are possible, and the need for marking SDEs as being part of match, we essentially move to symbolic transducers with memory, in which case some of the closure properties start breaking down [64,98]. Note, for example, that classical regular expressions and automata are closed under iteration, i.e., we can take any regular expression/automaton, apply an iteration operator and the result will still be a regular expression/automaton. This is a procedure that can be repeatedly applied, thus allowing for nested iterations. On the other

hand, both SASE and Cayuga construct acyclic automata allowing only self-loops on states to handle the operator of iteration, but not loops on the whole automaton (i.e., transitions from its final to its start state). This indicates that iteration cannot be arbitrarily used, as in regular expressions. It is therefore unclear which operators for defining a CER pattern may indeed be used compositionally and whether, if a pattern does make use of nested operators, its semantics will be as expected by a user (for a more detailed discussion of this issue, see [85]).

A related issue is that of the semantics of selection and consumption policies. On the one hand, consumption policies are often ignored, where reuse is implicitly the default policy, and their definition, whenever provided, is informal [73]. On the other hand, there is no consensus regarding the semantics of selection policies. Even at the terminology level, there is substantial divergence, where certain policies, like partition-contiguity, might be subsumed under a different notion (see again Section 2.1.2). As a final note with respect to the issues of semantics, we would like to point out that the lack of well-defined semantics can be an obstacle to applying certain optimization techniques, like query rewriting, since these require a methodology for determining when two queries are in fact equivalent.

From the point of view of expressiveness, automata are, as expected, well-suited to the detection of sequential patterns. There is no CER system that is based on automata and can handle concurrency though. A conceivable solution would be to define patterns with concurrent events by using (in)equality predicates on the timestamps of SDEs. Such a solution would, however, defeat the purpose of using automata and could possibly complicate their semantics even further. Automata assume that their input symbols arrive at a certain order, which, in CER, is the temporal order of the SDEs. This simple temporal model also allows the detected CEs to be temporally ordered. Now, assume, as an example, that we need to detect two concurrent durative SDEs by using a pattern like b during a, meaning that b must happen while a is happening, i.e., $b.start > a.start$ and $b.end < a.end$ (as in Allen's interval algebra [28,29]). If there in fact exist two such SDEs in a stream, the first issue is the order in which they should be presented to an automaton. An option would be to present b first, since this is the event for which we first know all the information we need, i.e., both its start and end timepoints. Thus, upon reading b the automaton could move to a next state. Upon reading a, it could check the inequality constraint regarding the start timepoints (the constraint about the end timepoints can be skipped since we assumed SDEs are presented according to their end timepoints) and then produce a new CE. The end timepoint of this new CE though would have to be equal to $b.end$. As a result, we have produced a new event whose end timepoint is actually behind the last end timepoint of our stream, that of a. Meanwhile, any other events happening between $b.end$ and $a.end$ might have already been processed by the engine, without taking into account our new CE. A way out of this conundrum would be to send the start and end timepoints of each event separately, e.g., $a.start, b.start, b.end, a.end$. This solution could work for SDEs, but not for CEs, since, for a CE, we cannot always know when it started until we have actually seen the last event of its sequence. For example, a WITHIN constraint forces us to wait for the last event in order to make sure that the time difference between the last and the first event is less than that imposed by it. This simple example illustrates a possible difficulty in handling concurrent events using automata; other formalisms, such as Petri Nets [126] which are often used for modeling concurrent processes, could possibly prove helpful.

The lack of a mechanism for incorporating background knowledge in automata-based CER systems is another obstacle for defining yet more expressive patterns. This does not seem to be a serious limitation though; [26] is an example of a system that employs automata with predicates drawn from a knowledge base. Since these types of automata have predicates which can be Boolean formulas, it is possible to use a solver underneath, providing not only logical facts, but also further (domain-dependent) axioms, along with a full-fledged inference engine. This is indeed the approach followed in the line of work concerning symbolic automata [147]. It is also interesting to note the lack of support for conjunctive (involving $\wedge$) patterns in most automata-based methods, although conjunction can often be a useful operator. Conjunctive patterns are generally more expensive than strictly sequential ones since the ordering constraint is lifted and more matches tend to be produced. In [69], optimization techniques for handling such conjunctive patterns are presented, based on static and runtime unsatisfiability checking.

Compared to automata, logic-based systems seem to have a somewhat different focus with respect to their expressiveness. They can naturally express concurrency, as in CRS and RTEC, and to support hierarchies of events. On the other hand, iteration is either not supported (RTEC) or needs to be explicitly bounded (CRS). This also implies that they do not also support complex mathematical operators, like aggregates (e.g., averages, maximum/minimum values) that need to be applied to all the events selected by an iteration operator. In fact, CRS does not support any such operators, even without iteration. Given that engines for logic inference can easily incorporate knowledge in the form of facts, it is surprising that only RTEC can actually take advantage of any background knowledge. Except for TESLA, they also tend to ignore the existence of the various selection and consumption policies. Finally, with respect to performance, logic-based approaches have proven at least as efficient as automata and tree-based approaches.

Tree-based approaches provide multiple physical plans, either for a single pattern (ZStream) or for multiple patterns

(E-Cube), which, in turn, allows for a more efficient pattern evaluation, according to the cost of each plan. Ideas similar to those of reordered execution and lazy evaluation behind ZStream have also been applied to automata-based systems [143, 104].

We would also like to make some remarks concerning the requirements for relational events and patterns without windows (see Section 2.1.5). With the exception of Cayuga, all other systems require windows in order to function. Cayuga can handle windowless patterns by limiting its expressive power. RTEC also relaxes to an extent the requirement for windows, at least for durative CEs. Durative CEs in RTEC need not have both their initiating and terminating timepoints within the same window. This is manageable in RTEC due to the fact that durative CEs are evaluated according to a deterministic version of the skip-till-next-match policy, i.e., any new initiation timepoints after the initial initiation do not result in new CE candidates being created. It is thus an open issue how to handle windowless patterns in the presence of relaxed policies. With respect to relational events, the issue is not that of a lack of expressive power. Even automata-based systems could handle such patterns, through a constraint imposing that the object identifiers of two successive events are different; doing, in a sense, the opposite of what partition-contiguity does. Efficiency is the real issue in this case, especially when the CE has high arity (more than two objects need to be related), as such patterns would result in a significant increase in the number of created runs for automata-based systems and in a more expensive searching process for logic-based systems.

Summarizing our discussion, the most obvious research gap may be the lack of a common formal framework and of a universally agreed terminology, also indicated by the fact that there still do not exist any standard CER benchmarks. Automata-based methods seem to support most of the core CER operators, but it is still unclear how these operators may be used and what their semantics should be. It also remains an open issue how the common case of relational events could be supported. On the other hand, logic-based methods have clearer semantics, but they do not support all operators, such as iteration (or support a limited version), while they are also less flexible with respect to the allowed selection policies. The absence of support for background knowledge and the inability to handle both instantaneous and durative events are also worth noting for most methods.

An area that has not received significant attention in CER is that of automatically learning a set of patterns from the stream of input SDEs. One example of a method for the automatic extraction of CER patterns may be found in [101], where a parallel system for learning theories in the form of Event Calculus rules (as in RTEC) is presented. The issue of parallelization will be discussed in the following sections.

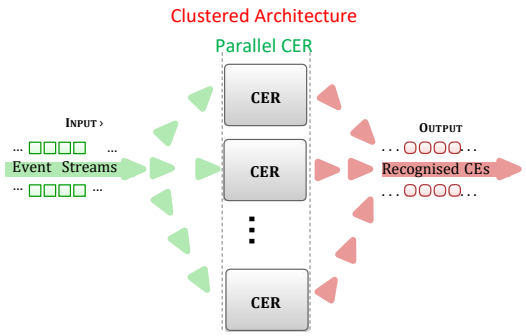


Fig. 5: Abstract clustered architecture for parallel CER.

Other examples for learning CER patterns are [118], [107], [43],[144].

3 Towards Scalable CER: CER on Big Data Platforms

CER engines [87, 67, 123, 11, 18, 35] are devoted to processing unbounded streams of event data as they arrive and provide real-time answers to continuous user queries. Most CER engines function on top of a centralized architecture and employ a serial processing model. That is, event data that arrive in the CER engine are processed serially by a single event processor [73] as shown in Figure 1. On the other hand, Big Data is characterized by sheer volume and high velocity of event arrivals in which case such a centralized, serial model causes the CER engine to become a computational bottleneck.

To enable CER in the Big Data era we need to move towards a clustered architecture composed of multiple processing units that can work in parallel. As shown in Figure 5, in such clustered architectures the computation is scaled-out to a number of machines each with multiple CPUs and cores that can work independently on parts of the given task and finally merge the partial results of each unit to derive a single outcome, i.e., the recognized CEs in the case of CER.

Parallel CER [59, 143, 94, 119, 39] usually aims at improving throughput, defined as the number of event tuples being processed per time unit. Low latency is another desirable property [143]. To continuously achieve high throughput and low latency, one may need to adjust the assignment of the processing load among the available units, on par with parallel processing. Elastic resource allocation or elasticity refers exactly to dynamically adjusting resource allocation, so that, at each point in time, the utilized resources match the current demands of running tasks as closely as possible.

Modern streaming Big Data platforms such as Apache Storm [8], Spark [7] and Flink [1] provide by design the primitives for the required scalability. They support parallel processing over clustered architectures, allow elastic resource allocation and transparently ensure various levels of resilience to failures while processing events. On the down

side, such platforms are designed as general purpose streaming engines. Therefore, the first step towards scalability is to incorporate CER functionality to general-purpose Big Data platforms. The question is how to bridge the gap between such platforms and CER engines so that one can make the most out of the two. In this section, we answer this question by outlining how CER is incorporated in modern, popular Big Data platforms. We will focus on Apache Storm, Spark and Flink. Our purpose is not to provide a full technical guide. Instead, we aim at (a) demonstrating where a CER engine of choice may be incorporated, (b) introducing the basic concepts that span these platforms and, importantly, elaborate on the notion of a physical 'task' for each. In Section 4, we review parallel and elastic CER approaches explaining that a physical task may either be a CER query, operator or operator instance [131].

3.1 CER on Spark Streaming

Conceptual View: The Spark Streaming API [5] works on one or more discretized streams, each termed a DStream. DStreams can be created either from input data streams stemming from sources such as Kafka [4] or Flume [3], or by transforming other DStreams. Data streams arrive at a Receiver process. To create a DStream, discretization takes place based on two types of time intervals. The "block interval" organizes incoming data streams into blocks of few tens of milliseconds, with each block being a data partition. Concatenating a number of blocks creates micro-batches. These are then forwarded to the core of Spark, as shown at the top part of Figure 6. There, a micro-batch is treated as an immutable collection of data tuples organized into partitions (blocks), called an RDD (Resilient Distributed Dataset).

As shown in Figure 6, as time passes new RDDs are instantiated and may undergo a number of transformations (map, reduceByKey, join, etc), window or output operations. These operations may either be 'narrow', like map, which operate on a single partition and essentially pipeline the data of that partition to a resulting single partition (right-middle part of Figure 6), or 'wide' operations like reduceByKey which require to map the data across the partitions in new RDDs (left-middle part of Figure 6). A series of such transformations or window operations form a Directed Acyclic Graph (DAG) where nodes are RDDs at various timestamps and arrows correspond to the desired operations on them. Such a DAG expresses the conceptual view (see Figure 6) describing the flow of data processing. There is also a newer streaming API available in Spark, namely Structured Streaming [6], but there is no effort on applying CER on it.

Physical & Cluster View: A Spark Cluster (see bottom of Figure 6), includes a Driver process at a Master (Driver) node and a number of Worker nodes. The Driver node is where the Spark application (i.e., the SparkContext) is created. A

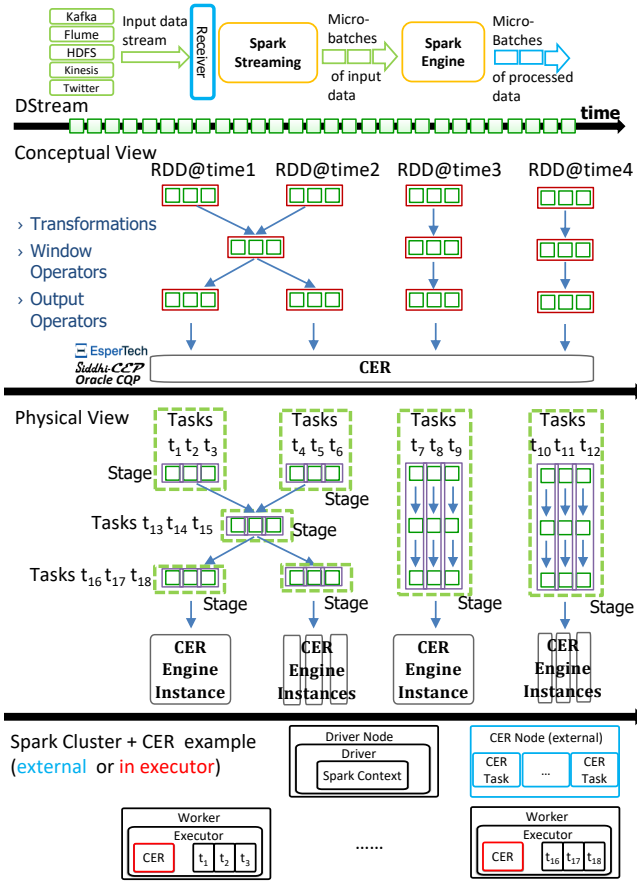


Fig. 6: CER on Spark Streaming.

Worker node includes one or more Executors (JVMs) running tasks assigned by the Driver.

In the physical view presented in Figure 6, each RDD undergoes a number of processing stages, translating the DAG of the conceptual view as shown in the figure. A stage is formed as a set of narrow transformations that can be pipelined and executed by a single Worker independently. Having divided the execution graph into stages as shown in the middle part of Figure 6, within each stage a data partition is assigned to a single task. Then tasks are assigned by the Driver to Workers. Such an exemplary assignment is shown at the bottom of Figure 6.

Parallelism can be tuned in a number of ways. For instance one can pre-partition (i.e., before reaching a Receiver processes) Kafka messages and create a DStream for each partition. Within Spark, the repartition transformation can create more or fewer partitions of a DStream. Redistributing streams using wide operations changes the partitioning of the streams as well. For example, the keyBy transformation repartitions data by hashing tuples based on key field(s).

CER Synergies with Spark Streaming: Two ways have been applied to enable synergies among CER and Spark Streaming. One way to go is to perform the required trans-

formations or window operations on Spark's RDDs (micro-batches) and then push the results to one or more, external to Spark, running instances of a CER engine. Such an approach has been described in the Stratio Decision Platform [20] where the chosen CER engine is Siddhi [18], but in principle this is also applicable on other CER engines such as Esper [11]. The bottom of Figure 6 shows how this rationale could work at the cluster level. The Worker nodes in the figure perform the required transformations, window or output operations as part of the Spark cluster. There is a CER Node marked in light blue in Figure 6. The red CER boxes - tasks are absent at the Executor level upon choosing this option. The CER Node in the cluster executes CER tasks corresponding to the exemplary CER engine instances drawn in the middle of Figure 6. Note that the actual number of such instances depends on the actual CER operation and the chosen parallelization scheme (these will be discussed in Section 4).

A more elaborate approach is to incorporate an instance of the CER engine as a CER task within each Executor, as shown in red colored CER boxes at the bottom of Figure 6. By employing this option, the CER Node in the figure is absent, the communication lags between Spark and the external CER engine instances are avoided and the local cache is shared among common Spark and CER tasks that run on the same Executor. Such an approach is adopted by OracleCQP [16, 9].

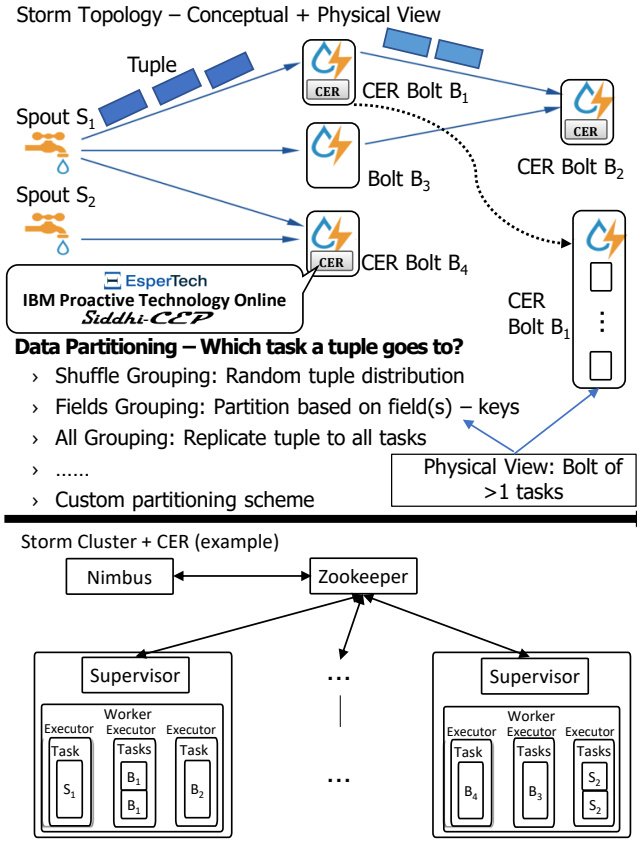


Fig. 7: CER on Apache Storm.

3.2 CER on Apache Storm

Conceptual View: The conceptual view of data processing in Apache Storm is represented by a Storm Topology, as shown in Figure 7. A Storm Topology is a DAG that includes Spouts and Bolts. Spouts are data stream sources; each Bolt, in turn, is where the actual processing takes place. Bolts can do anything from filtering, aggregations, joins, interacting with databases and more, before emitting tuples to other Bolts or applications.

Physical & Cluster View: There are three kinds of nodes on a Storm cluster as shown at the bottom of Figure 7. The Master node runs a daemon called Nimbus, responsible for assigning tasks to machines and monitoring for failures. A ZooKeeper coordinates various processes and stores all of the states associated with them. Finally, each Worker node runs a Supervisor daemon, which listens for work assigned to its machine and manages Worker processes.

Upon defining the topology, the developer has the ability to explicitly set the number of Worker processes (JVMs). A Worker process belongs to a specific topology and may include one or more Executors (see bottom of Figure 7). Each Executor is devoted to a Spout or Bolt of this topology. For each Spout or Bolt the developer can explicitly declare the number of its Executors (threads). A Spout's/Bolt's definition also allows for setting the number of tasks for the Spout/Bolt. The tasks of a particular Spout/Bolt are running instances of the exact same Spout/Bolt that produce/process different data partitions. For instance, the middle right of Figure 7 shows that Bolt B1 may be instantiated to a number of tasks. Moreover, Storm supports a number of grouping strategies that specify how data will be partitioned and exchanged among the tasks of Bolts. Custom groupings are possible, while built-in ones are also available [8], some of which are mentioned in Figure 7.

An example combining the physical view (tasks) with a possible cluster deployment is also shown in Figure 7. On the left, bottom part of Figure 7, there is a Worker process with three Executors. The leftmost Executor handles a single task for Spout S1, while the rightmost Executor runs a single task for Bolt B2. However, the middle Executor possesses two tasks for Bolt B1. Since B1 has two tasks, each of them can execute the same code on different data partitions, in parallel. Therefore, the maximum degree of parallelism in the example is 2.

CER Synergies with Apache Storm: The standard way to incorporate CER in Storm is, as shown in Figure 7, executing CER queries in CER engine instances embedded in Bolts of the Storm topology. This has been demonstrated for CER

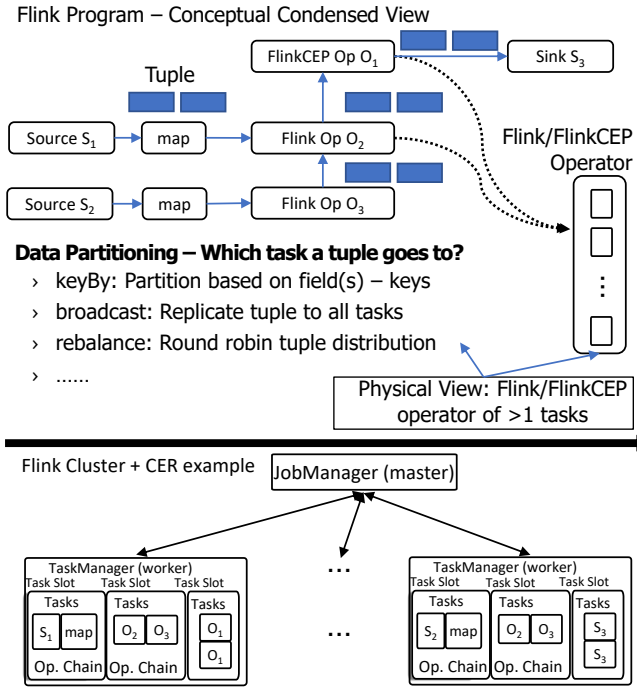


Fig. 8: CER on Apache Flink.

engines such as Esper [12] and Siddhi [156]. One step further, there exist examples of CER engines, such as IBM Proactive Technology Online [13,14], that split their functionality to a number of Bolts. A more detailed discussion on [14] follows in Section 5.3. Such an approach provides greater flexibility since different degrees of parallelism can be applied at various stages of CER within the CER engine.

3.3 CER on Apache Flink

Conceptual View: The basic building blocks of a Flink program are Data Sources (streams), Operators and Data Sinks bound together in a directed graph (see Figure 8) which is not necessarily acyclic. From a developer's viewpoint, Flink operators or data sinks may resemble Spark transformations, window operations or output operations, respectively. However, the implementation of these operators is much different. Flink is a true streaming engine treating batch processing as special case of streaming with bounded data and not vice versa as it is the case with Spark. In Flink, each data source or operator (map, keyBy, filter etc) is implemented as a long running operator similar to Spouts and Bolts in Storm. Flink also gives low-level control on the exact stream partitioning after a transformation, similar to Storm groupings.

Physical & Cluster View: A Flink cluster is composed of (at least one) Master and a number of Worker nodes. The Master node runs a JobManager for distributed execution and coordination purposes, while each Worker node incorporates a TaskManager which undertakes the physical execution of tasks. Each Worker (JVM process) has a number of task slots (at least one). Each operator or instance of an operator of the Flink program is assigned to a slot as shown in Figure 8 and tasks of the same slot have access to isolated memory shared only among tasks of the same slot. The new concept here involves task chaining. That is, Flink allows to place two operators (or instances of operators) together into one task i.e., thread (for instance, <source, map> in the figure) for performance reasons.

At the level of an operator/data source/data sink, parallelism is configured by calling a setParallelism method. Redistributing streams using wide operations (as in Spark) changes the partitioning of the streams as well. For instance, as shown in Figure 8, keyBy repartitions by hashing key field(s), broadcast replicates the operator outcome and rebalance performs round robin repartition.

FlinkCEP: Flink provides built-in support for CER via FlinkCEP [2], which was discussed in Section 2.2 (see also Table 1). Because of FlinkCEP, CER operators are incorporated in the Flink Program and translated in physical tasks as any other Flink operator. An example of a mixture of non-CER and CER operators in Flink, at the conceptual and physical view, is presented in Figure 8. Besides using FlinkCEP, Flink provides Storm compatibility [19] making it possible to migrate a Storm Topology with CER Bolts to a Flink Program.

3.4 Comparison of CER Implementations on Big Data Platforms

Table 2 presents the default features of CER implementations on Big Data platforms. More advanced features require custom code as we explain shortly. The first two rows of the table are derived from our previous discussion. Event delivery guarantees refer to how many times an event tuple will be processed in case of system failure and, as it will be discussed shortly, they directly affect the reliability of CER. exactly-once is the strongest guarantee where each event tuple will be processed exactly once, as it would when no failures occur. Flink provides exactly-once guarantees for FlinkCEP. at-least-once is a weaker guarantee where no event tuple will be lost but, it may be processed multiple times. Storm provides at-least-once guarantees. at-most-once means that after a system failure a tuple may be totally lost.

Note that Spark Streaming can provide exactly-once per se, but its combination with a CER engine as described in Section 3.1 affects this guarantee. An external CER node, as in the example highlighted in light blue color in Figure 6, can by default provide only the weaker at-most-once guarantee. This is the result of combining exactly-once of Spark Streaming and the essentially no delivery guarantee of a primitive external CER node setup. On the other hand, if one

CER over Big Data Platforms				
Feature	**Apache Spark Streaming**		**Apache Storm**	**Apache Flink**
CER @	External Node(s)	Executor	Storm Bolt	Native CER Operator
Event Processing Unit Size	Micro-batch of Events	Micro-batch of Events	Event Tuple	Event Tuple
Event Delivery Guarantees	at-most-once	at-least-once	at-least-once	exactly-once
Windowing	Time-based	Time-based	Time, Tuple-based	Time, Tuple-based
Event Temporal Processing	processing-time	processing-time	event-time, processing-time	event-time, processing-time, ingestion-time
Out of Order Processing	No	No	Yes, using event-time	Yes, using event-time
Ease of Use	✘ Separate node(s) for CER	✘ Manual launch of CER engine at each Executor or via custom cluster manager	✔ Import CER engine's functionality in Bolt's definition	✔ Native CER API
Flexibility	✘✘✘ RDD to event tuples conversion from/to the CER engine. Requires custom handling of tuple-based windows. Only processing time-based windows	✘✘✘	✔✔✔ Flexible, if CER engine can be incorporated as a library in Bolt. Support for time- and tuple-based windows. Support for processing and event time-based processing	✔✔✔ Most flexible. Support for time- and tuple-based windows. Supports all event temporal processing models
Reliability	✘✘ Due to at-most/least-once guarantees custom checks for selection/ consumption policies are needed. Unless custom cluster manager is developed. Out of order processing also requires custom code	✘✘	✘✔ at-least-once delivery requires custom checks for selection/ consumption policies. Out of order processing supported	✔✔ Most reliable, with FlinkCEP. exactly-once guarantees. Out of order processing supported

Table 2: Features that popular Big Data platforms attribute to CER. The ✔ and ✘ marks denote advantages and drawbacks, respectively, explained in the accompanying text.

incorporates the CER engine at each Executor, highlighted in red color in Figure 6, then Spark Streaming keeps providing exactly-once for the processed micro-batches treated as RDDs. However, each RDD will be pushed out to the CER task of the same Executor and will be converted to single event tuples. Pushing data out of an RDD format provides a default behavior of at-least-once [5].

Regarding windowing (see Section 2.1.4), contrary to Storm and Flink, Spark Streaming provides native support only for time-based windows. Despite the fact that a CER engine may natively support both tuple-(equivalently count-) and time-based windows, ensuring the overall functionality of tuple-based window operations as event tuples move from and to Spark Streaming requires custom code.

The sixth row of Table 2 refers to the temporal processing model. The processing-time option says that all time-based operations (like time-windows) will use the system clock of the machines that run the respective operations. The event-time option specifies that all time-based operations will use a timestamp attribute tagged on the event tuple by the event producer device, typically before the event tuple enters the corresponding Big Data platform. Finally, ingestion-time is the time that an event enters the corresponding Big Data platform (i.e., at a source operator in Flink). As Table 2 shows, Flink supports all these options, Storm omits ingestion-time, while Spark Streaming only supports processing-time.

Out of order event arrivals (see the seventh row of Table 2) may occur due to network congestion and latency on communication links delivering event tuples to the Big Data platform. The support of event-time has a direct effect on the ability of the platform to provide built-in functions for solving out of order event arrivals, as shown in the sixth row of Table 2. All three platforms define out of order arrivals based on event-time [6, 8, 1]. Because Spark Streaming does not support event-time, it only allows the developer to customly resolve such issues. Storm and Flink allow the specification of a slack interval parameter in built-in functions tailored to out of order handling. The platforms defer the computation of order-critical operators (such as windows) waiting for delayed events an amount of time equal to the slack interval. Events are buffered and re-ordered after delayed events arrive, but before the operator evaluation begins. Techniques for resolving such issues are also discussed in Section 6.

The second part of Table 2, i.e., the last three rows, summarizes the effect of the above features from an ease of use, flexibility and reliability viewpoint, respectively. For instance, Spark Streaming is marked as a not easy-to-use choice in the context of CER as it requires launching external CER nodes or CER tasks in Executors of the cluster. Moreover, it

is judged as inflexible as it requires micro-batches to be converted to individual tuples before being fed into a CER task (external or in an Executor) and vice versa. It further requires custom handling of tuple-based windows and supports only the processing-time temporal processing option. Therefore, it receives three ✘ marks regarding flexibility, while Storm and Flink which (i) process each tuple individually, (ii) provide native support for tuple- and time-based windows, (iii) support event-time and processing-time temporal processing models, receive three ✔ in the corresponding fields.

The last row of Table 2 concerns reliability features for each platform based on its ability to allow exactly-once event delivery guarantees and to provide built-in support for handling out of order events. at-most-once and at-least-once may affect the accuracy (and, thus, the reliability) of CER with respect to the desired event selection or consumption policy (Sections 2.1.2 - 2.1.3). In platforms providing such guarantees one should include custom code provisions for addressing the requirements of the application under consideration. For instance, assume that a consume consumption policy is specified in the CER query and an event tuple has already been consumed in a pattern match before a system failure. at-least-once may allow the same event tuple to be consumed again upon the failure recovery. In contrast, at-most-once may result in a tuple being totally skipped while it should not. Handling out of order event arrivals also affects the reliability of CER on Big Data platform synergies. For instance, a strict-contiguity policy will produce different CEs using the event-time option compared to using an ingestion-time option, unless event tuples arrive in order.

3.5 Open Issues in CER on Big Data Platforms

Benchmarking CER on Big Data platform synergies is the major open issue here and perhaps one of the most difficult to address among the open issues outlined in this survey. This is because it lies at the intersection of (and thus, requires):

- Benchmarks of Big Data platforms, such as the Yahoo! Streaming Benchmark [54] that puts Spark Streaming, Storm and Flink to the test.
- Benchmarks of CER engines, where there is a lack of industry-wide and well-accepted benchmarks[1]. For instance, efforts like BiCEP [124] were not completed.
- Benchmarking combinations of the above based on criteria such as those presented in Table 2.
- Benchmarking parallel CER approaches, as discussed in Section 4.4.

One of the earliest and established benchmarks both in general purpose stream processing systems and CER engines, covering the first two of the above bullet points, is the Linear Road Benchmark (LRB) [33,160]. However, recent results in benchmarking [99] show that LRB and other benchmarks such as StreamBench [116] introduce inaccuracies in the derived performance (throughput, latency or other) measurements upon applied to the context of Big Data platforms, or even include bottlenecks in their implementations.

These recent advancements should be accounted for in future efforts on introducing fair benchmarks for CER implementations on Big Data platforms. Since, to our knowledge, no prior work surveys synergies among CER and Big Data platforms the way we do in the current section and summarize in Table 2, this area currently remains unexplored. In addition, to our knowledge, no prior work surveys the properties of parallelization schemes as those discussed in Section 4.4. The exploration of all possible combinations of Big Data platforms (such as Spark, Storm, Flink) that can admit CER synergies, combined with the number of (potentially hybrid) parallel CER implementations discussed in the upcoming section, introduce a wide space of alternatives that a series of future works could cover.

4 Scalable (Parallel and Elastic) CER

Having achieved CER engine to Big Data platform synergies in one of the ways described in the previous section, it is the developer's responsibility to prescribe the desired parallelization strategy in her code before submitting a query, or even engage various parallelization schemes throughout the CER process. There is a number of parallelization strategies tailored to CER that have been proposed in the literature, summarized in Figure 9. The suitability of a certain scheme depends on the specifications, and therefore the needs, of a given CER query. In what follows, we describe the rationale and judge the suitability of the reviewed strategies based on the following criteria [81,39] (see Figure 10), which often constitute common pitfalls in delivering parallel CER in practice:

- Parallelization Granularity - Agility: This criterion refers to the ability to fine tune the distribution of workload so that parallel processing achieves (a) Load Balance among the processing units that participate in CER query processing and (b) limits Data Replication and Communication.
- Support for Event Selection Policies: Few parallelization schemes are capable of supporting all event selection policies, i.e., strict-contiguity, partition-contiguity, skip-till-next-match and skip-till-any-match.
- Support for Event Consumption Policies: Similarly to selection policies, not all parallel schemes are suitable for all event consumption policies, i.e., consume, reuse and bounded-reuse.

[1] http://www.espertech.com/esper/esper-faq/#benchmarks

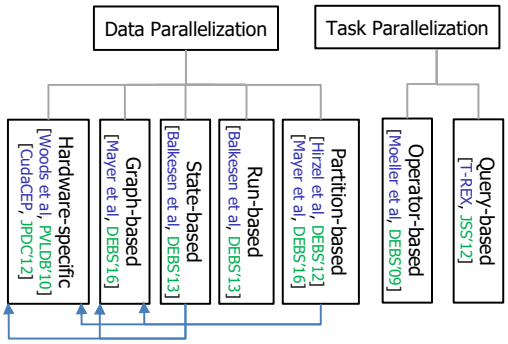


Fig. 9: Parallel CER Approaches. An outgoing/incoming arrow indicates that a data parallel scheme lends/borrows design concepts to/from another.

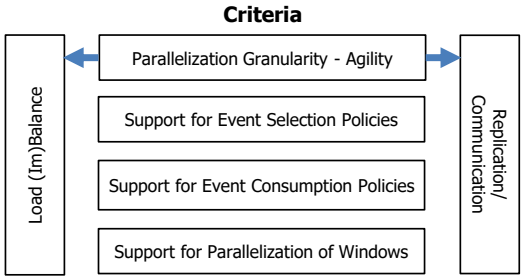


Fig. 10: Suitability Criteria for Parallel CER.

– Support for Parallelizing Window Operations: Sliding windows may be viewed as overlapping partitions of the input data that can be processed in parallel. Window parallelization is not trivial since computations may require continuous communication of state among parallel tasks. We will examine the cases of tuple-based windows and time-based windows.

Parallelization can be broadly categorized to Task and Data parallelization [131, 81]. In task parallelization, multiple CER queries are processed in parallel. In this case, the CER query is interpreted as a physical task (see Section 3) placed at the Executor or task slot of a Worker and all relevant data are pipelined and serially processed. To further avoid data and processing redundancy, queries may be decomposed to their operators. In this case, the physical task involves the execution of a certain operator and one can take advantage of operators shared among multiple queries. Following [131], data parallelization, on the other hand, focuses on a single operator of a query, with each operator running in multiple instances and each instance handling a different partition of the data. A physical task is assigned to an operator instance working on a certain data partition. Parallel CER categories are summarized in Figure 9.

Recall from Section 3 that parallel CER aims at improving throughput and/or keeping latency low. To continuously achieve these goals, elastic resource allocation takes place

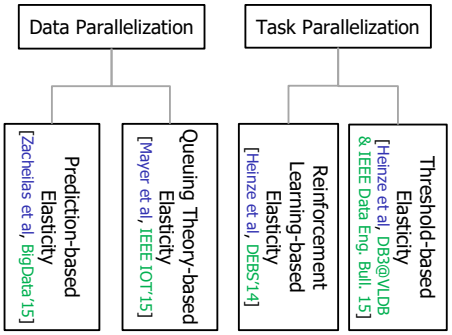


Fig. 11: Elastic CER approaches and parallelization category they have been applied to.

after applying a parallelization scheme and performing an initial assignment of computing resources (CPU, memory, bandwidth) to tasks. Then, statistics regarding resource utilization and physical task state (partial pattern matches, active windows, etc) are measured and analyzed. In case the analysis shows that assigned resources are over- or under-utilized, resource re-allocation is performed so that the allocated resources match the current needs of physical tasks. Elastic resource allocation can end-up in one of the following decisions: (a) scaling-out (i.e., increasing the amount of resources devoted to a physical task) due to over-utilization, (b) scaling-in (i.e., devoting less computing resources) due to under-utilization, (c) performing load balancing where running tasks are re-assigned to currently occupied computing resources in order to avoid over- and under-utilization and (d) taking no action.

In order to decide which decision is the best, elastic CER approaches take into consideration the collected statistics and predict the expected benefit from each of the adaptation options. The expected benefit quantifies better (balanced) resource utilization, increase of throughput or reduced computational latency and the cost of migrating currently running tasks to other virtual or physical machines, splitting/merging their execution to more/less instances and thus processing units. The following categories of elastic CER schemes have been introduced [93]: (i) threshold-based, (ii) reinforcement learning-based, (iii) queuing theory-based, (iv) prediction-based. Figure 11 summarizes the parallelization category to which these elastic schemes have been applied.

For each parallelization category (Figure 9), we accompany the description of the parallel schemes with the respective elastic resource allocation techniques that have been applied to them (Figure 11). Most of the presented schemes have been developed on top of automata-based systems. However, in principle, they are equally applicable to the rest of the system categories discussed in Section 2. Henceforth, we use the term 'processing unit' to refer to either a single-core CPU or a core of a multi-core CPU. For ease of presentation we make the following convention: upon describing a par-

allelization scheme we assume that each CER query, CER operator (for task parallelization) or CER operator instance (for data parallelization) is assigned a dedicated processing unit. We then show how elastic resource allocation may place multiple queries, operators or instances to a single processing unit, where they operate in a pseudo-parallel fashion also taking advantage of hyper-threading, and then make scale-in, scale-out, load balancing or no action decisions.

4.1 Task-based Parallelization and Elasticity

4.1.1 Query-based Parallelization

For query-based parallelization we base our description on T-Rex [59] which fosters an elaborate approach to implement this type of CER parallelism. However, we keep our discussion generic. The structure of query-based parallelism is shown in Figure 12. Each query, composed of multiple CER operators, is assigned to a processing unit and multiple CER queries are executed in parallel. In Figure 12 each column shows the stages of CER query processing which are outlined below.

As soon as an event tuple streams into the cluster, a splitter consults a static index of Automaton Models, each representing a query, to verify which query the tuple is input to. For each of these static automata, there is a number of active instances (called Sequences in Figure 12) along with their currently active states. These sequences essentially constitute the current set of partial pattern matches. To mark the active state of each automaton instance, a State Index is used as shown in Figure 12. Once all the instances to which the new tuple is input have been identified, further checks related to window operations or other filtering criteria are performed. Consequently, a new automaton instance may be created, or an existing automaton instance may proceed to a new state.

To limit memory usage throughout parallel query processing, an explicit copy of an event is only stored in a shared memory component (see the left of Figure 12), while automaton instances only keep pointers to these events. Going back to our discussion in Section 3, we would like to emphasize that shared memory can only be assumed within a JVM. A heap is created at JVM start-up and shared by all JVM threads. Queries running on separate JVMs require duplicating and communicating input event tuples. Upon a full pattern match, i.e., when an automaton instance reaches its final state, the pattern is forwarded to the Generator component (see Figure 12), which retrieves the set of events that compose the pattern from the shared memory. The recognized CEs are then forwarded to the relevant subscribed applications. A similar rationale has been adopted by Cayuga [44].

With respect to our criteria (Figure 10), since parallelization is applied at the query level, query-based parallelization is capable of supporting all event selection and consumption

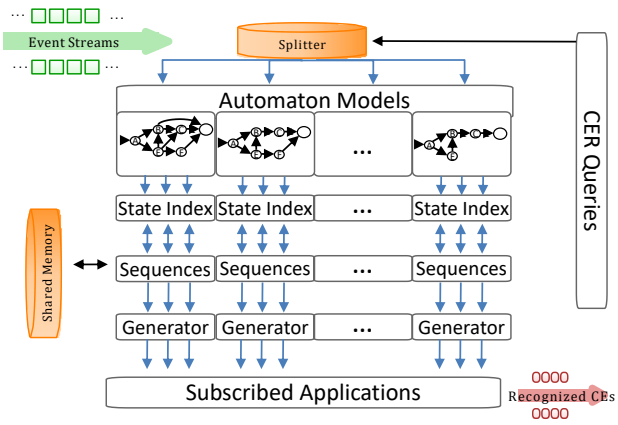


Fig. 12: Query-based Parallelization Scheme [59].

policies. On the other hand, it does not allow for parallel processing of windows and poses limitations on balancing the load among the processing units since the load is solely dependent on the complexity of the query assigned to a particular processing unit. Due to the fact that shared memory usage is not always an option, tuples may need to be replicated and communicated multiple times. Although for JVMs running on the same machine communication does not necessarily mean that data hit the network card, further tuning is required to group queries that share input on the same machine. But even then, since the query is treated as an atomic unit, operators shared among multiple queries need to be evaluated multiple times, i.e. redundancy is introduced in processing the queries.

4.1.2 Operator-based Parallelization

The basic concept in operator-based parallelism, as implemented in NextCEP [143], is to perform intra-query and multi-query optimizations while processing each individual operator in parallel, i.e., each operator is assigned to a processing unit. Multi-query optimization is achieved by leveraging operator sharing. That is, a single operator included in multiple queries needs to be executed only once, by a specific processing unit. Furthermore, the incoming data of this operator need not be replicated for multiple queries as in query-based parallelism. Intra-query optimization is achieved by query rewriting, also taking into account operator statistics. The key idea here is that each given query is analyzed to its operators and each operator is rewritten in multiple equivalent forms, leveraging operator properties including (i) commutativity i.e., $Op(Op_i, Op_j) \equiv Op(Op_j, Op_i)$ with $Op \in \{\wedge, \vee\}$ and Op_i, Op_j any of $\{\wedge, \vee, ;\}$ and (ii) associativity i.e., $(Op_i, Op_j), Op_k \equiv Op_i, (Op_j, Op_k)$ for $\{\wedge, \vee, ;\}$.

One query rewriting is then chosen based on simple cost estimation formulas incorporating incoming event rates and the principle that operators with the rarest input events should

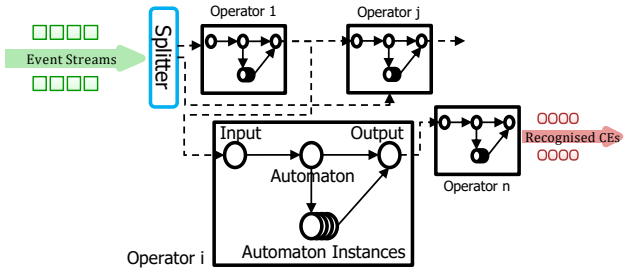


Fig. 13: Operator-based Parallelization Scheme [143].

be evaluated first [72]. This principle is applied since an operator with infrequent input events will rarely produce a full pattern match and therefore only a limited number of CEs will be fed to the next operator that receives input from the former. The chosen query form is transformed into a query graph as shown in Figure 13. The Splitter directs relevant input to each operator. Operators that receive only SDEs are evaluated first and then communicate full pattern matches (CEs) to connected operators. Operators without outgoing edges produce the CER query output. The execution of each operator within a processing unit can be materialized by structures similar to the ones presented in Section 4.1.1 (input event handler, automaton, automaton instances, operator-specific output construction and evaluation), only this time applied per operator. As already noted, multi-query optimization is allowed, because by decomposing a query to its operators, operators that are shared among queries need not be evaluated more than once. Nevertheless, in principle, an operator can be considered as shared if its specifications, such as input event types and predicates, match exactly among multiple queries.

This restriction is overcome in the work of [137] which proposes the SPASS framework, aiming at optimizing the parallel execution of multiple CER operators even if their specifications do not precisely match. The basic idea is that an optimizer module identifies a set of common sub-patterns among the CER operators being executed. It decomposes these CER operators using associativity, commutativity and other Boolean algebra properties to new, equivalent expressions which isolate the shared sub-pattern. For instance, a CER operator written in SASE form as $\mathsf{SEQ}(A, B, C, D)$ can be rewritten to $\mathsf{SEQ}(\mathsf{SEQ}(A, B), \mathsf{SEQ}(C, D))$ and if $\mathsf{SEQ}(A, B)$ is shared among many queries it is isolated and executed independently. During the evaluation of the CER operators, the intermediate results (partial pattern matches) produced by shared sub-patterns are incorporated in shared memory views so that they are accessible by all CER operators incorporating these sub-patterns. Therefore, the sub-patterns do not need to be constructed more than once. Such an idea is useful in maximizing the potential of operator-based parallelization via exploiting operator sharing, however, it does not incorporate event selection and consumption policies that may differ among operators. Additionally, as was the case with shared memory in query-based parallelization, it requires further fine-tuning at the cluster level to make sure that all CER operators sharing a sub-pattern can access shared memory views avoiding partial matches' replication.

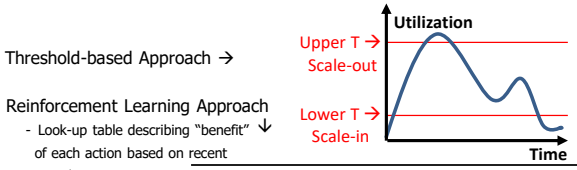
Operator-based parallelization, as discussed in [143], provides support for the whole set of event consumption policies, while event selection policies are also supported excluding skip-till-any-match. That is because skip-till-any-match violates the "unique immediate successor" property. This property implies that in order to apply the associativity property of the sequence operator during query rewriting, there should be no ambiguity which event is the immediate successor of another, so that any possible rewriting produces equivalent output. Finally, operator-based parallelization does not examine the parallel processing of windows.

4.1.3 Threshold-based Elasticity

Elasticity on par with task (query- or operator-) parallelization in CER has been considered in the FUGU resource allocator [91,93,90,92]. FUGU initially assigns the set of physical tasks, each corresponding to a query or operator, to processing units using a First Fit Bin Packing heuristic discussed shortly. Then, one of the elasticity schemes that FUGU incorporates is the threshold-based approach discussed in [91, 93,90,92]. The basic principle behind this approach is that a low and a high threshold are used in order to judge under- and over-utilization of processing units (computing resources in general), respectively, as shown in Figure 14. The elastic allocator periodically checks whether either of these thresholds is crossed so that it schedules a scale-in (due to under-utilization) or scale-out (due to over-utilization) procedure.

There are two approaches to impose such thresholds. They may either be local i.e., per processing unit, or global i.e., a quantity measuring whether the global recourse utilization exceeds or drops below a threshold. In case local thresholds are used, a subset of the tasks (CER queries or operators) assigned to an overloaded processing unit is selected (in [93] using a subset sum algorithm) so that the load taken away from the processing unit reduces its utilization below the high threshold. The moved CER queries or operators are re-assigned to non-overloaded processing units or new ones are occupied, i.e., scaling-out. An under-loaded processing unit is released (scale-in decision) by moving all CER queries or operators assigned to it to other non-overloaded peers (if possible).

Global thresholds may be defined using a function synthesizing the utilization of different processing units such as the average utilization adopted in [93]. If this average exceeds the high threshold, an amount of load is moved from each processing unit. The amount of load that will be moved



Reinforcement Learning Approach
- Look-up table describing "benefit" ↓ of each action based on recent experience

Utilization	Scale-in	No Action	Scale-out
80%	0.28	0.7	0.88
90%	0.28	0.5	0.9
100%	0.1	0.4	1.0

Fig. 14: Threshold-based and Reinforcement Learning-based Approaches [91,93,90,92].

can be determined, for instance, by measuring how much the high threshold is exceeded and almost equally distributing this quantity among the processing units. Another approach could well be that only the most overloaded processing units have their tasks being re-allocated. If the low threshold is exceeded, the processing units with the minimal utilization, so that the new average surpasses the low threshold again, are released and tasks are re-assigned (if possible).

Regarding CER query/operator placement to processing units and load balance, in every scale-in or scale-out decision as discussed above, a First Fit Bin Packing heuristic is used for assigning tasks to processing units [93]. In First Fit Bin Packing, tasks are ordered based on their expected workload. The available processing units are the bins along with their remaining utilization capacity. A CER query/operator is placed in the first processing unit that has enough remaining capacity to host it or in a new processing unit.

Various versions of the above technique can be constructed so that resource re-allocation is only permitted when thresholds are crossed in multiple consecutive checks. Moreover, one can schedule the elastic allocator to perform periodic checks so that frequent re-allocations are avoided. Migration costs including the introduced computational latency upon moving tasks can be taken into account for judging whether task re-allocation is beneficial [108,90,92]. Similar threshold-based approaches have also been discussed in the broader stream processing literature [86,74].

4.1.4 Reinforcement Learning-based Elasticity

Reinforcement learning-based elastic resource allocation, in FUGU [93], attempts to exploit past experience in order to drive future scale-in or scale-out decisions. The elastic CER resource allocator constructs a look-up table as shown in Figure 14 and learns the values of its cells. The rows of the table are current utilization values, while columns correspond to scale-in, scale-out or no action decisions. The value of each cell quantifies the benefit of the corresponding decision given the current resource utilization. Initially, the look-up table may point to scale-in decisions for all entries below a configured lower threshold and to scale-out above an upper threshold. Afterwards, given the current resource utilization value, a corresponding decision is taken and the benefit learned in practice updates accordingly the corresponding table cell. For instance, according to the table in Figure 14, for any value of the current utilization reaching or exceeding 80%, the higher benefit comes from a scaling-out decision. Being part of the FUGU approach, reinforcement learning-based elastic CER relies on First Fit Bin Packing heuristic for CER query/operator placement and load balance. A reinforcement learning approach is also described in [115], but using Markov Decision Processes instead.

In principle, both threshold- and reinforcement learning-based elastic schemes can function on par with data parallelization (besides task parallelization) schemes. Nonetheless, in the literature these approaches have not been implemented in a way that involves increasing or decreasing the running instances on a per CER operator fashion.

4.2 Data Parallelization and Elasticity

4.2.1 Partition-based Parallelization

We continue our discussion with data parallelization and partition-based parallelism in particular. Recall that in data parallelization relevant techniques focus on a single operator which is instantiated a number of times, with each instance receiving a different partition of the input event data. Two types of partition-based parallelism have been proposed, namely partition-key based parallelism [94] and pattern-sensitive parallelism [119]. As shown in the middle part of Figure 15, the structure is similar in both schemes. A splitter partitions the input, a number of operator instances undertake the processing of data partitions and a merger synthesizes (based on the operator's specifications) partial results from operator instances so as to extract the final CEs. The difference between partition-key and pattern-sensitive parallelization concerns the way the data is partitioned and whether an event tuple may belong to one or multiple partitions.

In partition-key based parallelization [94] the input event stream is assumed to be keyed or it can be pre-processed so that tuples possess key fields. Hence the splitter partitions the data based on the key of the incoming event data tuples. Such a scheme is very often useful in practice where the CER query groups incoming tuples based on key field(s). Such examples are shown at the top of Figure 15 involving telecommunication companies where CER queries may group tuples by caller or callee, social media analysis where queries often group incoming event tuples by user id, or location-based services provided by applications, where event tuples are grouped by the area in which events occur.

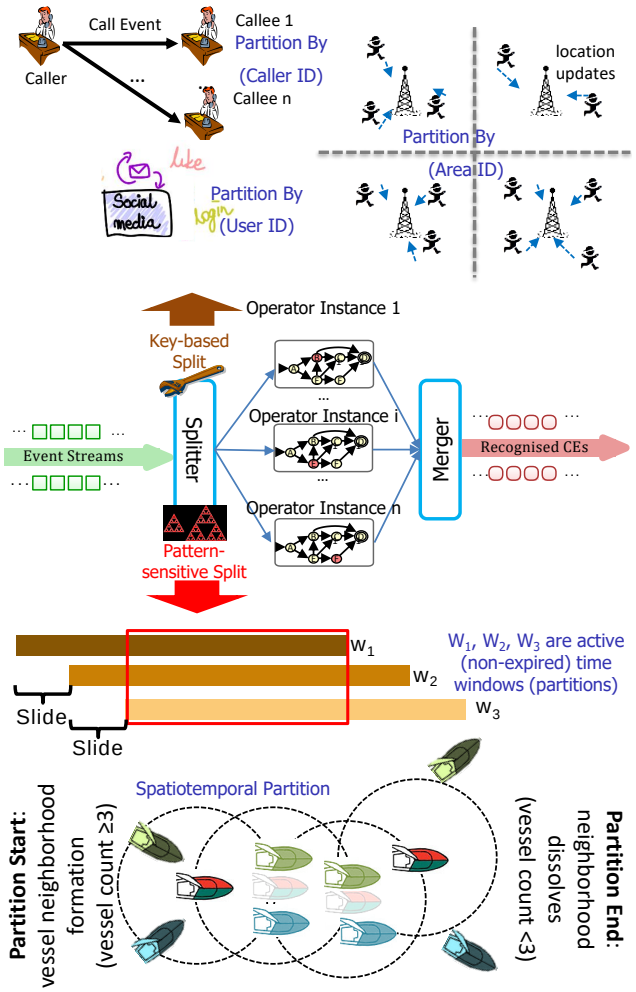


Fig. 15: Partition-based (Partition-key [94] and Pattern-sensitive [119]) Parallelization Schemes.

Due to data partitioning, only the partition-contiguity event selection policy can be supported by this scheme. All event tuples with the same key are included in one partition, which imposes partition isolation. Hence, the data concerning a certain key need not be replicated or communicated among the processing units. Due to partition isolation all event consumption policies are applicable. Load distribution strongly depends on the frequency of the keys of the incoming event tuples.

In pattern-sensitive parallelization [119], the occurrence of an event may act as a Boolean trigger for creating a new partition. An event may, depending on the specifications of the CER operator, (i) open a new partition, (ii) be included in an existing partition, (iii) terminate a partition. The bottom part of Figure 15 depicts examples of pattern-sensitive parallelism. An example involves sliding windows. In time-based sliding windows, there is a window slide and a window size parameter which are time intervals, with the size of the slide being smaller than the window size. A new partition - window is created when a time interval equal to a window slide passes. A partition is no longer active, i.e., is terminated, when a time interval equal to the window size has passed since its creation. Each partition W_1, W_2, W_3 in Figure 15 is a window that extends within a constant sized time interval which is the window size. All events with timestamps between the start and end of each partition (active window) are part of it and some of them expire after a window slide. The red rectangle in the figure marks the overlap among all partitions, including events that need to be replicated/communicated. In tuple-based sliding windows (not depicted in Figure 15), instead of time intervals, the window slide and the window size are measured in terms of the number of event tuples that arrive in the system.

The second example illustrated at the bottom of Figure 15 involves spatio-temporal partitions taking for example the maritime surveillance domain. In this example, a dynamic partition may be created when the number (count) of vessels within a certain spatial radius exceeds a given threshold. As vessels move together, new vessels enter or existing vessels exit the spatial neighborhood. The events of interest that these vessels cause are assigned to the created partition. A dynamic partition may be terminated when the count of vessels in the neighborhood drops below a threshold. Since such spatio-temporal neighborhoods may overlap, each vessel along with its events may belong to more than one partition.

The pattern-sensitive parallelization scheme is a suitable technique for parallelizing the processing of time- and tuple-based windows. Due to data partitioning, only the partition-contiguity event selection policy can be supported by this scheme as well. An incoming tuple may belong to more than one partitions as in our above examples. Therefore, data replication, in the worst case to a degree equal to the number of active partitions, cannot be avoided. Because, partition isolation does not hold in pattern-sensitive parallelization, the only consumption policy that can be supported is the reuse policy. All the remaining consumption policies require continuous communication among the processing units in order to determine which partition will consume an event first (in consume) or decide on how many times an event has been consumed (in bounded-reuse). Since partitions are dynamic and streams are volatile this scheme is prone to load imbalance.

There exist efforts that attempt to handle data replication and communication overhead [122] or examine support for consumption policies over sliding windows [121]. However, their success is highly dependent on constantly accurate predictions [122], or approximation with loose quality guarantees [121].

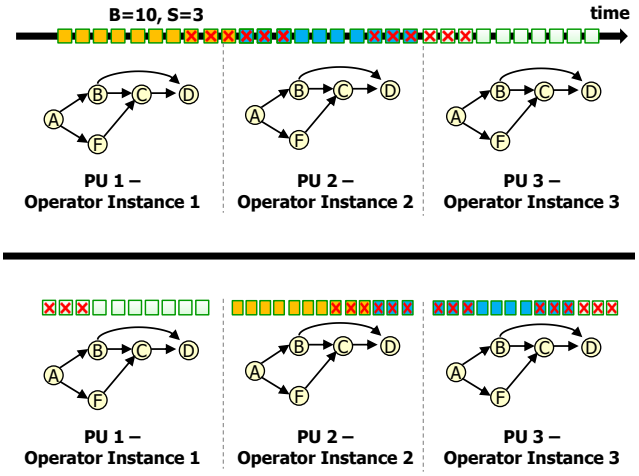


Fig. 16: Run-based [94] Parallelization. Colored boxes are event tuples. Batches of tuples are marked with different colors. PU stands for Processing Unit. Tuples marked with ✘ are replicated in a pair of PUs. CER operator instance evaluation is based on the depicted NFA.

4.2.2 Run-based Parallelization

Run-based parallelism [39] employs the neat observation that if at most N events can participate in a full pattern match, according to the CER operator specifications, then data can be partitioned to batches of $B \geq N$ size. Each such batch is a data partition that can be processed by different instances of an operator assigned to different processing units. Some full pattern matches may extend to consecutive batches by including input events from both. Therefore, there is the additional restriction that $S = N - 1 \leq \frac{B}{2}$ and S tuples at the start of a batch need to be replicated to the processing unit that handles the chronically preceding batch. Each processing unit detects all matches that start in the first $B - S$ events in a batch. The rationale is illustrated in Figure 16.

An example of a CER operation supported by this scheme involves tuple-based windows. This is because the size of the batch B can be set to a factor of the tuple-based window size (W most recent tuples arriving in the system), this way creating corresponding partitions. On the contrary, in time-based windows, the batch size cannot be set a priori. As another example of a supported operation consider a sequence operator SEQ$(e_1, \ldots, e_N)$ along with a strict-contiguity selection policy.

Run-based parallelism has the potential of balancing the load among the processing units, since each operator instance processes an equivalent amount of tuples. Furthermore, despite the fact that it requires data replication and communication based on the S parameter, this is restricted to an a priori known number of processing units and is query dependent instead of being linked to volatile input data distributions. Regarding support for event selection policies, due to the fact that each batch is processed isolated, the restriction that arises relates to partition-contiguity. In case a data partition, i.e., the batch, internally contains logical CER partitions (e.g. based on a key), the size of these partitions needs to also be a priori known or at least upper bounded in order to allow for implementing the partition-contiguity selection policy. This is often not the case in practice; for instance an upper bound does not exist in key-based partitions. Finally, with respect to event consumption policies, the reuse policy is supported. Nonetheless, consume and bounded-reuse require additional communication among processing units that handle consecutive batches. This is necessary so that the pair of the processing units can determine where an event tuple is consumed first (in consume) or to decide on how many times an event has been consumed (in bounded-reuse).

4.2.3 State-based & Hardware-centric Parallelization

In partition-based and run-based data parallelization, each operator instance holds a full version of the corresponding CER operator, assigned to a certain processing unit, working on different data partitions. In state-based parallelization [39, 60] each processing unit is not assigned a full-fledged version (instance) of the CER operator, but handles certain states of the NFA that corresponds to the operator. This in turn means that each processing unit initially handles events of certain types that are input to the operator.

The splitter distributes incoming event tuples based on their type, directing them to the responsible processing unit as shown at the top of Figure 17. Each processing unit independently performs checks on filtering conditions or predicate evaluation per state. In order to evaluate predicates that engage more than one event type, the corresponding event types may need to be replicated to the relevant processing units. For instance, consider a query in SASE-like format:

PATTERN SEQ (A,OR(B,F),C,D)
WHERE $A.attribute_1 > 30$
AND $B.attribute_1 = C.attribute_2$
AND $A.attribute_1 < 60$
WITHIN 60 SECONDS

Each of A, B, C, D, F is assigned to a different processing unit. The condition $A.attribute\_1 > 30$ as well as the condition $A.attribute_1 < 60$ are evaluated by the processing unit handling tuples of event type A. For evaluating $B.attribute_1 = C.attribute_2$ it suffices for the processing unit that handles event type B to receive additional information about $C.attribute_2$.

Besides having each processing unit working independently on filters and other predicates, as new events arrive, the evaluation of the CER operator takes place in an incremental fashion. This is achieved by pipelining event tuples that

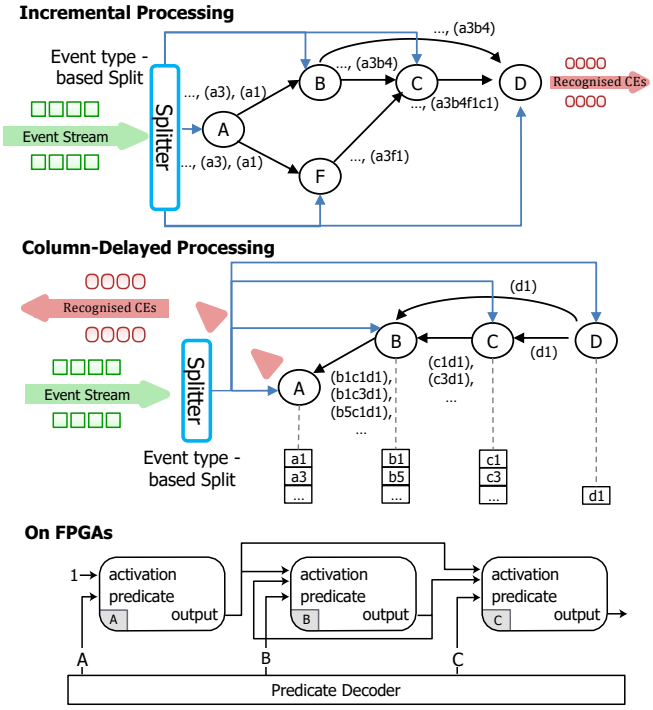


Fig. 17: State-based Parallelization [39, 60, 155]. Each state of the depicted NFA is assigned to a processing unit. Black arrows in the case of incremental processing (top) are equivalent to NFA state transition edges (D acts as merger), while Column Delayed Processing (CDP) reverses these arrows as it performs the CER operator evaluation in reverse order (A acts as merger). NFA-state 'F' is omitted from CDP for readability purposes. Event tuples of type A, B, C, D, F are marked with a_i, b_i, c_i, d_i, f_i respectively. The 'On FPGAs' (bottom) part illustrates only a subset of the corresponding hardware NFA states, for readability purposes.

pass filters and predicates among the NFA states, and thus the responsible processing units, following the NFA arrows. An example of such an incremental evaluation is depicted at the top of Figure 17, where event tuples of type A are forwarded to the processing units handling type B as well as F. Partial pattern matches make it through the NFA edges being directed to the processing unit handling C (from the processing units handling B, F) and the one handling event type D (from the unit of B). In case a full pattern match is detected at the final NFA state, a CE is produced.

The above scheme does not allow for fine tuning the load among the processing units, because it depends on the complexity of the predicates per state and the distribution of certain event types in the incoming stream. Additionally, the maximum degree of parallelism is restricted by the number of NFA states and the type of predicates. Moreover, the incremental operator evaluation that pipelines data towards the final states, also tends to assign higher pattern evaluation load to the corresponding processing units that handle those states. To understand why, consider for instance expansive event selection policies (i.e., for o input events they can produce more than o, up to o^2 complex events), such as skip-till-any-match. Furthermore, as our above discussion demonstrates, data often requires to be replicated and are always communicated among the processing units. Another limitation is that sliding windows cannot be processed in parallel using this scheme. On the bright side, state-based parallelization is the only technique in the data parallelization category that can support all event selection and consumption policies.

In order to alleviate part of the operator evaluation load from the cluster, Column-Delayed Processing (CDP) has been proposed [72, 60]. The key observation that motivates CDP is that incremental evaluation produces and forwards towards the final NFA states, partial matches that may be highly unlikely to produce a full pattern match. In the example of Figure 17, if no event of type D occurs within the specified time window, all the incrementally produced, partial pattern matches were processed in vain. To make sure that the processing load is indeed devoted to promising partial pattern matches, CDP reverses the pipelining procedure.

In particular, CDP requires that all processing units cache the incoming events of the type they were assigned, provided they pass respective filters or predicates. The cached event tuples will be further processed only when the processing unit handling the final state of the NFA receives an event of that type. Then, the operator evaluation process begins, but this time backwards. CDP is depicted in the middle of Figure 17.

Hardware-centric State-based Parallelization: An implementation of state-based parallelization using both the incremental evaluation of CER operators as well as the backtracking process of CDP was discussed in [60]. There, these versions of state-based parallelism were tested on both multi-core CPU and CUDA settings. CUDA [10] constitutes a widespread architecture for programming on GPUs.

Cache management and the transfer of events from the main memory to the GPU memory appears a critical aspect in implementing state-based parallelism over CUDA [60]. Each multi-processor creates, manages, schedules, and executes threads in groups of parallel threads called warps. The developer can take advantage of the fact that CUDA groups and computes in a single, memory-wide operation concurrent memory accesses to contiguous areas from threads having contiguous identifiers in the same warp. In a multi-core CPU environment, the implementation of state-based parallelism would opt for exploiting shared memory within a JVM, as also discussed in query-based parallelization (Section 4.1.1) and at the thread level keep only pointers to events in the shared memory. This would be efficient in an implementation on CPUs, but it would lead to memory fragmentation in a GPU setting, making it impossible to control memory

accesses from contiguous threads. Accordingly, in the CUDA implementation caches should not hold pointers to events, but copies of events.

The generic observation made in [60] is that the use of GPUs brings impressive speed-ups when CER operators are of high complexity (engaging various event types and predicates), because then the advanced processing power of the hardware overcomes the delay introduced by the need of copying events from the main to the GPU memory and vice versa. At the same time, multi-core CPUs scale better with the number of queries.

Another hardware-centric implementation of the state-based parallelism is presented in [155]. The idea there, is to insert Field Programmable Gate Arrays (FPGAs) directly into the event data path between the network interface and the CPUs of the cluster. A FPGA module includes (a) Configurable Logic Blocks (CLBs) forming a two-dimensional array, (b) Interconnections between the CLBs to implement the user logic, (c) a switch matrix which provides switching between interconnects depending on the logic and (d) I/O Pads to communicate with (in our case CER) applications. CLB contains a Multiplexer, flip flop registers and Look Up Tables (LUT) based function generators. LUT implements combinational logical functions; the Multiplexer is used for selection logic, and flip flop stores the output of the LUT.

A compiler module translates CER patterns (in the form of regular expressions in [155]) into hardware NFAs as follows. Each NFA-state and its respective incoming transition is mapped to a LUT-flip flop pair. Such pairs can work in parallel concurrently checking with a Predicate Decoder if the current tuple satisfies a given predicate (see bottom of Figure 17). The Predicate Decoder consists of pure combinatorial logic that takes a SDE as input and returns a predicate vector as output, having one bit for each predicate indicating whether it was satisfied by the SDE. This way, the NFA can decide which transitions to take next. Finally, more related to partition-based parallelism (Section 4.2.1) rather than state-based parallelism, [155] includes a stream partitioner module so that incoming data can be initially partitioned based on some identifier.

4.2.4 Graph-based Parallelization

The data parallelization approaches we have examined so far (Sections 4.2.1 – 4.2.3) process partitions of the data independently on separate processing units. Graph parallelism derives parallelism by partitioning graph-structured data across processing units and then resolving dependencies (along graph edges) through iterative computation and communication [83].

The work of [120] attempts to engage graph-parallel algorithms with the CER process. However, the proposed GraphCEP approach keeps graph parallelism detached from the CER operator evaluation itself. Therefore, in the current section we also discuss how to achieve graph-parallel CER operator evaluation.

GraphCEP [120]: GraphCEP uses two stages of parallelism. At the first stage, partition-based parallelism is applied (see Section 4.2.1). For each partition of events, a second stage of graph-based parallelism is applied for graph structured data that may be affected by incoming event tuples. These two stages are shown at the leftmost part of Figure 18. For instance, in a maritime surveillance application, the first stage may partition event tuples based on vessel type. The second stage may perform graph-parallel computations within each vessel partition such as finding closest pairs or spatio-temporal clusters of vessels. The graph-parallel stage of each operator instance in Figure 18 operates using the GAS paradigm, presented below.

GAS (Gather-Apply-Scatter) paradigm: Consider we have two graphs. The query graph and the physical network of processing units along with their communication links. Figure 18 depicts the latter graph. The query graph (not shown in the figure) is partitioned and each partition is assigned to a processing unit. Partitioning strategies using edge cuts or vertex cuts are possible [82].

Graph parallelism can be implemented by arranging the computation using gather-apply-scatter iterations where the "think like a vertex" concept is applied. During the scatter phase every vertex of the query graph sends data to its neighbors in parallel, while during the gather phase every vertex of the query graph collects data from its neighbors and aggregates it, in parallel. In the apply phase every vertex of the query graph transforms its data locally (depending on the given task). Figure 18 illustrates the GAS paradigm at the query graph partition level, i.e., showing the processing units and their links. Each vertex in Figure 18 is a processor handling a different partition of the query graph, while edges describe communication among processing units that are assigned dependent (connected through edges) graph partitions. In other words, the GAS concept is implemented both internally within a partition of the query graph that is handled by a processing unit (in which case no communication is necessary), as well as among the processing units handling different graph partitions as shown (black arrows) in Figure 18.

The query graph in CER can be the NFA used for evaluating an operator. But in GraphCEP, graph parallelism is not focused on evaluating a CER operator such as $\{;, \vee, \wedge\}$, but on combining CER with parallel computations on graphs. Below we introduce the application of the GAS idea for parallelizing the NFA-based CER operator evaluation itself.

Introducing Graph-parallel CER Operator Evaluation: Our discussion here refers to involved CER operators, which is exactly when graph parallelism is of the essence, i.e., when the processing of complex graphs needs to be scaled-out. In

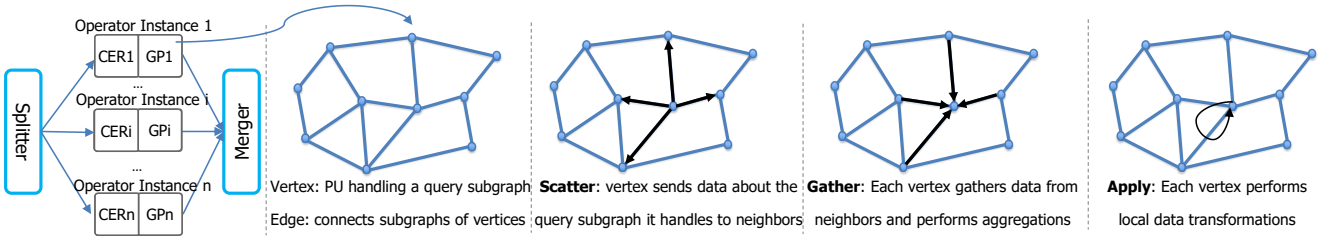


Fig. 18: GraphCEP [120] and the GAS paradigm [83]. GP stands for the Graph-Parallel part of each operator instance. PU stands for "Processing Unit".

case NFAs are used for CER operator evaluation, complexity is interpreted as a high number of states and complex organizations of edges denoting state transitions.

The plausible observation for graph-parallel CER operator evaluation is that the NFA representing the operator is itself a graph. A new event tuple may add edges and vertices to an NFA instance by triggering a transition (edge) to a new active state (vertex). Therefore, in an offline fashion, we may choose a partitioning (using edge or vertex cuts as suggested in [82]) of the NFA, termed as a sub-NFA, so that the load assigned to each processing unit is balanced and communication is minimized i.e., by employing up-to-date event statistics. Each sub-NFA can be assigned to a processing unit.

Having assigned sub-NFAs to processing units, during the gather phase all states (vertices) of the NFA receive data from their incoming edges, in parallel. These incoming edges may belong to the same sub-NFA or to another NFA partition assigned to a different processing unit. Thus, for partitions assigned to different processing units, the gather operation entails inbound communication. The gathered data may be SDEs transfered by the splitter as well as partial matches. During the apply phase, every processing unit applies filters and predicates on the received data and generates new (partial or full) pattern matches in parallel and for each state. Finally, during the scatter phase, all states of the NFA push data to their outgoing edges, in parallel. These outgoing edges may belong to the same sub-NFA or to a sub-NFA assigned to another processing unit. Thus, for sub-NFAs assigned to different processing units, the scatter operation entails outbound communication.

For NFA-based CER operator evaluation, the proposed GAS application may resemble state-based parallelization. However, notice that state-based parallelization requires that the processing of each state (event type) is assigned to a different processing unit. This is much simpler, since one does not need to seek for an appropriate partitioning of the graph (NFA) but also suboptimal, because it does not allow to balance the load and exacerbates the communication cost among the processing units. Therefore, regarding our suitability criteria (see Figure 10), graph-based parallelism maintains the same properties with state-based parallelism, but overcomes important limitations regarding load balance, need for replicating data and excessive communication.

4.2.5 Queuing Theory-based Elasticity

Queuing theory-based elastic resource allocation has been applied in the context of partition-based parallelization [119], but in principle it can also support any scheme falling in the data parallelization category. This holds because elasticity focuses purely on adaptively setting the number of running instances of an operator that will be handled by a processing unit. The idea is to model the whole parallel CER structure of Figure 15 as a queue where exponential event tuple arrivals and deterministic or exponential departures of complex events are assumed. Between the input and output of a queue, a number of processing units serving operator instances exist, as shown in Figure 19.

To decide whether fewer or more processing units should be occupied, a probabilistic input buffer limit is used. If at time t the probability of the input queue size $Q(t)$ being smaller than the buffer limit (BL), is lower than a threshold (P_{thres}), a scale-out decision is made. Thus, a processing unit that, for instance, currently handles two operator instances will keep one of them and the other operator instance will be placed at a newly occupied unit. If this probability is found to be below the threshold multiple times, the scale-in option is preferred, i.e., operator instances are placed in the same processing unit to be executed in pseudo-parallel mode. Otherwise, no action is taken. [119] does not discuss further task placement or load balance heuristics.

4.2.6 Prediction-based Elasticity

Prediction-based elastic CER [158] is another scheme that has been applied on top of data parallelism. The scheme works as sketched in Figure 20. We have a look-ahead time horizon that we split in sliding time windows, numbered from 1 to H in Figure 20. In each window we have a set of states (gray circles in the figure). A state denotes the option

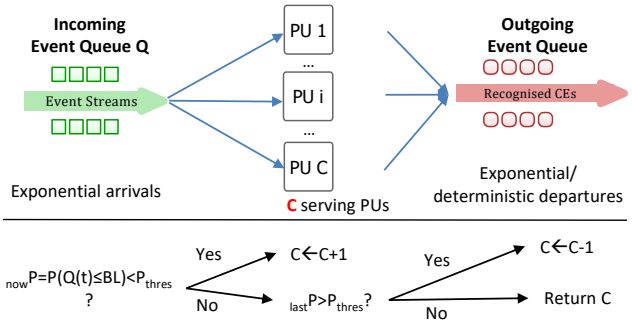


Fig. 19: Queuing Theory-Based Elastic CER [119].

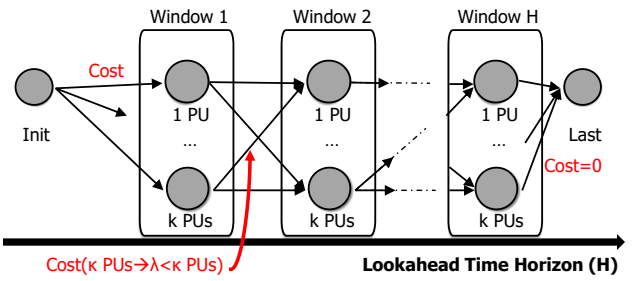


Fig. 20: Prediction-Based Elastic CER [158]. Each edge has a different weight (cost).

of utilizing $1 \leq i \leq k$ processing units for the execution of running instances of a CER operator. Transition edges connect states of contiguous, chronically ordered windows and denote a scale-in or scale-out decision depending on whether the transition is made to a state with more or less processing units. Each transition edge has a weight which is the cost of making the transition. The cost function used in [158] is a formula that incorporates the expectation about resource utilization and observed latency. The cost itself is viewed as a random variable projected to the future.

Thus, the basic goal is, given the current initial state of parallelization (with a corresponding number of processing units) and a final state at the end of the look-ahead time horizon, to choose one state per window so that the sum of the cost of the edges that connect these states is minimized. In order to compute the best combination of states, it suffices to compute the shortest path between the initial and final state of the graph as shown in Figure 20. The authors of [158] experiment with various alternatives for predicting the weight of a transition edge, including Gaussian Processes, Support Vector Machine or Neural Network-based predictions. Gaussian Processes exhibit the best prediction accuracy.

The recent approach of [159] complements [158] with a task placement scheme so that the load is balanced. [159] models the problem of balanced operator instance (and thus) placement as a Job Shop Scheduling problem, where a number of tasks should be assigned to a number of processing units, so that a metric trading-off load balance with running task migration cost is minimized. A dynamic task placement algorithm is presented based on a well-known greedy approximation of the Job Shop problem.

4.3 Summary of CER Parallelization Scheme Properties and Hybrid Schemes

The discussion presented so far showed that there is no one-size-fits-all solution when it comes to parallelizing CER. With respect to our suitability criteria, presented at the beginning of Section 4 and in Figure 10, we summarize the properties of the various parallelization schemes in Table 3.

Depending on the CER operator or query it is often necessary to resort to hybrid parallel CER approaches to maximize throughput, reduce computational latency, and better balance the load. For instance, a CER query may group events based on some key such as those shown in Figure 15 and simultaneously impose a sliding window over the incoming events that are to be taken into account in pattern matches. Such a query may use a hybrid parallelization strategy that combines partition-key based and pattern-sensitive parallelism (Section 4.2.1). The question that arises, is how the properties of the overall parallelization scheme evolve upon using such a hybrid approach. We provide guidelines on this issue.

The expected behavior of a hybrid scheme is marked in the last column of Table 3. With respect to event selection and consumption policies, if one of the parallelization schemes engaged in the hybrid approach does not support a policy, then the overall hybrid approach cannot support it. This is denoted by a logical AND in the last column of Table 3. On the other hand, for achieving sliding window parallelization, load balance, limited event data replication or communication during the parallel execution, it suffices for one of the techniques involved in the hybrid scheme to possess such a property for the overall scheme to support it. This is denoted by a logical OR in the last column of Table 3. Whether combinations of parallelization schemes are possible or not depends on the specifications of the CER query or operator. For instance, one cannot mix operator-based parallelization and run-based parallelization if the parameter S used in the latter cannot be set to a constant (see Section 4.2.2).

Regarding event selection and consumption policies, their support depends on how data is partitioned, excluding the case of operator-based parallelization where query rewriting prevents the use of skip-till-any-match policy, for the reasons discussed in Section 4.1.2. Therefore, if at least one parallelization scheme, partitions data or rewrites the query so that a policy cannot be supported, encapsulating the rationale of that scheme in another one (or vice versa) cannot amend this property.

Criterion		Task Parallelism		Data Parallelism					Hybrid
		Query-based	Operator-based	Partition Key-based	Pattern Sensitive	State-based	Graph-based	Run-based	
Selection Policies	Sc	✔	✔	✘	✘	✔	✔	✔	
	Pc	✔	✔	✔	✔	✔	✔	❐	A
	Stnm	✔	✔	✘	✘	✔	✔	✔	
	Stam	✔	✘	✘	✘	✔	✔	✔	N
Consumption Policies	Co	✔	✔	✔	✘	✔	✔	❐	
	Re	✔	✔	✔	✔	✔	✔	✔	D
	BRe	✔	✔	✔	✘	✔	✔	❐	
Window Parallel	TuW	✘	✘	✘	✔	✘	✘	✔	
	TiW	✘	✘	✘	✔	✘	✘	✘	O
Agility	LB	✘	✘	✘	✘	✘	✔	✔	
	Rep/Com	✘	✘	✔	✘	✘	✔	❐	R

Table 3: Summary of CER Parallelization Schemes' Properties. The ✔ and ✘ marks in each column denote that the criterion is supported or not, respectively, by the corresponding parallelization scheme. ❐ denotes that the corresponding criterion is supported under certain circumstances. For instance, in Section 4.2.2 we argued about the fact that run-based parallelization does not directly support the consume and bounded-reuse consumption policies, but since data is replicated to at most two processing units, these units can impose consume and bounded-reuse by limiting communication only among the involved pair. Stam : skip-till-any-match, Stnm : skip-till-next-match, Sc : strict-contiguity, Co : consume, Re : reuse, BRe : bounded-reuse, TuW : Tuple-based Window, TiW : Time-based Window, LB: Load Balance, Rep/Com : Data Replication/Communication.

The next criterion regards the capability to parallelize windows. To attribute such a property to a hybrid scheme at least one of pattern-sensitive or run-based scheme should be adopted. Therefore, the logical OR in the last column of Table 3.

The Load Balance, Replication and Communication criteria require some additional commentary on why it suffices for at least one of the parallelization schemes in a hybrid approach to possess these properties, for the overall scheme to maintain them i.e., OR in the last column of Table 3. Let us first elaborate more on how OR is interpreted for Replication and Communication. For instance, partition-key based parallelization does not replicate data by imposing partition isolation, but pattern-sensitive parallelization requires data replication for overlapping windows, or state-based parallelism creates multiple copies of events and partial matches during incremental/backtracked operator evaluation. Thus, OR in the respective cell of Table 3 seems a bit counter-intuitive. The observation here is that since data is first partitioned based on a key, the second level of parallelism creates partitions internal to a key partition. Therefore, the hybrid scheme can be configured to run all the second level partitions of the data related to a given key, on the same machine or JVM. Thus, overlapping windows for the same key can be co-hosted and shared memory will be used to hold actual events, while only pointers can be kept at threads. Data does not need to be replicated or hit the network card. Similar observations can be derived for other combinations.

Finally, regarding load balance, consider for instance, a second level, run-based parallelization scheme is encapsulated in a first level partition-key based scheme. The first level scheme does not provide native support for load balancing as the load of an operator instance depends on the distribution (frequency) of the respective keys. Now assume that the key distribution is skewed and only few, very frequent keys appear in the incoming event stream. Applying the second level parallelism will re-partition the initially large partition of events of a certain key and create smaller, equally sized partitions of the data, therefore fairly distributing the load among the processing units. In case graph-based parallelism is imposed as the second level scheme, load balance is achieved by appropriately partitioning the entire NFA graph, which this time is instantiated for each input key. Similar observations can be derived for other combinations.

4.4 Open Issues in Parallel and Elastic CER

Graph-Parallel CER: In Section 4.2.4 we commented on the absence of graph-parallel schemes tailored to CER and introduced the application of the GAS idea to NFA graph-based CER operator evaluation. A key observation we make is that a CER query or operator may itself be interpreted as a complex graph other than a NFA, the processing of which needs to be parallelized. Such a graph may be a Petri Net [42], a Markov Logic Network [146], a tree or a (stratified) logic program such as those discussed in Section 2. The introduction of a graph-parallel scheme tailored to CER can be quite useful when CER operators need to be evaluated over such graphs. This is because the GAS rationale enables the parallel processing of complex graphs, when the other data parallelization schemes we examined are difficult or impossible to be implemented, due to complex relationships among operators and operator instances, that make it hard to first partition the input and then merge the outcomes of the parallel processing. For instance, none of the data parallelization schemes we described in this section allows the parallel

processing of CER operator instances that are evaluated over Markov Logic Networks, stratified logic programs or Petri Nets in the general case, since it is non-trivial to even appropriately construct splitters and mergers (see Figure 15). Thus, graph-parallel schemes tailored to CER is an open area for further research as a key enabler for scaling-out CER beyond automata-based techniques.

The Need for Rules of Thumb: In Section 4.3 we presented, to our knowledge for the first time, the properties attributed to CER when choosing a hybrid CER parallel scheme synthesizing some of the basic schemes we examined. In the last column of Table 3 we presented properties of hybrid parallelizations schemes. We also commented that for a hybrid approach to support, for instance, an event selection strategy; all the engaged parallel CER schemes should do the same. Nevertheless, having commented on the properties of such hybrid schemes, the open research issue that remains is the identification of rules of thumb about when the use of a hybrid approach is worthwhile. Such rules will indicate when the choice of a hybrid scheme outperforms each of the parallel CER approaches that are being synthesized and to which extent. For instance, a rule of thumb may say that if the reuse consumption policy is used and a data parallelization scheme is applied for 20% of the most CPU demanding query operators, while the remaining 80% of CER operators are executed on operator-based parallelization, the overall throughput of the CER process will increase by 75%. Such a rule of thumb has a great impact in practice since it cuts down the time and effort devoted to fine tuning multiple CER queries by incorporating prior knowledge extracted out of benchmarks and experiences in real-world application settings.

CER-oriented Elastic Resource Allocator: There is no elastic resource allocation scheme that is indeed tailored to CER. Indicatively, at the current status of related works on elastic CER, event selection and consumption policies are not taken into account. This is despite the fact that an elastic CER resource allocator needs to account for the migration cost, which is affected by such policies. The migration cost of a CER operator under strict-contiguity is likely to differ significantly from the cost of the same operator being executed under skip-till-any-match (the latter can produce a much larger set of partial pattern matches). Complexity analysis can be performed a priori to narrow down the search space and guide heuristics on which CER tasks are preferable to move.

Similarly, when an elastic CER allocator examines if it would be preferable to create multiple operator instances for an operator previously running in one instance, it can take into account the operator's consumption policy to properly estimate migration (state transfer) costs. For instance, event tuples under a bounded-reuse policy should probably not be replicated among different host machines with tasks that require them. This is because under such a policy, all hosts need to communicate with each other anyway before consuming a tuple, in order to check if the bounded-reuse constraint is violated, i.e. if other tasks consumed the tuple the permitted number of times. Therefore, since communication among machines will take place anyway, an elastic resource allocator should instead keep these tuples only at the host of the initial, single operator instance, which can count (via receiving reports) the number of times these tuples have been consumed among the newly introduced operator instances. In any case, it makes sense to examine replication versus communication cost trade-offs under a bounded-reuse policy.

The development of an elastic resource allocator specialized for CER is particularly useful in cloud environments. An elastic CER resource allocator at the side of the cloud provider can act as a CER optimizer which monitors and adapts the number of reserved resources, such as virtual machines, to achieve fair monetary costs under the pay-as-you-go pricing model. Keeping computational latency under certain thresholds is important as well to ensure compliance with Quality-of-Service (QoS) criteria defined on Service Level Agreements (SLAs) between clients and cloud providers. At the client side, an elastic CER resource allocator can minimize monetary costs simultaneously abiding by computational latency constraints of time-critical applications.

CER Task Placement Algorithms: The current state-of-the-art in driving the decision of CER task placement to processing units uses simple and potentially suboptimal heuristics such as the First Fit Bin Packing [91,93,90,92] or a Job Shop Scheduling [159] heuristic. In all such cases the problem is NP-hard and therefore greedy heuristics are used. The question that is yet to be answered is whether certain CER characteristics such as event selection policies allow for greedy algorithms providing a constant factor approximation of the optimal task placement. For selection strategies such as skip-till-any-match it may be the case that no greedy algorithm can allow a constant factor approximation of the optimal task placement. However, restrictive selection policies such as strict-contiguity or partition-contiguity are more likely to provide such an approximation.

5 Geographically Distributed CER

Our discussion so far has focused on scalable CER within clustered architectures (data centers, cloud) where Big Data stream-in and, in order to recognize CEs in a timely manner, parallel and elastic CER techniques are applied. Hence, throughput is maximized and computational latency is harnessed. Nonetheless, SDEs, more often than not, are not produced within computer clusters or just appear there. There is a network of data gathering devices, placed in application fields of interest, that collect and relay relevant information

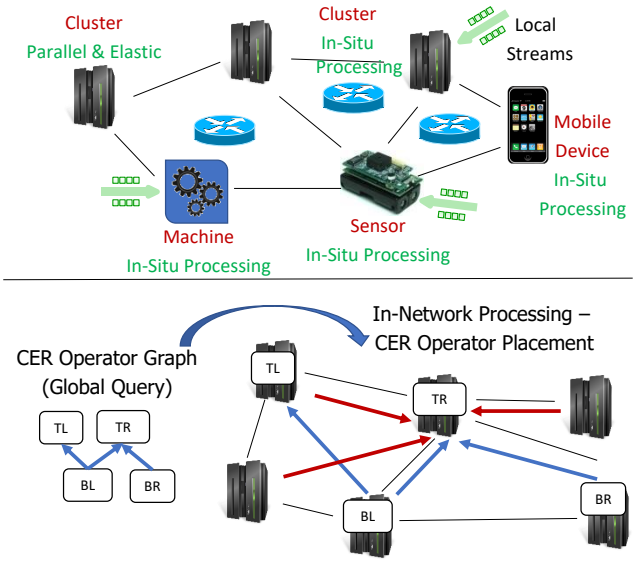


Fig. 21: Geographically Distributed Architectures - In-Situ & In-Network Processing. As an instance, consider BR := SEQ(e_1, e_2), BL := AND(e_3, e_4), TL := AND(BL, e_5) and TR := SEQ(BR, BL, e_6). SDEs $e_1, \ldots, e_6$ are omitted from the CER operator graph for readability purposes. Blue arrows indicate communication due to operator dependence (one operator provides input to another) according to the CER query graph. Red arrows show an example of the corresponding sites contributing relevant to the operator event data (such as SDE e_6) that surpass in-situ filters.

towards clustered architectures. In massive-scale scenarios, multiple data centers or clouds may even be part of this, geo-distributed, architecture.

Consider, for instance, a maritime monitoring scenario where position signals in the form of AIS messages originate from multiple vessels. Certain base stations or even satellites initially receive such messages and later forward them to shipping companies', authorities' or ship tracking intelligence ventures' potentially geographically distributed data centers. Additional examples of geographically distributed architectures involve, but are not limited to, smart energy grids which may span one or more countries, Industry 4.0 settings, networks of ATMs in banking and networks of smartphones in telecommunication scenarios.

In all the above cases, naive event data centralization is not a viable option. Since CER queries continuously run for protracted periods of time, constantly centralizing data at a single site before applying parallel and elastic CER would require excessive communication in the first place. Therefore, sooner or later communication links shall become unresponsive, hindering the timely delivery of event data and, at the cluster side, the timely execution of CER analytics. Prior work [160] has pointed out that the maximum stream processing rate that can be achieved in such distributed settings is network bound. Therefore, what is important in geo-distributed CER is to reduce communication in the network and control the network latency. Besides, returning to our discussion in Section 4.4, the pay-as-you-go model also applies to communicated data [125, 84, 30], while QoS criteria related to network latency and compliance with SLAs are also of the essence.

Therefore, in this section we discuss techniques for enabling CER to scale one step further out, from clustered architectures discussed in Section 4 to fully distributed settings composed of data gathering sites, relay sites and multiple clusters or clouds, as shown in Figure 21. In principle, there exist two generic tools for enabling communication efficient CER over geo-dispersed settings: (a) in-situ processing, which involves the installation of local filters at sites so that they communicate local data within the scope of CER analytics, only when it is absolutely necessary, in which case the corresponding filter is satisfied (see the top of Figure 21), (b) in-network processing, i.e., instead of collecting all streaming event tuples that satisfy in-situ filters at a single site, CER operators are evaluated in network, at various sites near the event sources, so that event data are aggregated early and only the aggregated information is further forwarded towards the CER query source. The rationale is depicted at the bottom of Figure 21, where an abstract CER query graph, representing the global query posed over the distributed architecture, is mapped to the physical network of sites. Each operator is assigned and evaluated at a particular site as shown in the figure. Thus, in-network operator placement expresses a mapping (assignment) of CER operators to clusters.

In principle, in-network processing via operator placement can be performed as in traditional geo-distributed stream processing systems [132, 141, 135, 46, 106, 56, 148]. The main difference in geo-distributed CER comes from the in-situ processing rationale accepted by CER operators and the ability of the proposed techniques to combine such CER-tailored in-situ filters with in-network operator placement. Based on this, as well as on service-oriented (pricing, QoS) parameters mentioned earlier in our discussion, we evaluate related approaches, with respect to their suitability for geo-distributed CER relying on the following criteria, summarized in Table 4:
[A] Algorithmic Criteria: the goal and usefulness of in-situ and in-network processing has already been analyzed in our previous discussion. Therefore, suitable techniques should effectively blend:

- **[A1]** In-network processing - operator placement.
- **[A2]** CER-tailored in-situ processing.

[B] Service-Oriented Criteria: Minimizing the communication cost of geo-distributed CER loosens the network bound on the maximum stream processing rate [160]. Moreover, pay-as-you-go network pricing in modern cloud platforms entails that communication cost minimization also aids in

	Criterion			
	Algorithmic		Service	
Related Work	A1: In-Network	A2: In-Situ	B1: Network Pricing	B2: QoS Network Latency
Akdere *et al* [24]	✘	✔	✔	✔
Cardellini *et al* [46], SODA [154]	✔	✘	✘	✔
SAND [23], SBON [132], SPADE [78], DistCED [133], FAIDECS [153], DHCEP [142], Kumar *et al* [106], Rizou [141]	✔	✘	✘	✘
SIENA [48], Gryphon [22], Cordies [103], PADRES [110]	✘	✘	✘	✘
Iridium [135], JetStream [136]	✘	✘	✔	✔
SQPR [97], Geode[148]	✔	✘	✔	✘
Amini *et al* [31], Repantis *et al* [139], Benzing *et al* [41]	✘	✘	✘	✔
FERARI [75]	✔	✔	✔	✔

Table 4: Geo-distributed CER approaches versus suitability criteria.

minimizing monetary costs. Compliance with SLAs adds up network latency constraints to the problem. Therefore, suitable techniques for geo-distributed CER should seek to optimize a **constrained, bi-criteria optimization** problem including communication/network pricing cost ([B1]) and network latency related Quality-of-Service (QoS) ([B2]):

- **[B1]** Network Pricing: To minimize costs in the pay-as-you-go model, minimization of intra- and multi-query communication by CER operators is a prerequisite.
- **[B2]** Network-Aware QoS criteria: Network latency-constrained optimization allows compliance with QoS criteria defined on SLAs or requirements of time critical applications. Latency-constrained optimization is often not supported as shown in Table 4.

Both in-situ and in-network processing are algorithmic prerequisites for geographically distributed CER. Merely incorporating just one of [A1],[A2] can yield severely suboptimal communication and network latency performance [62]. From a service viewpoint (i.e., [B]), suboptimality results in overcharges in the pay-as-you-go model. To justly lower charges as much as possible, but also abide by QoS constraints, [B1][B2] should be supported.

5.1 CER-Oriented In-Situ Processing

The main tool for installing local filters on individual sites in the context of CER, is based on the push-pull rationale. The push-pull logic has been introduced in the context of geo-distributed CER in the work of [24], but has been used in other CER contexts as well [143, 152]. According to [24] all CER operators of a given query are assigned and get evaluated at a central site. Thus, the technique lacks support for in-network processing (criterion [A1]). Given a CER operator $\mathsf{OP} \in \{;, ,, \wedge\}$, implying that push-pull makes sense only for CER operators that require all their input events to produce a full pattern match, the local filters are installed in every

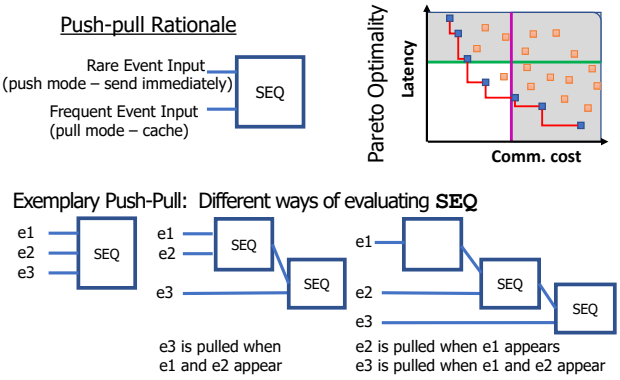


Fig. 22: Push-pull Rationale on $\mathsf{SEQ}(e_1, e_2, e_3)$.

site contributing input event instances for OP. The local filter itself says that one or more of the most rare event types participating in OP are marked in push mode, in which case corresponding event tuples are communicated to the central site as soon as they appear. On the contrary, more frequent event types are marked in pull mode which means that sites cache the corresponding event tuples, until either the central site reverses the mode of these event types to push, or the tuples expire due to window constraints. Communication savings are ensured because frequent events are not transmitted before the apparition of rare ones and, in the meantime, some of them expire. On the other hand, setting event types in pull mode means that tuples are communicated on-demand instead of being instantly flashed to the central site. Therefore, setting event types to pull mode increases network latency.

Different push-pull alternatives can be prescribed by increasing the number of pull steps or the ordering of event types at each step. The number of steps cannot exceed the number of event types that are input for the CER operator. The bottom of Figure 22 shows some, but not all, push-pull alternatives for the distributed evaluation of a pattern $\mathsf{SEQ}(e_1, e_2, e_3)$. From left to right, the first option is to set

all e_1, e_2, e_3 in push mode. Such a strategy essentially does not impose an in-situ filter and all events are transmitted to the central site as soon as they appear somewhere in the network. As shown in the middle, bottom of Figure 22 another alternative is to set e_1, e_2 in push mode and e_3 in pull mode. This alternative entails one pull step. By doing so, e_3 will be cached at the sites it is produced, while e_1, e_2 will be directly communicated to the central site. Finally, on the right, bottom of Figure 22, an alternative that uses a two step pull is shown. More precisely, e_1 is set in push mode, if the central site receives e_1, e_2 is pulled, while on e_2's occurrence the central site pulls e_3.

As mentioned above, increasing the number of pull steps has greater potential for reducing communication and increasing network latency. Hence, Akdere et al [24] seek Pareto optimal solutions to a constrained bi-criteria $(comm\_cost, network\_latency)$ optimization problem as shown at the top of Figure 22. The two axes in the graph represent communication (and potentially monetary) cost and network latency dimensions. Pareto optimal solutions form the marked skyline. This skyline is composed of push-pull alternatives that are not dominated in both $(comm\_cost, network\_latency)$ dimensions from any other push-pull alternative. The part of the skyline included in the shaded areas in the graph corresponds to Pareto optimal solutions that are pruned due to network latency (horizontal) or communication (monetary) cost (vertical) constraints. Hence, as marked in Table 4 Akdere et al [24] supports [A2],[B], but falls short with respect to [A1] because it simply collects events that pass in-situ filters at the query source without in-network processing.

5.2 In-Network Processing - CER Operator Placement

Techniques for in-network operator placement share the goal of picking the proper network site for assigning the execution of an operator. However, the way these sites are picked can determine their suitability for distributed CER with respect to communication and monetary cost minimization, and their abidance by network latency constraints. Here we provide an overview of in-network processing approaches and discuss their ability to comply with criteria [A],[B].

5.2.1 Geo-distributed CER Approaches

There is a number of approaches on geo-distributed CER including SIENA [48], Gryphon [22], Hermes [133], PADRES [110] and the more recent works of Cordies [103], DHCEP [142] and FAIDECS [153]. SIENA and Gryphon do not consider in-network aggregation of events, but have focused on the efficient routing of SDEs by reducing the communication costs between clients and brokers in pub/sub systems, thus avoiding the flooding of events to all subscribers. Consequently, they do not comply with criteria [A] and [B]. Hermes (a.k.a DistCED) uses a Distributed Hash Table (DHT) to determine in-network operator placement, while FAIDECS employs Hermes's DHT for the same reason. However, as discussed in [132] DHT tables minimize the hop count as opposed to network latency or bandwidth. Thus, DHT routing paths lead to inefficient in-network placements and respective techniques fall short with respect to criteria [A2], [B]. Although PADRES and Cordies opt for optimizations involving network traffic and routing delay, they neither take into account any network-, latency- or system-specific information, nor provide any specific algorithmic suite for operator placement. Thus, they cannot support [A],[B].

DHCEP, which neglects [A2], uses network usage in its optimization process. Network usage is defined as the sum of products of $dataRate \times latency$ on communication links. This is a popular metric also in distributed stream processing techniques, which are discussed below. However, using such a blended metric does not allow for latency-constrained optimization and network pricing separately. Therefore such techniques cannot adapt to criterion [B]. Despite the fact that DHCEP [142] extensively talks about heavy constraints in the optimization process, all these constraints involve processing, security or domain restrictions, but not bandwidth or latency separately (i.e., [B1] and [B2]).

5.2.2 Broader Geo-distributed Stream Processing

One of the earliest works in operator placement for geo-Distributed Stream Processing (DSP) is the work of [23], which however is DHT-based. Such techniques cannot provide latency-constrained optimization and communication or monetary cost minimization. The work of SBON [132] seeks to optimize a sum of a quantity similar to network usage, that is $dataRate \times latency^2$. Apart from the problematic behavior of this metric (e.g. when the length of a path is doubled, the latency quadruples), it cannot support network latency-constrained optimization and communication or monetary cost minimization. The same holds for efforts [106, 141] that employ a similar utility or usage metric. Although [106, 141] and [56] claim to support latency constraints, this comes after [106, 141] or before [56] having determined operator placement instead of pruning infeasible solutions while exploring the search space. [46, 47] propose a general formulation of the optimal DSP placement which takes into account network resources and encompasses different solutions proposed in the literature. Nonetheless, it does not optimize operator sharing falling short in [A2], [B1].

The recently proposed JetStream [136] trades-off network bandwidth minimization with timely query answering and accuracy, but while exploring in-network operator placement solutions it restricts itself to the MapReduce rationale (placement on source nodes), nearest site of relevant data presence or central location. This is only a set of straightforward solu-

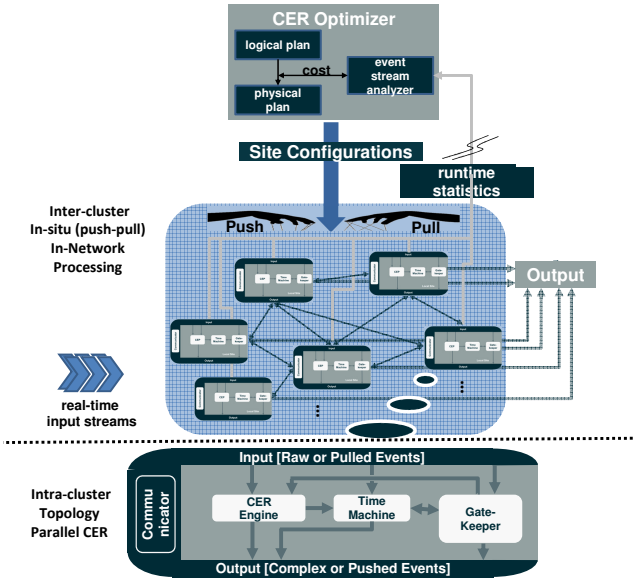


Fig. 23: FERARI Architecture [75].

tions that any technique can accommodate and a restricted subset of possible in-network placement solutions. Iridium [135] assumes control over where relevant data is transfered and moves these data around sites to optimize query response latency. Wise network resource usage is only accounted for during data placement instead of operator execution. Moreover, its network-related metrics account for data rates in uploading and downloading data, but not for intra-query communication costs where one operator placement decision affects upstream/ downstream operators. Finally, Geode [148] focuses on minimizing bandwidth cost and does not account for latency.

5.3 In-Network, In-Situ & Parallel CER

The FERARI framework [75] enables CER over multiple clusters or clouds by covering criteria [A],[B] in their entirety. Geographically distributed CER is optimized by a query optimizer that receives as input the CER query and determines the in-network placement of CER operators patched with a proper push-pull strategy. To achieve that, FERARI enhances the algorithms of [24] with in-network placement. It seeks for overall Pareto optimal solutions in terms of both in-network processing and push-pull strategy not only per operator, but also among operators shared by multiple queries.

Contrary to [24] which requires every event type to be observable by every site in the network, FERARI allows different sites to observe only a subset of the event types that participate in the posed CER query and the FERARI optimizer uses this information during in-network CER operator placement. As shown at the upper part of Figure 23 the key component of FERARI is an optimizer module. The optimizer utilizes statistics of the frequency of each event type (involved in running queries) per site, as well as latency measurements for communication links. Based on the gathered statistics, the best, in terms of Pareto optimality $(comm\_cost, network\_latency)$, execution plan, composed of CER operators' in-network placement at sites as well as push-pull strategy, is picked.

FERARI's intra-cluster processing is built on Apache Storm. Each site is assumed to run a Storm topology. A site's Storm topology, shown at the bottom of Figure 23, is comprised of the following components:

Input Spout: A Spout where streaming tuples arrive or pushed events from other sites are fed into the CER Engine.

CER Engine: It receives the input events from the Input Spout and, having processed them according to the CER operator placed at its site, emits derived events towards the Time Machine component. The CER Engine itself is composed of a number of Bolts.

Time Machine: A Storm Bolt that caches events in pull mode and CEs from the CER Engine and deals with out of order issues as those discussed in Section 3.4 and further analyzed in Section 6.

Gatekeeper: A Storm Bolt responsible for handling generic streaming operators (such as aggregations).

Communicator: A Storm Bolt responsible for the push/pull based communication to/from sites.

The CER Engine is ProtonOnStorm [14]. The parallelization rationale resembles partition-based parallelism (see Section 4.2.1). Upon the reception of an event tuple, multiple independent parallel instances of a Routing Bolt, determine the metadata that should be assigned to the tuple, the CER operator name and the context name, which are added to it. ProtonOnStorm uses the Apache Storm's field grouping option on the metadata routing fields - the agent name and the context name - to route the information to a Context Processing Bolt. The event tuple is then processed by the context service and the relevant context partition id is added to it. At this point, ProtonOnStorm uses the field grouping on context partition and agent name fields to route the event to specific instances of the relevant CER operator. If a CE that needs to be transmitted to a remote cluster or cloud is detected, it will be routed to the Time Machine Bolt and it will be pushed to the site where in-network processing has assigned the CER operator.

5.4 Open Issues in Geo-distributed CER

Geo-distributed CER Optimization Algorithms: Although the need to combine in-situ and in-network processing in geo-distributed CER has been demonstrated in FERARI [75], the actual algorithms have not been provided. Thus, still, there seems to be no published algorithm that combines the two

optimizations. The task is non-trivial since for any given in-network placement there is a number of alternative in-situ filters that may be materialized. Exploring the space of all possible combinations requires (i) a modeling of the setup as a proper optimization problem, (ii) optimal, but expectedly computationally demanding, algorithms for solving this problem and (iii) efficient greedy, heuristic algorithms to reduce the computational complexity of the optimal ones.

Adaptive Geo-distributed CER: Even when a proper combination of in-network CER operator placement and in-situ (push-pull) processing is determined, this decision needs to be adapted at runtime as an optimal combination at the current time may not remain optimal in the long run. Such adaptive geo-distributed CER schemes are needed to practically maintain low communication costs and restricted network latency. Respective algorithms should provide decision making mechanisms for taking adaptation decisions and cost formulas quantifying the expected benefit of adaptation, taking into consideration CER operator migration (to other sites) costs and/or in-situ filter alteration. Moreover, issues and practical aspects regarding the combination of adaptive (as previously described) and elastic CER at the intra-cluster level are not accounted for in the literature and thus constitute open research issues.

Multi-Constrained Optimization: In real-world settings network latency constraints are not the only ones a geo-distributed CER prototype should account for. Intra-cluster capacity constraints should also be taken into consideration. For instance, if in-network processing assigns multiple heavy load CER operators to a site, the memory or CPU capacity of the site may not be enough to carry out their evaluation. Besides, network latency is not the only dimension that may affect alignment with SLAs and QoS requirements. Network congestion is another real-world aspect that affects QoS. The difficulty in this case is that the in-network and in-situ combinations that have been chosen may cause themselves traffic, leading to network congestion. Then, the cost formulas for determining the geo-distributed CER evaluation fail. Techniques which can account for network congestion by incorporating (inbound, outbound) bandwidth capacity limits per link during geo-distributed CER evaluation are also needed.

6 Out Of Order Processing Issues

Network latencies, communication or machine failures may cause events to arrive out of order both at geo-distributed networks of sites or within a cluster (site) implementing CER on Big Data platforms. Table 5 summarizes the categories of techniques handling out of order streams and their properties. Out of order arrivals can affect the accuracy of the CER process while imposing selection and consumption policies, as well as the computation of windows of events.

	Property			
Method	Cache Usage	CPU Usage	CE Inaccuracy	Timely CEs Delivery
Slack	✘	✔	✔	✘
Compensation	✘	✘	✘	✔
Checkpoint	✘	✔	✔	✘
Approximation	✔	✔	✘	✘

Table 5: Properties of out of order processing approaches. A ✘ mark in a table cell means this property is considered a drawback. For instance, compensation-based techniques are CPU intensive as they recompute CEs in case of out of order events arrive. A ✔ refers to advantageous properties, i.e., slack-based techniques cannot produce (even temporarily) inaccurate CEs. Timely CE delivery means that CE production cannot be delayed if events arrive in order.

In Section 3.4, we mentioned that some CER on Big Data platforms alternatives support only simplified versions of *slack-based* out of order event handling [37,96]. The basic idea behind such techniques is to cache relevant events and defer the execution of order-critical operators for a specified slack interval, so that even delayed events should have arrived before operator evaluation begins.

To avoid the delay of slack-based techniques in CE delivery, *compensation-based* techniques [34,45,113,51,127] have been proposed. The rationale is that order-critical operators are evaluated as if no out of order issues exist. Relevant events used in the evaluation process are cached for a specified slack interval. This cache is used for checking out of order arrivals. Upon such an arrival, the operator is re-evaluated, which incurs additional CPU usage. Thus, the CER application is compensated for inaccurate CEs [113] later on.

Checkpoint-based techniques [113,51] focus on discovering event tuples and timepoints at the event stream, beyond which no out of order issues need to be examined. For instance, Liu et al [113] define such a checkpoint via a Partial Order Guarantee (POG) model which exploits event type metadata to deduce that events of a certain type cannot occur after a certain timepoint. For illustration purposes, consider a maritime monitoring scenario. Based on a vessel's maximum speed metadata, a nearCoast event may not be possible to occur if the last location update of a vessel is sufficiently far from the coastline. Similarly, if a nearPort event occurs, there may have been a nearCoast event that has not arrived yet. Checkpoint-based techniques reduce cache (buffer) usage in a best effort (based on available metadata) way, but delay CE delivery similar to slack-based techniques.

Approximation-based techniques [38,109,57,140] either exploit recent event (SDE and CE) history to produce the most likely approximate results quickly and potentially com-

pensate for inaccuracies later on [38,109,140], or employ stream summarization [57] to limit buffer usage and generate approximate events with a priori defined accuracy guarantees. Hence, such techniques limit buffer and CPU usage to partly report CEs in a timely fashion, but the CER application may not be compensated later on for unreported CEs.

Further research on handling out of order arrivals is of particular interest in the scope of future benchmarking efforts as those discussed in Sections 3.4 – 3.5 and Table 2.

7 Research Challenges Beyond Scalability

Throughout our study we elaborated on open issues and future research directions in a per section fashion. Here, we list open issues that touch important aspects beyond the scope of our survey. First, there is a need for a formal comparison of language expressiveness and recognition complexity. Some steps towards this have already been taken [85,162]. This way, it would be possible to identify the appropriate language (subset) addressing the requirements of a given application.

Second, there is also a need to adjust the parallelization and distribution techniques presented in this paper, for handling the *lack of Veracity* of Big Data. Some very preliminary steps towards this directions stem from [150].

Third, more often than not, the rules that dictate the patterns which constitute interesting CEs are folded in correlations, interconnections and trends in high velocity streams originating from a variety of data sources. In dynamic settings there is no way to extract those rules timely and in an online fashion manually. Therefore, automatic discovery of such patterns is imperative. Although there are efforts towards online discovery of CER rules [100,101], concept drifts should be detected and the set of monitored rules should get updated to avoid monitoring obsolete patterns.

Finally, approximate CER may take the form of (a) forecasting a full pattern match [26,25], before it takes place or (b) operating on a summary of SDEs so that large part of the workload is shed provided quality guarantees can be derived. The goal of (a) is to proactively respond to monitored events, while the goal of (b) is to enable scalability at extreme scales. However, complex event forecasting and approximate CER operating on specialized event data summaries is still in its infancy [111]. Therefore, both aspects of approximate CER constitute an open field of study and research.

8 Acknowledgments

This work has received funding from the EU Horizon 2020 research and innovation program INFORE under grant agreement No 825070.

References

1. Apache Flink v. 1.7. https://flink.apache.org/. [Online; accessed 31-March-2019].
2. Apache FlinkCEP. https://ci.apache.org/projects/flink/flink-docs-stable/dev/libs/cep.html. [Online; accessed 31-March-2019].
3. Apache Flume. https://flume.apache.org/. [Online; accessed 31-March-2019].
4. Apache Kafka. https://kafka.apache.org/. [Online; accessed 31-March-2019].
5. Apache Spark Streaming. https://spark.apache.org/docs/latest/streaming-programming-guide.html. [Online; accessed 31-March-2019].
6. Apache Spark Structured Streaming. https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html. [Online; accessed 31-March-2019].
7. Apache Spark v. 2.4.0. https://spark.apache.org/. [Online; accessed 31-March-2019].
8. Apache Storm v. 2.0.0. http://storm.apache.org/. [Online; accessed 31-March-2019].
9. Bringing complex event processing to spark streaming. https://www.youtube.com/watch?v=naCRk9wAd6g. [Online; accessed 31-March-2019].
10. CUDA Zone. https://developer.nvidia.com/cuda-zone. [Online; accessed 31-March-2019].
11. Esper. http://www.espertech.com/esper. [Online; accessed 31-March-2019].
12. Esperonstorm. https://github.com/tomdz/storm-esper. [Online; accessed 31-March-2019].
13. Ibm proactive technology online. https://github.com/ishkin/Proton/tree/master/IBM%20Proactive%20Technology%20Online. [Online; accessed 31-March-2019].
14. Ibm proactive technology online on storm. https://github.com/ishkin/Proton/tree/master/IBM%20Proactive%20Technology%20Online%20on%20STORM. [Online; accessed 31-March-2019].
15. Jess, the rule engine for the java platform. https://www.jessrules.com/jess/docs/71/. [Online; accessed 31-March-2019].
16. Oracle cep cql language reference. https://docs.oracle.com/cd/E16764_01/doc.1111/e12048/intro.htm. [Online; accessed 31-March-2019].
17. Sase source code. https://github.com/haopeng/sase/. [Online; accessed 31-March-2019].
18. Siddhi CEP. https://github.com/wso2/siddhi. [Online; accessed 31-March-2019].
19. Storm compatibility beta. https://ci.apache.org/projects/flink/flink-docs-stable/dev/libs/storm_compatibility.html. [Online; accessed 31-March-2019].
20. Stratio Decision. https://github.com/Stratio/Decision. [Online; accessed 31-March-2019].
21. J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. Efficient pattern matching over event streams. In *SIGMOD*, 2008.
22. M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching events in a content-based subscription system. In *PODC*, pages 53–61, 1999.
23. Y. Ahmad and U. Çetintemel. Network-aware query processing for stream-based applications. In *VLDB*, pages 456–467, 2004.
24. M. Akdere, U. Çetintemel, and N. Tatbul. Plan-based complex event detection across distributed sources. *PVLDB*, 1(1):66–77, 2008.
25. E. Alevizos, A. Artikis, and G. Paliouras. Event forecasting with pattern markov chains. In *DEBS*, pages 146–157, 2017.

26. E. Alevizos, A. Artikis, and G. Paliouras. Wayeb: a tool for complex event forecasting. In *LPAR*, 2018.
27. E. Alevizos, A. Skarlatidis, A. Artikis, and G. Paliouras. Probabilistic complex event recognition: A survey. *ACM Comput. Surv.*, 50(5):71:1–71:31, 2017.
28. J. F. Allen. Maintaining knowledge about temporal intervals. *Commun. ACM*, 26(11):832–843, 1983.
29. J. F. Allen. Towards a general theory of action and time. *Artif. Intell.*, 23(2):123–154, 1984.
30. Amazon. Cloud Services Pricing Amazon Web Services (AWS). `https://aws.amazon.com/pricing/services/`. [Online; accessed 31-March-2019].
31. L. Amini, N. Jain, A. Sehgal, J. Silber, and O. Verscheure. Adaptive control of extreme-scale stream processing systems. In *ICDCS*, pages 71–71, 2006.
32. A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: semantic foundations and query execution. *VLDB J.*, 15(2):121–142, 2006.
33. A. Arasu, M. Cherniack, E. F. Galvez, D. Maier, A. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts. Linear road: A stream data management benchmark. In *VLDB*, pages 480–491, 2004.
34. A. Artikis, A. Margara, M. Ugarte, S. Vansummeren, and M. Weidlich. Complex event recognition languages: Tutorial. In *DEBS*, 2017.
35. A. Artikis, M. J. Sergot, and G. Paliouras. An event calculus for event recognition. *IEEE Trans. Knowl. Data Eng.*, 27(4):895–908, 2015.
36. A. Artikis, A. Skarlatidis, F. Portet, and G. Paliouras. Logic-based event recognition. *Knowledge Eng. Review*, 27(4):469–506, 2012.
37. S. Babu, U. Srivastava, and J. Widom. Exploiting *k*-constraints to reduce memory overhead in continuous queries over data streams. *ACM Trans. Database Syst.*, 29(3):545–580, 2004.
38. M. Balazinska, Y. Kwon, N. Kuchta, and D. Lee. Moirae: History-enhanced monitoring. In *CIDR*, pages 375–386, 2007.
39. C. Balkesen, N. Dindar, M. Wetter, and N. Tatbul. RIP: run-based intra-query parallelism for scalable complex event processing. In *DEBS*, pages 3–14, 2013.
40. H. Barringer, Y. Falcone, K. Havelund, G. Reger, and D. E. Rydeheard. Quantified event automata: Towards expressive and efficient runtime monitors. In *FM*, 2012.
41. A. Benzing, B. Koldehofe, and K. Rothermel. Efficient support for multi-resolution queries in global sensor networks. In *COMSWARE*, pages 11:1–11:12, 2011.
42. J. Boubeta-Puig, G. Daz, H. Maci, V. Valero, and G. Ortiz. Medit4cep-cpn: An approach for complex event processing modeling by prioritized colored petri nets. *Information Systems*, 2017.
43. W. Brendel, A. Fern, and S. Todorovic. Probabilistic event logic for interval-based event recognition. In *CVPR*, pages 3329–3336, 2011.
44. L. Brenna, J. Gehrke, M. Hong, and D. Johansen. Distributed event stream processing with non-deterministic finite automata. In *DEBS*, 2009.
45. A. Brito, C. Fetzer, H. Sturzrehm, and P. Felber. Speculative out-of-order event processing with software transaction memory. In *DEBS*, pages 265–275, 2008.
46. V. Cardellini, V. Grassi, F. Lo Presti, and M. Nardelli. Optimal operator placement for distributed stream processing applications. In *DEBS*, pages 69–80, 2016.
47. V. Cardellini, V. Grassi, F. L. Presti, and M. Nardelli. Optimal operator replication and placement for distributed stream processing systems. *SIGMETRICS Performance Evaluation Review*, 44(4):11–22, 2017.
48. A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Trans. Comput. Syst.*, 19(3):332–383, 2001.
49. I. Cervesato and A. Montanari. A calculus of macro-events: Progress report. In *TIME*, 2000.
50. U. Çetintemel, D. J. Abadi, Y. Ahmad, H. Balakrishnan, M. Balazinska, M. Cherniack, J. Hwang, S. Madden, A. Maskey, A. Rasin, E. Ryvkina, M. Stonebraker, N. Tatbul, Y. Xing, and S. Zdonik. The aurora and borealis stream processing engines. In *Data Stream Management*, Data-Centric Systems and Applications, pages 337–359. Springer, 2016.
51. B. Chandramouli, J. Goldstein, and D. Maier. High-performance dynamic pattern matching over disordered streams. *PVLDB*, 3(1):220–231, 2010.
52. S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, F. Reiss, and M. A. Shah. Telegraphcq: Continuous dataflow processing. In *SIGMOD*, 2003.
53. F. Chen and G. Rosu. Parametric trace slicing and monitoring. In *TACAS*, 2009.
54. S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. J. Peng, et al. Benchmarking streaming computation engines: Storm, flink and spark streaming. In *IPDPSW*, pages 1789–1792, 2016.
55. C. Choppy, O. Bertrand, and P. Carle. Coloured petri nets for chronicle recognition. In *14th Ada-Europe International Conference*, 2009.
56. N. Cipriani, M. Eissele, A. Brodt, M. Grossmann, and B. Mitschang. Nexusds: a flexible and extensible middleware for distributed stream processing. In *IDEAS*, pages 152–161, 2009.
57. G. Cormode, F. Korn, and S. Tirthapura. Time-decaying aggregates in out-of-order streams. In *PODS*, 2008.
58. G. Cugola and A. Margara. TESLA: a formally defined event specification language. In *DEBS*, 2010.
59. G. Cugola and A. Margara. Complex event processing with T-REX. *Journal of Systems and Software*, 85(8):1709–1728, 2012.
60. G. Cugola and A. Margara. Low latency complex event processing on parallel hardware. *J. Parallel Distrib. Comput.*, 72(2):205–218, 2012.
61. G. Cugola and A. Margara. Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.*, 44(3):15:1–15:62, 2012.
62. G. Cugola and A. Margara. Deployment strategies for distributed complex event processing. *Computing*, 95(2):129–156, 2013.
63. G. Cugola, A. Margara, M. Matteucci, and G. Tamburrelli. Introducing uncertainty in complex event processing: model, implementation, and validation. *Computing*, 97(2):103–144, 2015.
64. L. D'Antoni and M. Veanes. Extended symbolic finite automata and transducers. *Formal Methods in System Design*, 47(1):93–119, 2015.
65. L. D'Antoni and M. Veanes. The power of symbolic automata and transducers. In *CAV*, 2017.
66. A. J. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. M. White. Towards expressive publish/subscribe systems. In *EDBT*, 2006.
67. A. J. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, and W. M. White. Cayuga: A general purpose event monitoring system. In *CIDR*, 2007.
68. Y. Diao, N. Immerman, and D. Gyllstrom. Sase+: An agile language for kleene closure over event streams. *UMass Technical Report*, 2007.
69. L. Ding, S. Chen, E. A. Rundensteiner, J. Tatemura, W. Hsiung, and K. S. Candan. Runtime semantic query optimization for event stream processing. In *ICDE*, pages 676–685, 2008.
70. C. Dousson. Extending and unifying chronicle representation with event counters. In *ECAI*, 2002.
71. C. Dousson, P. Gaborit, and M. Ghallab. Situation recognition: Representation and algorithms. In *IJCAI*, 1993.

72. C. Dousson and P. L. Maigat. Chronicle recognition improvement using temporal focusing and hierarchization. In *IJCAI*, 2007.
73. O. Etzion and P. Niblett. *Event Processing in Action*. Manning Publications Company, 2010.
74. R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, and P. R. Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *SIGMOD*, 2013.
75. I. Flouris, V. Manikaki, N. Giatrakos, A. Deligiannakis, M. N. Garofalakis, M. Mock, S. Bothe, I. Skarbovsky, F. Fournier, M. Stajcer, T. Krizan, J. Yom-Tov, and T. Curin. FERARI: A prototype for complex event processing over streaming multi-cloud platforms. In *SIGMOD*, pages 2093–2096, 2016.
76. A. Galton and J. C. Augusto. Two approaches to event definition. In *Database and Expert Systems Applications DEXA*, 2002.
77. M. N. Garofalakis, J. Gehrke, and R. Rastogi, editors. *Data Stream Management - Processing High-Speed Data Streams*. Data-Centric Systems and Applications. Springer, 2016.
78. B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo. Spade: the system s declarative stream processing engine. In *SIGMOD*, pages 1123–1134, 2008.
79. M. Ghallab. On chronicles: Representation, on-line recognition and learning. In *KR*, 1996.
80. C. Ghezzi, D. Mandrioli, and A. Morzenti. TRIO: A logic language for executable specifications of real-time systems. *Journal of Systems and Software*, 12(2):107–123, 1990.
81. N. Giatrakos, A. Artikis, A. Deligiannakis, and M. N. Garofalakis. Complex event recognition in the big data era. *PVLDB*, 10(12):1996–1999, 2017.
82. J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, pages 17–30, 2012.
83. J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. Graphx: Graph processing in a distributed dataflow framework. In *OSDI*, pages 599–613, 2014.
84. Google. Google Cloud Platform Pricing Calculator. `https://cloud.google.com/products/calculator/`. [Online; accessed 31-March-2019].
85. A. Grez, C. Riveros, and M. Ugarte. A formal framework for complex event processing. In *ICDT*, 2019.
86. V. Gulisano, R. Jiménez-Peris, M. Patiño-Martínez, C. Soriente, and P. Valduriez. Streamcloud: An elastic and scalable data streaming system. *IEEE Trans. Parallel Distrib. Syst.*, 23(12):2351–2365, 2012.
87. D. Gyllstrom, E. Wu, H. Chae, Y. Diao, P. Stahlberg, and G. Anderson. SASE: complex event processing over streams (demo). In *CIDR*, 2007.
88. S. Hallé. From complex event processing to simple event processing. *CoRR*, abs/1702.08051, 2017.
89. U. Hedtstück. *Complex event processing: Verarbeitung von Ereignismustern in Datenströmen*. Springer Vieweg, Berlin, 2017.
90. T. Heinze, Z. Jerzak, G. Hackenbroich, and C. Fetzer. Latency-aware elastic scaling for distributed data stream processing systems. In *DEBS*, pages 13–22, 2014.
91. T. Heinze, Y. Ji, Y. Pan, F. J. Grüneberger, Z. Jerzak, and C. Fetzer. Elastic complex event processing under varying query load. In *BD3@VLDB*, volume 1018 of *CEUR Workshop Proceedings*, pages 25–30. CEUR-WS.org, 2013.
92. T. Heinze, Y. Ji, L. Roediger, V. Pappalardo, A. Meister, Z. Jerzak, and C. Fetzer. FUGU: elastic data stream processing with latency constraints. *IEEE Data Eng. Bull.*, 38(4):73–81, 2015.
93. T. Heinze, V. Pappalardo, Z. Jerzak, and C. Fetzer. Auto-scaling techniques for elastic data stream processing. In *DEBS*, pages 318–321, 2014.
94. M. Hirzel. Partition and compose: parallel complex event processing. In *DEBS*, pages 191–200, 2012.
95. J. Hopcroft, R. Motwani, and J. Ullman. *Introduction to automata theory, languages, and computation*. Pearson/Addison Wesley, 2007.
96. Y. Ji, H. Zhou, Z. Jerzak, A. Nica, G. Hackenbroich, and C. Fetzer. Quality-driven processing of sliding window aggregates over out-of-order data streams. In *DEBS*, pages 68–79, 2015.
97. E. Kalyvianaki, W. Wiesemann, Q. H. Vu, D. Kuhn, and P. Pietzuch. Sqpr: Stream query planning with reuse. In *ICDE*, pages 840–851, 2011.
98. M. Kaminski and N. Francez. Finite-memory automata. *Theor. Comput. Sci.*, 134(2):329–363, 1994.
99. J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, and V. Markl. Benchmarking distributed stream data processing systems. In *ICDE*, pages 1507–1518, 2018.
100. N. Katzouris, A. Artikis, and G. Paliouras. Online learning of event definitions. *TPLP*, 16(5-6):817–833, 2016.
101. N. Katzouris, A. Artikis, and G. Paliouras. Parallel online event calculus learning for complex event recognition. *Future Generation Comp. Syst.*, 94:468–478, 2019.
102. H. Kawashima, H. Kitagawa, and X. Li. Complex event processing over uncertain data streams. In *3PGCIC*, pages 521–526, 2010.
103. G. G. Koch, B. Koldehofe, and K. Rothermel. Cordies: Expressive event correlation in distributed systems. In *DEBS*, 2010.
104. I. Kolchinsky, I. Sharfman, and A. Schuster. Lazy evaluation methods for detecting complex events. In *DEBS*, 2015.
105. R. A. Kowalski and M. J. Sergot. A logic-based calculus of events. *New Generation Comput.*, 4(1):67–95, 1986.
106. V. Kumar, B. F. Cooper, and K. Schwan. Distributed stream management using utility-driven self-adaptive middleware. In *ICAC*, pages 3–14, 2005.
107. O. Lee and J. E. Jung. Sequence clustering-based automated rule generation for adaptive complex event processing. *Future Generation Comp. Syst.*, 66:100–109, 2017.
108. C. Lei and E. A. Rundensteiner. Robust distributed query processing for streaming data. *ACM Trans. Database Syst.*, 39(2):17:1–17:45, 2014.
109. C. Li, Y. Gu, G. Yu, and B. Hong. Aggressive complex event processing with confidence over out-of-order streams. *J. Comput. Sci. Technol.*, 26(4):685–696, 2011.
110. G. Li and H. Jacobsen. Composite subscriptions in content-based publish/subscribe systems. In *Middleware*, 2005.
111. Z. Li and T. Ge. History is a mirror to the future: Best-effort approximate complex event matching with insufficient resources. *PVLDB*, 10(4):397–408, 2016.
112. H. Liu and H. Jacobsen. Modeling uncertainties in publish/subscribe systems. In *ICDE*, pages 510–521, 2004.
113. M. Liu, M. Li, D. Golovnya, E. A. Rundensteiner, and K. T. Claypool. Sequence pattern query processing over out-of-order event streams. In *ICDE*, 2009.
114. M. Liu, E. A. Rundensteiner, K. Greenfield, C. Gupta, S. Wang, I. Ari, and A. Mehta. E-cube: multi-dimensional event sequence analysis using hierarchical pattern query sharing. In *SIGMOD*, 2011.
115. K. Lolos, I. Konstantinou, V. Kantere, and N. Koziris. Elastic management of cloud applications using adaptive reinforcement learning. In *Big Data*, 2017.
116. R. Lu, G. Wu, B. Xie, and J. Hu. Stream bench: Towards benchmarking modern distributed stream computing frameworks. In *UCC*, pages 69–78, 2014.
117. D. C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley, 2001.
118. A. Margara, G. Cugola, and G. Tamburrelli. Learning from the past: automated rule generation for complex event processing. In *DEBS*, pages 47–58. ACM, 2014.
119. R. Mayer, B. Koldehofe, and K. Rothermel. Predictable low-latency event detection with parallel complex event processing. *IEEE Internet of Things Journal*, 2(4):274–286, 2015.

120. R. Mayer, C. Mayer, M. A. Tariq, and K. Rothermel. Graphcep: real-time data analytics using parallel complex event and graph processing. In *DEBS*, pages 309–316, 2016.
121. R. Mayer, A. Slo, M. A. Tariq, K. Rothermel, M. Gräber, and U. Ramachandran. SPECTRE: supporting consumption policies in window-based parallel complex event processing. In *Middleware*, pages 161–173, 2017.
122. R. Mayer, M. A. Tariq, and K. Rothermel. Minimizing communication overhead in window-based parallel complex event processing. In *DEBS*, pages 54–65, 2017.
123. Y. Mei and S. Madden. Zstream: a cost-based query processor for adaptively detecting composite events. In *SIGMOD*, 2009.
124. M. Mendes, P. Bizarro, and P. Marques. Towards a standard event processing benchmark. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, pages 307–310. ACM, 2013.
125. Microsoft. Bandwidth Pricing Details. `https://azure.microsoft.com/en-us/pricing/details/bandwidth/`. [Online; accessed 31-March-2019].
126. T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, apr 1989.
127. C. Mutschler and M. Philippsen. Adaptive speculative processing of out-of-order event streams. *ACM Trans. Internet Techn.*, 14(1):4:1–4:24, 2014.
128. A. Paschke. Eca-ruleml: An approach combining ECA rules with temporal interval-based KR event/action logics and transactional update logics. *CoRR*, abs/cs/0610167, 2006.
129. A. Paschke and M. Bichler. Knowledge representation concepts for automated SLA management. *Decision Support Systems*, 46(1):187–205, 2008.
130. K. Patroumpas, E. Alevizos, A. Artikis, M. Vodas, N. Pelekis, and Y. Theodoridis. Online event recognition from moving vessel trajectories. *GeoInformatica*, 21(2):389–427, 2017.
131. P. Pietzuch. How event-based systems took over the world. In *DEBS, 2016*, https://www.ics.uci.edu/d̃ebs2016/ebs-prp-debs16.pdf. Keynote Speech [Online; accessed 31-March-2019].
132. P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. Network-aware operator placement for stream-processing systems. In *ICDE*, 2006.
133. P. R. Pietzuch, B. Shand, and J. Bacon. A framework for event composition in distributed systems. In *Middleware*, 2003.
134. T. C. Przymusinski. On the declarative semantics of deductive databases and logic programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216. Morgan Kaufmann, 1988.
135. Q. Pu, G. Ananthanarayanan, P. Bodik, S. Kandula, A. Akella, P. Bahl, and I. Stoica. Low latency geo-distributed data analytics. In *SIGCOMM*, pages 421–434, New York, NY, USA, 2015.
136. A. Rabkin, M. Arye, S. Sen, V. S. Pai, and M. J. Freedman. Aggregation and degradation in jetstream: Streaming analytics in the wide area. In *NSDI*, pages 275–288, 2014.
137. M. Ray, C. Lei, and E. A. Rundensteiner. Scalable pattern sharing on event streams. In *SIGMOD*, pages 495–510, 2016.
138. C. Ré, J. Letchner, M. Balazinska, and D. Suciu. Event queries on correlated probabilistic streams. In *SIGMOD*, pages 715–728, 2008.
139. T. Repantis, X. Gu, and V. Kalogeraki. Synergy: Sharing-aware component composition for distributed stream processing systems. In *Middleware*, pages 322–341, 2006.
140. N. Rivetti, N. Zacheilas, A. Gal, and V. Kalogeraki. Probabilistic management of late arrival of events. In *DEBS*, 2018.
141. S. Rizou. Concepts and algorithms for efficient distributed processing of data streams. *University of Stuttgart*, 2013.
142. B. Schilling, B. Koldehofe, and K. Rothermel. Efficient and distributed rule placement in heavy constraint-driven event systems. In *HPCC*, pages 355–364, 2011.
143. N. P. Schultz-Møller, M. Migliavacca, and P. R. Pietzuch. Distributed complex event processing with query rewriting. In *DEBS*, 2009.
144. J. Selman, M. R. Amer, A. Fern, and S. Todorovic. PEL-CNF: probabilistic event logic conjunctive normal form for video interpretation. In *ICCVW*, pages 680–687, 2011.
145. A. Skarlatidis, G. Paliouras, A. Artikis, and G. A. Vouros. Probabilistic event calculus for event recognition. *ACM Trans. Comput. Log.*, 16(2):11:1–11:37, 2015.
146. A. Skarlatidis, G. Paliouras, G. A. Vouros, and A. Artikis. Probabilistic event calculus based on markov logic networks. In *RuleML 2011*, pages 155–170, 2011.
147. M. Veanes, P. de Halleux, and N. Tillmann. Rex: Symbolic regular expression explorer. In *ICST*, 2010.
148. A. Vulimiri, C. Curino, P. B. Godfrey, T. Jungblut, J. Padhye, and G. Varghese. Global analytics in the face of bandwidth and regulatory constraints. In *NSDI*, pages 323–336, 2015.
149. J. Wang, S. Song, X. Zhu, and X. Lin. Efficient recovery of missing events. *PVLDB*, 6(10):841–852, 2013.
150. Y. H. Wang, K. Cao, and X. M. Zhang. Complex event processing over distributed probabilistic event streams. *Computers & Mathematics with Applications*, 66(10):1808–1821, 2013.
151. S. Wasserkrug, A. Gal, O. Etzion, and Y. Turchin. Complex event processing over uncertain data. In *DEBS*, volume 332, pages 253–264, 2008.
152. M. Weidlich, H. Ziekow, A. Gal, J. Mendling, and M. Weske. Optimizing event pattern matching using business process models. *IEEE Trans. Knowl. Data Eng.*, 26(11):2759–2773, 2014.
153. G. A. Wilkin, P. Eugster, and K. R. Jayaram. Decentralized fault-tolerant event correlation. *ACM Trans. Internet Technol.*, 14(1):5:1–5:27, aug 2014.
154. J. Wolf, N. Bansal, K. Hildrum, S. Parekh, D. Rajan, R. Wagle, K.-L. Wu, and L. Fleischer. Soda: An optimizing scheduler for large-scale stream-based distributed computer systems. In *Middleware*, pages 306–325, 2008.
155. L. Woods, J. Teubner, and G. Alonso. Complex event detection at wire speed with fpgas. *PVLDB*, 3(1), 2010.
156. WSO2. Creating a Storm Based Distributed Execution Plan. `https://docs.wso2.com/display/CEP410/Creating+a+Storm+Based+Distributed+Execution+Plan`. [Online; accessed 31-March-2019].
157. E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *SIGMOD*, 2006.
158. N. Zacheilas, V. Kalogeraki, N. Zygouras, N. Panagiotou, and D. Gunopulos. Elastic complex event processing exploiting prediction. In *Big Data*, pages 213–222, 2015.
159. N. Zacheilas, N. Zygouras, N. Panagiotou, V. Kalogeraki, and D. Gunopulos. Dynamic load balancing techniques for distributed complex event processing systems. In *DAIS*, 2016.
160. E. Zeitler and T. Risch. Massive scale-out of expensive continuous queries. *PVLDB*, 4(11):1181–1188, 2011.
161. H. Zhang, Y. Diao, and N. Immerman. Recognizing patterns in streams with imprecise timestamps. *PVLDB*, 3(1):244–255, 2010.
162. H. Zhang, Y. Diao, and N. Immerman. On complexity and optimization of expensive queries in complex event processing. In *SIGMOD*, pages 217–228, 2014.
163. D. Zimmer and R. Unland. On the semantics of complex events in active database management systems. In *ICDE*, 1999.