Automated Pattern Extraction in Complex Event Processing Systems

Styliani Kyrama[1,*,†], Anastasios Gounaris[1,†]

[1] *School of Informatics, Aristotle University of Thessaloniki*

Abstract

Complex Event Processing (CEP) is employed in various domains to detect the occurrence of sequential patterns of interest, thus enabling timely reactions and informed decision-making. Although CEP systems can efficiently detect predefined patterns, they exhibit an important limitation; they do not examine detected patterns for systematic continuations or alternative endings that could enrich the information provided to users. Several works attempt to discover new patterns, drawing from the fields of CEP, Sequential Pattern Mining, Machine Learning, and Deep Learning; however, these approaches typically require labeled data, perform offline analysis, or both. To address this limitation, we propose a mechanism for CEP systems that employs a pattern-trie structure to maintain an overview of pattern evolution, together with three update algorithms. We provide an effective and efficient solution for exploring frequent extensions and variations of the original pattern online, during CEP execution. Our evaluation indicates that online pattern exploration can be integrated with CEP execution without significant additional overhead relative to the underlying CEP execution.

Keywords

CEP, automated pattern detection, pattern-trie, frequent pattern evolution

1. Introduction

Modern systems increasingly process large volumes of data to extract valuable knowledge and insights, often originating from sensors or other Internet of Things (IoT) devices, rather than traditional databases [1, 2]. Such processing may include the detection of sequential pattern occurrences, the generation of alerts, and timely updates to users, enabling rapid reaction and informed decision-making [3]. To support these requirements, many systems across diverse domains employ Complex Event Processing (CEP) solutions [4, 5, 6, 7]. Despite the maturity of CEP, most of the existing approaches rely on pattern queries specified by domain experts. For example, in the field of healthcare, a doctor can define situations of interest that they want to monitor for a specific patient. However, defining a descriptive and accurate set of rules or patterns is challenging, even for a domain expert. Subtle relationships or unknown correlations within the data may be overlooked. This dependency limits their applicability in environments where domain experts cannot easily specify all relevant patterns or where event correlations evolve dynamically.

Many optimizations have been proposed in CEP systems, either to improve performance even in demanding scenarios [8, 9, 10, 11] or to enhance accuracy under real-world inconsistencies [12, 13, 14], such as out-of-order or late arrivals. Yet, all these solutions assume that the pattern logic is predefined, which in practice, is very restrictive. Several research efforts have attempted to move beyond static, pre-defined CEP queries by learning or discovering patterns from historical or operational data.

In this work, we propose an online automatic pattern extraction approach for CEP systems, implemented as an add-on on top of the LimeCEP engine[15]. While traditional CEP systems detect only the original pattern explicitly defined by the user, our approach explores and maintains frequent pattern

DOLAP 2026: 28th International Workshop on Design, Optimization, Languages and Analytical Processing of Big Data, co-located with EDBT/ICDT 2026, March 24, 2026, Tampere, Finland

*Corresponding author.

†These authors contributed equally.

kyrastyl@csd.auth.gr (S. Kyrama); gounaria@csd.auth.gr (A. Gounaris)
https://www.linkedin.com/in/styliani-kyrama/ (S. Kyrama); http://datalab.csd.auth.gr/~gounaris/ (A. Gounaris)
0000-0002-5592-8506 (S. Kyrama); 0000-0003-3976-7615 (A. Gounaris)

evolutions during CEP execution, with minimal overhead. We introduce a lightweight pattern-trie architecture together with three complementary counting algorithms that enable practical online pattern suggestion within the CEP engine, while preserving throughput characteristics under demanding workloads. Specifically, we consider two types of evolutions: (i) extended patterns, which append additional event types to the original pattern to capture richer event correlations, and (ii) altered patterns, which replace the final event type of the pattern and act as complementary suggestions that may reveal relevant event structures overlooked by the user. Unlike prior work on sequential or episode mining, our approach performs online exploration of pattern evolutions directly inside the CEP engine, preserving the exact execution semantics of the original pattern query, including window and selection policies. For example, given a user-defined pattern ABC, an extended evolution is ABCD, while an altered evolution is ABE. Our evaluation shows that automated pattern extraction can run alongside CEP execution on large windows and real-world streams, without introducing additional throughput overhead.

Outline. Next, we provide background information and we formalize our problem. The architecture and our novel algorithms are presented in Section 3. We evaluate our proposal in Section 4, and we discuss related work in Section 5, before we conclude in Section 6.

2. Preliminaries

2.1. CEP Basics

CEP systems continuously evaluate pattern queries over incoming event streams. A pattern is typically an ordered sequence of event types $\langle ET_1, \dots, ET_n \rangle$, evaluated under a selection policy (e.g., *skip-till-next-match* (STNM) or *skip-till-any-match* (STAM)), a consumption policy (e.g. consume, reuse), and time constraints w [3, 16, 1]. During runtime, the CEP engine detects matches of the pattern from the events in the incoming stream. Formally, given a stream $S = \langle e_1, e_2, \dots \rangle$, where e_i is an event at timestamp i, a match of pattern $p = (\langle ET_1, \dots, ET_n \rangle, sp, w)$ is a subsequence $\langle e_{i_1}, \dots, e_{i_n} \rangle$ such that $\text{type}(e_{i_j}) = ET_j$, $e_{i_1}.ts < \dots < e_{i_n}.ts$, and $e_{i_n}.ts - e_{i_1}.ts \leq w$ ($e.ts$ denotes the timestamp of e). CEP engines implement these semantics through different execution models (e.g., NFA-, tree-, or graph–based architectures [1]). Our method is implemented on top of LimeCEP [13], but the problem formulation relies on generic CEP semantics. Without loss of generality, in this work we assume that the consume policy is always set to *reuse* [3, 1].

Limitations of current CEP engines. Existing CEP systems operate strictly with *user-defined* pattern queries. This leads to two core limitations: (i) lack of runtime adaptability, i.e., the CEP system cannot integrate new correlations or shifts of incoming data in the active pattern; and (ii) lack of introspective analysis, i.e., the engines do not examine the produced matches for frequent continuations, or alternative end events that may systematically appear in the incoming event stream.

2.2. Motivation Example

Consider a smart-building IoT system monitoring HVAC units, vibration sensors, and power consumption. A domain expert submits to the CEP system the pattern

$$p = (\langle \texttt{TempRise}, \texttt{FanStart}, \texttt{PowerIncrease} \rangle, sp = \text{STNM}, w = 10min)$$

in order to detect potential overheating episodes.

However, during continuous monitoring and evaluation of the stream, it is observed that after a sequence of $\langle \texttt{TempRise}, \texttt{FanStart}, \texttt{PowerIncrease} \rangle$ events, a `VibrationSpike` event often occurs, indicating that the fan is starting to operate under mechanical stress. Additionally, it is observed that after the arrival of the $\langle \texttt{TempRise}, \texttt{FanStart} \rangle$ combination, a `PressureAlert` event regularly occurs besides `PowerIncrease`, suggesting an alternative expression of the same physical condition. A CEP engine will continue to match only the expert-defined pattern p, without revealing the frequent extension

$$p^+ = (\langle \texttt{TempRise}, \texttt{FanStart}, \texttt{PowerIncrease}, \texttt{VibrationSpike} \rangle, sp = \text{STNM}, w = 10min)$$

or the frequent variation

$$p' = (\langle \texttt{TempRise}, \texttt{FanStart}, \texttt{PressureAlert} \rangle, sp = \text{STNM}, w = 10min).$$

Without a mechanism that identifies such frequently recurring extensions or variations of the original pattern, the CEP system provides only a partial view of the actual behaviour observed in the stream. This highlights the need for CEP engines to enhance user-defined patterns with meaningful suggestions observed directly in the incoming stream.

2.3. Problem Definition

In this section, we formalize the problem of systematically exploring frequent extensions and variations of a user-defined CEP pattern query from an incoming event stream. The notation used in this Section follows that of [3, 16, 1].

Given a pattern p defined as a sequence of events $\langle ET_1, ET_2, \ldots, ET_n \rangle$, with selection policy as sp, window w, and a continuous incoming event stream S, the goal is to explore two kinds of pattern evolutions, while maintaining same sp, and w:

- **Extensions** p^+: patterns of the form $p^+ = (\langle ET_1, ET_2, \ldots, ET_n, ET_{n+1} \rangle, sp, w)$, obtained by appending a new event type ET_{n+1} to the end of p event sequence; and
- **Variations** $p^{'}$: patterns of the form $p^{'} = (\langle ET_1, ET_2, \ldots, ET_{n-1}, ET_n' \rangle sp, w)$, obtained by replacing the last event type of p event sequence with a new event type ET_n' that does not already appear in p.

Each possible pattern evolution, either p^+ or $p^{'}$, should satisfy two conditions: (i) consistency with the event-time semantics of the original pattern p, i.e. satisfy the window and temporal constraints; and (ii) its confidence must exceed user-defined threshold. Regarding the second condition, an extension of the pattern $p^+ = \langle ET_1, \ldots, ET_n, ET_{n+1} \rangle$, should satisfy

$$Prob(\langle ET_1, \ldots, ET_{n+1} \rangle) = \frac{\text{freq}(\langle ET_1, \ldots, ET_{n+1} \rangle)}{\sum\limits_{r \in \{\text{p}, \text{p}^+, \text{p}'\}} \text{freq}(r)} \geq \theta_{\text{conf}}$$

where $\text{freq}(\langle \text{ET}_1, \ldots, \text{ET}_{\text{n}+1} \rangle)$ denotes the frequency of event type sequence $\langle ET_1, \ldots, ET_{n+1} \rangle$, i.e., the number of valid occurrences in the stream S, and $\sum_{r \in \{p, p^+, p'\}} \text{freq}(r)$ indicates the number of matches including all possible extensions and variations. Corresponding confidence constraints apply also to variations $p^{'}$. The exploration of such possible pattern evolutions must be performed incrementally, in a streaming manner.

3. Automated Pattern Extraction in LimeCEP

The pattern exploration as defined in Section 2.3 is implemented on top of LimeCEP engine [13], as an add-on component, without modifying the core CEP semantics. The source code of our implementation is publicly available[1]. Its integration within the overall system architecture is shown in Figure 1. In a nutshell, LimeCEP is a lightweight CEP engine, designed for real-time pattern detection over event streams with out-of-order arrivals, and high input rates. Instead of computing and maintaining partial matches as NFA-based CEP solutions, LimeCEP indexes incoming events in a timestamp-ordered in-memory structure, and evaluates patterns retrospectively. This approach does not suffer from explosion of the space required when partial matches increase, and enables efficient pattern detection under disorder, while partial reprocessing of events, combined with speculation, allows LimeCEP to correct previously emitted matches when late events arrive.

[1]https://github.com/KyraStyl/LimeCEP/tree/semi-automated

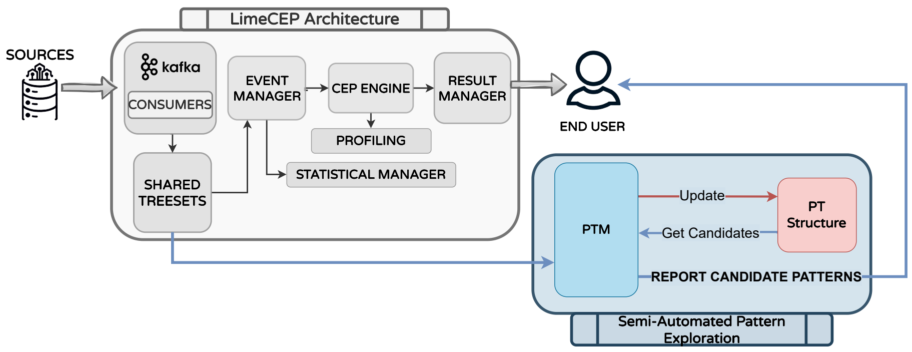


Figure 1: High-Level Architecture

3.1. Architecture

To support the continuous exploration and generation of new pattern candidates, either extensions or variations of the user-defined pattern, we introduce two modules: (i) a *Pattern Trie* data structure, which compactly maintains all possible explored patterns (along with their frequency and statistics), as paths in a tree-like structure, similar to prefix trees, and (ii) a *Pattern Trie Manager*, which is responsible for monitoring the incoming stream, and updating the *pattern trie* structure. Each root-to-leaf path in the trie corresponds to a candidate pattern, and the associated counts are computed under the same window and selection policy as the original query.

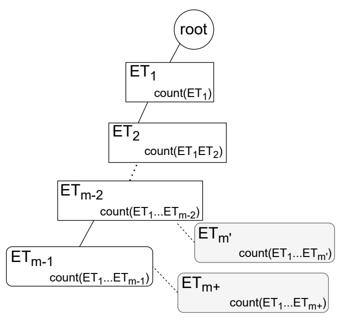
Pattern Trie Manager. The Pattern Trie Manager (PTM) is responsible for exploring candidate pattern extensions and variations, as formalized in Section 2.3. Conceptually, the PTM continuously monitors the stream S for events of *unknown* types T, i.e., event types that do not appear in the user-defined pattern p. Whenever such an event is received, the PTM evaluates the corresponding candidate patterns. More specifically, given a monitored pattern $p = (\langle ET_1, \ldots, ET_k \rangle, sp, w)$ and a new event type T, the PTM considers two candidate evolutions:

- the extended pattern $p^+ = (\langle\langle ET_1, \ldots, ET_k\rangle \parallel T\rangle, sp, w)$;
- the varied pattern $p^{'} = (\langle\langle ET_1, \ldots, ET_{k-1}\rangle \parallel T\rangle, sp, w)$, where $\langle ET_1, \ldots, ET_k\rangle$, i.e. the event-type sequence of pattern p without its last symbol, is the $\text{prefix}(p)$.

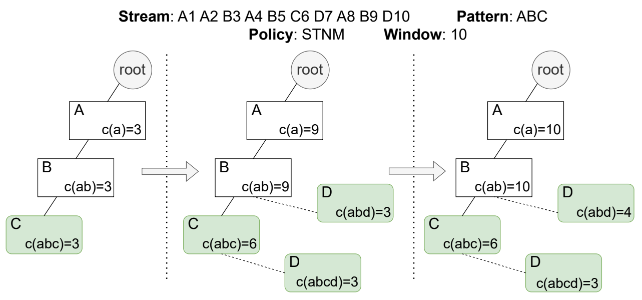
For each such candidate pattern, the PTM estimates its frequency under the constraints imposed by w and selection policy sp of the original user-defined pattern p. In this work, as already stated, we do not consider event consumption policies, and therefore, we adopt the *reuse* policy (as defined in [3, 1]), where events may participate in multiple matches without restrictions. Once the frequency of a candidate evolution has been estimated, the PTM records this information in the *pattern trie*, by updating its corresponding path. The update of pattern statistics is carried out using one of three counting algorithms: (i) Brute Force, (ii) Reverse Count, and (iii) Incremental Count, which differ in efficiency but yield identical results. These algorithms are introduced in Section 3.2. PTM uses the trie to maintain a consistent view of the candidate space and to determine whether an extension or variation satisfies the user-configured confidence threshold. Whenever a candidate pattern meets this requirement, PTM suggests it to the user as a possible evolution to the original query.

Pattern Trie Representation. The *pattern trie* is a prefix-tree structure that compactly stores all patterns explored by the PTM. Each root-to-node path encodes a prefix $\langle ET_1, \ldots, ET_k\rangle$ of a candidate pattern, while each root-to-leaf path, whose final node is marked as ending, corresponds to a complete candidate pattern. Using a shared-prefix hierarchy, it is ensured that common subsequences among the candidate patterns are stored only once in the *pattern trie* structure. This enables efficient accumulation of statistics across related patterns. Figure 2a illustrates the general structure of the *pattern trie*.

Each node stores one event type, and its prefix is given implicitly by the path from the root to this node. In order to support pattern maintenance and exploration, each node also stores:



(a) Pattern Trie representation



(b) Pattern Trie example

Figure 2: Pattern Trie structure

- a *count* representing the number of valid matches in which this root-to-node prefix (sub-pattern) participates, based on window w and selection policy sp constraints, denoted as $count(ET_1 \dots ET_i)$ for each node i with event type ET_i;
- an *isEndOfPattern* flag, indicating whether this node marks the end of a complete pattern (represented as rounded boxes in Figure 2a);
- a set of *children* nodes, mapping event types to successor nodes, representing possible extensions for this root-to-node prefix; and
- a reference to this node's *parent*, for efficient navigation along the prefix, and computation of the $Prob$, as defined in Section 2.3, for this root-to-node prefix or complete pattern.

According to the formal definitions, the *frequency* of an event-type sequence is the number of valid occurrences in the stream, whereas the *count* is the number of valid matches in which the sequence participates. Although these two are closely related, they are not identical. The *pattern trie* stores only the *count* for each prefix; however, the corresponding *frequency* of a sequence n can be derived using the following relation:

$$\text{freq}(n) = \begin{cases} \text{count}(n) - \displaystyle\sum_{c \in \text{children}(n)} \text{count}(c), & \text{if } n.\text{isEndOfPattern == true}, \\ \text{count}(n), & \text{otherwise.} \end{cases}$$

The *pattern trie* is updated whenever the PTM identifies an occurrence of a candidate pattern p_x in the incoming event stream S. The *counts* stored along the corresponding path p_x are incremented, ensuring that prefix statistics are maintained across all explored patterns.

Figure 2b illustrates how the *pattern trie* evolves as events are processed and matches are detected in the stream. Consider the event stream $S = \langle A_1, B_2, B_3, A_4, B_5, C_6, D_7, A_8, B_9, D_{10} \rangle$, and the pattern p defined by the user is the following $p = (\langle A, B, C \rangle, sp = STNM, w = 10)$. The tree is initialized with the root-to-leaf path ABC, as shown in the left part of Figure 2b. When C_6 arrives, the PTM detects three valid matches, namely $m_1 = \langle A_1, B_3, C_6 \rangle$, $m_2 = \langle A_2, B_3, C_6 \rangle$, and $m_3 = \langle A_4, B_5, C_6 \rangle$ under the sp and w constraints imposed by p, and updates the count of the ABC path of the *pattern trie* to 3.

The arrival of D_7 (middle figure) introduces an event type D not contained in p, and therefore triggers the exploration of the candidate patterns $p^+ = (\langle A, B, C, D \rangle, sp = STNM, w = 10)$ and $p^{'} = (\langle A, B, D \rangle, sp = STNM, w = 10)$. Correspondingly, a new ending child node D extends the ABC prefix path, and another ending child node D is added to the AB prefix path. During this exploration, the PTM detects three matches for p^+ path, $m_4^+ = \langle A_1, B_3, C_6, D_7 \rangle$, $m_5^+ = \langle A_2, B_3, C_6, D_7 \rangle$, and $m_6^+ = \langle A_4, B_5, C_6, D_7 \rangle$ and another three matches for path of $p^{'}$, $m_7^{'} = \langle A_1, B_3, D_7 \rangle$, $m_8^{'} = \langle A_2, B_3, D_7 \rangle$, and $m_9^{'} = \langle A_4, B_5, D_7 \rangle$, leading to updated counts of 6 for $ABCD$ and 3 for ABD. The result *pattern trie* is depicted in the middle part of the figure.

The next event triggering this process for the PTM is event D_{10}. Under the STNM policy, only one new match is detected for the $p^{'}$ candidate pattern, namely match $m_{10}^{'} = \langle A_8 B_9 D_{10} \rangle$. PTM updates the corresponding ABD path, by incrementing the *count* by one, producing the final form of the *pattern trie* as depicted in the right part of the figure. In each update step, PTM checks whether the candidate pattern satisfies the confidence threshold. Considering a θ_{conf} equal to 40%, in the last step, i.e. on the arrival of D_{10}, where the candidate pattern ABD as a variation path is updated with $count(ABD) = 4$, PTM computes the $Prob(\langle A, B, D\rangle)$ as $Prob(\langle A, B, D\rangle) = freq(\langle A, B, D\rangle)/\sum_{r\in\text{root.children}} freq(r) = count(ABD)/\#matches = {}^4\!/_{10} = 0.4 \geq \theta_{conf}$, meaning that among all valid matches detected the 40% is a sequence of $\langle A, B, D\rangle$ (within the window and policy constraint as imposed by the original pattern p). Therefore, the candidate pattern $\langle A, B, D\rangle$ is suggested to the user as an interesting pattern that could alter the original one. In contrast, the extension $\langle A, B, C, D\rangle$ is not yet suggested, since its confidence remains below the required threshold, i.e. $Prob(\langle A, B, C, D\rangle) = freq(\langle A, B, C, D\rangle)/\#matches = {}^3\!/_{10} = 0.3 < \theta_{conf}$.

3.2. Update Algorithms

To ensure that the *pattern trie* remains continuously updated and consistent with the evolving event stream, we developed three algorithms of increasing efficiency for maintaining the *counts* and computing global probabilities of candidate patterns. We present (i) a baseline *Brute-Force* method, that enumerates matches using a full CEP evaluation, and two optimized variants exploiting LimeCEP's internal structures: (ii) *Reverse Count*, that reconstructs counts by traversing backwards through the window, and (iii) *Incremental Count*, which reuses prefix counts that are stored in the STS structure of LimeCEP system, with minimal window correction. Each version produces identical logical results but with increasingly improved runtime efficiency.

3.2.1. Brute-Force Algorithm

The brute-force method provides the most direct, but also the most expensive, way to update the *pattern trie*. For every newly arrived event e whose type is not presence in the pattern types, candidate patterns $p_x = \{p^+, p^{'}\}$ are created and the algorithm runs an *onDemand* CEP process, to detect all the corresponding matches for pattern p_x within the window w under selection policy sp. Each enumerated match contributes one increment to the corresponding path p_x in the *pattern trie*.

Algorithm 1: **Brute-Force Trie Update**

```
Require: Pattern $p$, event stream $S$
 1: prefix ← extractPrefix($p$)                      // prefix of the pattern
 2: for each event $e \in S$ do
 3:     if ($T = type(e)) \notin p$ then
 4:         $p^+$ ← prefix ‖ $T$                     // pattern extension $p^+$
 5:         $p^{'} \leftarrow p \parallel T$          // pattern variation $p^{'}$
 6:         find_matches($p^+$)
 7:         update_trie_counts($p^+$)
 8:         find_matches($p^{'}$)
 9:         update_trie_counts($p^{'}$)
10:     else
11:         find_matches_and_update_trie_counts($p$)  // original pattern $p$
```

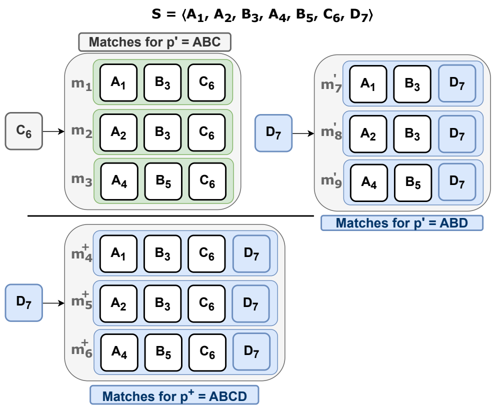


Figure 3: Example BF

This approach guarantees correctness and serves as a reference baseline. However, since every update triggers a complete match re-evaluation, its cost is proportional to the full window size and becomes prohibitive under high event rates or large windows. An illustrative example regarding the matches is provided in Figure 3 continuing the example of Figure 2b.

Complexity. The Brute-Force method recomputes all matches of each candidate pattern p_x by processing the entire active window for this specific event that triggered the process. Since detection of matches is performed for both the extension p^+ and the variation p', each update incurs the full cost of a CEP match evaluation. This leads to poor scalability as the window grows, and makes the approach suitable mainly as a correctness baseline rather than for continuous high-throughput processing.

Portability. The Brute-Force algorithm is fully portable across CEP systems (e.g. SASE, openCEP, FlinkCEP), as it requires only CEP functionality of match detection. No assumptions are made about the internal representation of CEP engine (e.g. automata-based, tree-based etc.). For example, in SASE, a second NFA could be instantiated whenever a new event type arrived in the system.

3.2.2. Optimization: Reverse Count

To improve the efficiency on pattern exploration, we present an optimization to the Brute-Force algorithm, named *Reverse Count* (RC). The RC method avoids full CEP match computation and detection, by counting the number of matches based on predecessor events. Given a candidate pattern $p = (\langle ET_1, \ldots, ET_m\rangle, sp, w)$, where ET_m is the type of the newly arrived event e, the algorithm begins from this event and going backwards, to find the relevant preceding events of previous types in pattern. For each relevant event retrieved, it computes recursively the count, until it reaches a the starting type, i.e. $ET_{m-1} \rightarrow ET_{m-2} \rightarrow \cdots \rightarrow ET_1$.

Algorithm 2: **Reverse Count**

```
Require: Window $W_t$, pattern sequence $p = \langle E_1, \ldots, E_k\rangle$, prefix $pref$, event stream $S$,
    selection policy $sp \in$ {STNM, STAM}
 1: for each event $e \in S$ do
 2:     $T \leftarrow type(e)$
 3:     if $T \notin p$ then
 4:         $p^+ \leftarrow p \parallel T$
 5:         $p' \leftarrow pref \parallel T$
 6:         for each pattern $P_x$ in $\{p^+, p'\}$ do
 7:             $p_{types} = [ET_1, ET_2, \ldots, ET_m]$ where $ET_m = T$
 8:             /* Backwards retrieval stage */
 9:             Initialize $CountSet[ET_m] \leftarrow \{e\}$          // set (STNM) or multiset (STAM)
10:             for $i = m - 1$ down to 1 do
11:                 $CountSet[ET_i] \leftarrow \emptyset$
12:                 for each event $ev_{i+1} \in CountSet[ET_{i+1}]$ do
13:                     Retrieve all events of type $ET_i$ in $W$ that precede $ev_{i+1}$
14:                     Add them to $CountSet[ET_i]$      // STAM: with duplicates; STNM: w/o
    duplicates
15:             /* Forward counting stage */
16:             Count($ET_2$) $\leftarrow |CountSet[ET_1]|$
17:             for $i = 3$ to $m$ do
18:                 for each event $ev_i \in CountSet[ET_i]$ do
19:                     $count(ev_i) \leftarrow \sum_{ev_{i-1} \prec ev_i} count(ev_{i-1})$
20:             TotalCount($p_x$) $\leftarrow \sum_{ev_m \in CountSet[ET_m]} count(ev_m)$
21:             Update trie path for $p_x$ with TotalCount($p_x$)
```

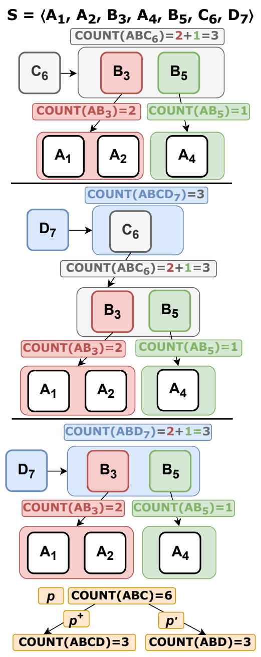


Figure 4: Example RC

Consider the stream $S = \langle A_1, A_2, B_3, A_4, B_5, C_6, D_7\rangle$, and the user-defined pattern is $p = (\langle A, B, C\rangle, sp = STNM, w = 10)$. When event D_7 arrives, RC starts the exploration of $p^+ = (\langle A, B, C, D\rangle, sp = STNM, w = 10)$, and therefore we have $ET_4 = D$. The first step is to identify and retrieve all preceding events of the previous type (i.e. $ET_3 = C$) that satisfy the window and policy constraints, namely C_6. For each of these predecessors, it computes recursively the count and sum it up to the total count. The count of C_6 will be computed based on the preceding events within w under sp, namely B_3 and B_5. Then, respectively, the count of B_3 will be computed based on the set of preceding events of type A, i.e., A_1 and A_2. Since A is the starting type, the recursion will stop, and will return $count = 1$ for each event pEv in the set of As. Then, RC starts the forward counting stage, and computes the corresponding counts. This step-by-step process is illustrated in Figure 4.

This procedure is followed independently for both extension p^+ and variation $p^{'}$ of the pattern.

Complexity. Unlike the Brute-Force method, which performs a full match detection for every update, RC algorithm limits the processing only to the predecessor events relevant to the triggering event. As a result, RC achieves substantially lower update latency, while still producing accurate match counts even without reconstructing the actual matches.

Portability. The RC algorithm remains portable across different CEP engines, since it requires simple functionality of retrieving the predecessors of a specific type within a window and under a selection policy. This can be implemented on top of any system like SASE or openCEP, without assuming any particular internal representation structure. However, its efficiency depends on the data structures provided by the system for the event storage. Engines that maintain their incoming events in time ordered indices per type, like LimeCEP with TreeSet structures (STS), can provide efficiently an event's predecessors, while for engines that keep events in (not-indexed) lists, additional mechanisms may be required to enable efficient retrieval.

3.2.3. Optimization alternative: Incremental Count

Although the RC algorithm has improved performance compared to the BF algorithm, by avoiding full match detection, it still recomputes the count for predecessor events repeatedly, i.e., every time an event appears in the predecessor set, RC recomputes its contribution by recursively traversing its previous events. This results in significant overhead, since under the $reuse$ consumption policy adopted in this work, each event can participate in multiple matches.

Algorithm 3: **Incremental Count**

```
Require: Window $W_t$, pattern $P = [E_1, \ldots, E_k]$, prefix $pref$, event stream $S$, selection policy $S_p \in \{\text{STNM}, \text{STAM}\}$
 1: for each event $e \in S$ do
 2:     $T \leftarrow type(e)$
 3:     if $T \notin P$ then
 4:         $p^+ \leftarrow p \| T$
 5:         $p^{'} \leftarrow pref \| T$
 6:         for each pattern $p_x$ in $\{p^+, p^{'}\}$ do
 7:             Let $p_x = [ET_1, ET_2, \ldots, ET_m]$ where $ET_m = T$
 8:             $predType \leftarrow ET_{m-1}$
 9:             /* Retrieve relevant events */
10:             $PredSet \leftarrow$ all events of type $predType$ in $w$ preceding $e$
11:             /* Compute Window-correct counts */
12:             for each event $ev \in PredSet$ do
13:                 $stored \leftarrow ev.count$
                                     // stored in STS, computed at insertion under $sp, w$
14:                 $\Delta \leftarrow e.ts - ev.ts$                  // time difference
15:                 $max\_valid \leftarrow max(0, w - \Delta)$
16:                 $corrected(ev) \leftarrow min(stored, max\_valid)$
17:             /* Compute count for $e$ */
18:             $count(e) \leftarrow \sum_{ev \in PredSet} corrected(ev)$
19:             STS.add($T$, $e$, $count(e)$)
20:             Update trie path for $p_x$ with $count(e)$
```

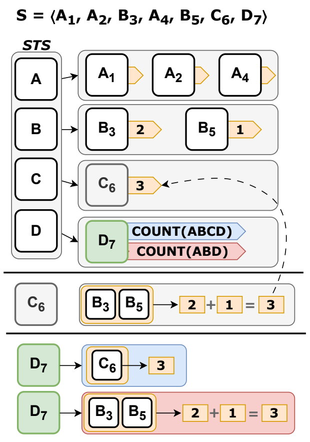


Figure 5: Example IC

To eliminate this, we introduce *Incremental Count* (IC) algorithm, which leverages the counts already stored in the internal structure of the CEP system, along with each event. More specifically, we modified the LimeCEP's internal engine to compute this score once, on event arrival, for the user-defined pattern and under the constraints imposed by it. The pre-computed counts are stored in the internal structure of LimeCEP, namely STS, alongside with the events, so that they are easily accessible; therefore, whenever a later computation requires the contribution of a specific event, IC directly retrieves this stored count from the STS. In cases where a predecessor event ev is close to the window boundary, a part of its stored count may not be valid for the currently processed event. Instead of recomputing the count for ev from scratch, the IC algorithm applies a lightweight adaptation to the count that excludes expired contributing events. Intuitively, IC checks the time distance of ev from the end of the current window defined by the triggering event end as $\Delta = end.ts - ev.ts$, and then subtracts this difference from the window size $max\_valid = max(0, w - \Delta)$. Then, the window-correct count for this event ev is computed as $\text{corrected}(ev) = \min\big(\text{count}(ev),\ max\_valid\big)$, therefore reducing its stored count by the amount that falls outside the valid time range, allowing IC to maintain correct match counts.

In the previous example, when B_3 arrives, the system retrieves its predecessors $\langle A_1, A_2\rangle$), computes its count as 2, and stores it. The same applies to B_5, whose count is 1. When C_6 arrives, IC identifies its predecessors $\langle B_3$ and $B_5\rangle$, but instead of starting a recursive process similar to RC algorithm, it directly fetches their stored counts and sums them, immediately computing the corresponding $count(ABC)$ for event C_6, that is $count(C_6) = 2 + 1 = 3$. This value is then stored in the STS. When D_7 arrives, IC again retrieves the relevant predecessors (only C_6 event for the extension pattern $ABCD$, and events $\langle B_3, B_5\rangle$ for the variation pattern ABD) and computes their total by adding the previously stored counts. The resulting counts are then inserted into the *pattern trie*. The step-by-step execution is illustrated in Figure 5. Regarding the *window-correct counts*, if an event D_{12} arrives, the IC will explore the candidate pattern ABD and retrieve the relevant predecessors of D_{12} for type B including B_3. Even though B_3 is a relevant preceding event, its count contains invalid prefix contributing (i.e. match $\langle A_1, B_3\rangle$). To fix this, IC follows the process described above, computing the the invalid contributions, and correcting the count for B_3 using the formula $corrected(B_3) \leftarrow min(B_3.count, w - (D_{12}.ts - B_3.ts)) = min(2, 10 - (12 - 3)) = min(2, 1) = 1$. Therefore, the window-correct count for B_3 when the end event is D_{12} is 1 instead of 2.

Complexity. The IC algorithm avoids the redundant recomputations performed by the RC algorithm and reduces the update cost to the cost of retrieving a small number of stored counts from the STS. Since predecessor sets are substantially smaller than the full window, IC provides the best scalability among the three update methods proposed.

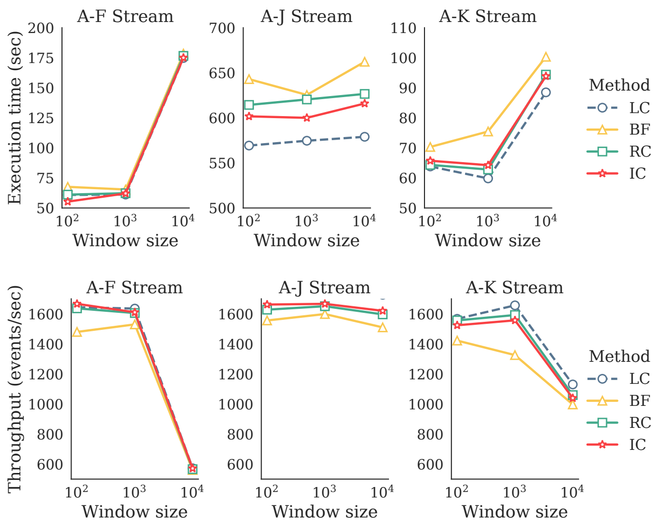
Portability. The IC algorithm can, in practice, be implemented on top of any CEP engine with some modifications since it conceptually requires only: (i) retrieval of predecessor events under a window and selection policy, and (ii) the capability to maintain per-event counts. However, achieving the same efficiency as in LimeCEP is not guaranteed. IC relies heavily on LimeCEP's TreeSet-based event storage structure, which provides logarithmic-time $O(\log n)$ access to predecessor events. Systems that do not maintain a similar indexed time-ordered structure, such as SASE, would require additional and more sophisticated mechanisms to support IC efficiently.

4. Experimental Evaluation

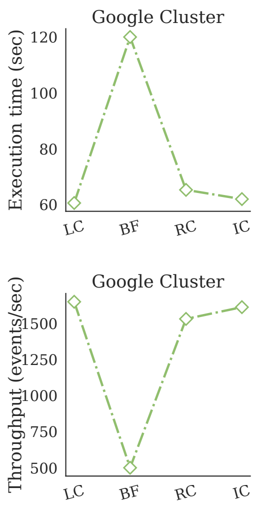
Setup, Competitors, Queries and Datasets. All experiments were conducted on LimeCEP extended with the proposed PTM and *pattern trie*, running on a single machine equipped with 32,GB RAM and an 8-core 3.0GHz CPU. Each experiment was repeated five times, and we report median results. We evaluate the runtime overhead and mining behavior introduced by automated pattern exploration under varying event streams, window sizes, and configurations. We compare four configurations: (i) **LC**, LimeCEP without pattern exploration; (ii) **BF**, the brute-force approach; (iii) **RC**, the reverse-count approach; and (iv) **IC**, the incremental-count approach. Their implementation details are described in Section 3.2. The evaluation uses three synthetic event streams: **A–F** with 100,000 events over 6 event types (A–F), **A–K** with 100,000 events over 11 types (A–K), and **A–J** with 1,000,000 events over 10 types (A–J). We also use a real-world stream derived from Google Cluster traces [17], containing 9 event types corresponding to task life-cycle state transitions. For the synthetic streams, we evaluate the pattern query $P_1 = (\langle A, B, C\rangle, sp = \text{STNM}, w)$, with window sizes $w \in$ 0.1K, 1K, 10K minutes. For the real-world dataset, we use $P_2 = (\langle \text{Submit, Enable, Schedule}\rangle, sp = \text{STNM}, w = 100)$.

Execution Time. Figure 6 depicts the execution time and throughput under different configurations and window sizes for the synthetic data streams (Figure 6a) and the Google Cluster traces (Figure 6b).

In synthetic streams, the brute-force approach exhibits a substantially rapid increase in execution time as the window size increases, which leads to performance degradation. This is more pronounced in the second stream. Although execution time increases for all approaches as the window size increases, the



(a) Synthetic event streams



(b) Real-world event stream

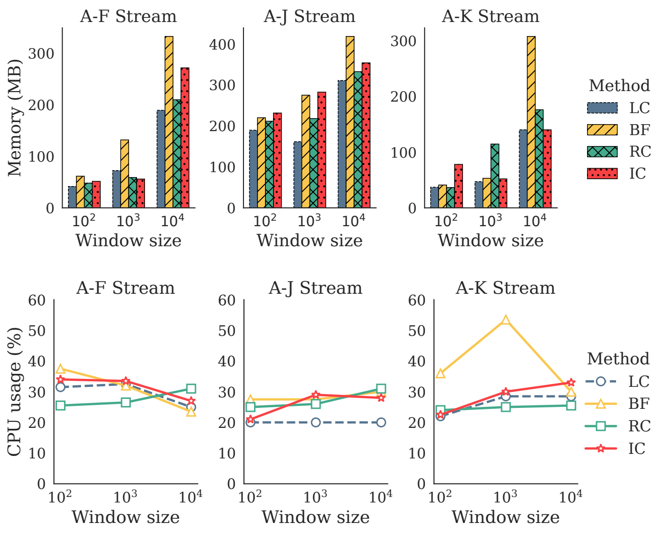
Figure 6: Execution time and throughput

performance of BF is significantly worse than optimized alternatives. This can also be verified from the throughput figures, where BF has the lowest throughput. This behavior is expected, as BF recomputes all valid matches from scratch for each explored candidate pattern, causing its cost to grow incrementally with increasing window sizes, especially when the number of matches increases. Contrary, RC and IC, by exploiting predecessor counts, they maintain substantially more stable execution times than BF, while following the same throughput trend as LC across all window sizes. The LC approach represents the lower bound in runtime performance, executing the original CEP query without mining overhead, while exhibiting the same window-dependent throughput degradation inherent to LimeCEP. RC and IC exhibit comparable performance on all synthetic datasets. This result is expected, since RC already operates efficiently by traversing preceding events. IC approach is not designed to universally outperform RC, rather to eliminate repeated recomputation by reusing prefix counts maintained within the CEP engine.

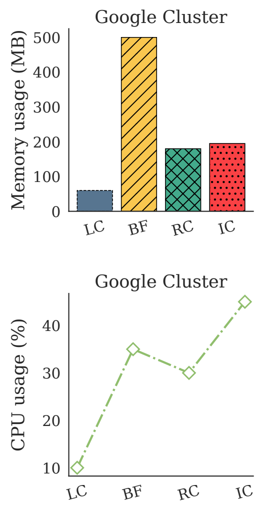
On the real-world dataset (Figure 6b), BF fails to execute the detection and is aborted after 5 hours, whereas, both RC and IC complete execution within approximately 1 minute and maintain comparable throughput to LC, demonstrating the practicality of automated pattern exploration under realistic event streams. However, their relative performance differs with IC exhibiting lower execution time (61 vs. 65.5 s), achieving 7% higher throughput (1639 vs. 1526 events/second).

Memory & CPU Regarding the memory consumption, both figures for synthetic (Figure 7a) and real-world streams (Figure 7b) highlight the overhead of full match detection that BF performs in order to explore the candidate patterns, against the more efficient count-based RC and IC solutions. BF consistently exhibits the highest memory usage across all experiments, with the size of memory consumption growing rapidly as window size increases. In contrast, RC and IC algorithms maintain a compact internal state, therefore their memory footprint increases more gradually with the window size, and remains consistently lower than that of BF. In some cases, however, IC exhibits higher memory overhead than RC algorithm. This difference stems from the additional metadata maintained by IC in memory, namely pre-stored prefix counts for each event. The benefit of storing these counts depends on the degree of reuse of events across multiple matches. The higher the number of matches an event participates in, the more times the same count would need to be recomputed, and consequently, the greater the benefit of having such counts pre-stored. This trade-off suggests that the IC algorithm is particularly beneficial in settings with high match overlap, such as under the STAM selection policy (no experiments are presented due to space limitations).

CPU utilization exhibits a complementary trend, with BF having higher and more variable CPU usage, due to repeated full match detection across multiple candidate pattern evolutions. In contrast, RC and IC



(a) Synthetic event streams



(b) Real-world event stream

Figure 7: Memory and CPU overhead

demonstrate more stable performance across window sizes, with IC incurring slightly higher CPU usage than RC, due to additional processing required to maintain and update the incremental counts. This overhead, however, is not translated into increased execution time or additional throughput degradation. Moreover, as discussed earlier, IC is expected to be more advantageous in more demanding settings, e.g. under STAM policy, where the cost of repeatedly computing counts outweighs the overhead of maintaining them.

Mining Process. To further analyze the behavior of the mining process itself, we evaluate the cost of pattern exploration under a more demanding configuration, using the STAM selection policy. We consider the pattern $P_1 = (\langle A, B, C \rangle, sp = \text{STAM}, w = 100)$ over the **A-K** synthetic stream. Under this configuration, BF detects approximately 9.9×10^7 matches, while LC, RC, and IC detect 4.9×10^5 matches corresponding to the original pattern execution. RC and IC further identify without explicitly detecting that there are also 8.6×10^6 pattern variations and 9×10^7 pattern extensions. The brute-force approach requires approx. 2K seconds to complete and consumes approximately 500 MB of memory. In contrast, both RC and IC complete execution within approximately one minute, with RC requiring 74.6 seconds and IC 71 seconds, respectively. These results highlight a substantial speedup, about 27× for IC over BF and 26× for RC over BF, confirming that complete match detection is infeasible under high-overlap workloads. To highlight the differences between RC and IC under higher overlap, we further executed the real-world query P_2 under $sp = \text{STAM}$ with a larger window $w = 1000$. RC requires 193 minutes and achieves 8.64 events/second, whereas IC requires 121 minutes and achieves 13.74 events/second, corresponding to a 37% reduction in execution time and a 59% increase in throughput. IC is not intended to universally outperform RC; rather, it eliminates repeated backward traversals by reusing prefix counts maintained within the CEP engine.

5. Related Work

Existing CEP systems, such as SASE[18], Cayuga[19], Esper[2], and FlinkCEP[3], have established the main execution paradigms for pattern detection over event streams, using mainly NFAs, with selection and consumption policies on match construction. Other works, such as ZStream[20], CET[21], CORE-CEP[22] and T-REX[23] aim at improving the performance of CEP systems through structural sharing

[2]https://www.espertech.com/esper/
[3]https://nightlies.apache.org/flink/flink-docs-master/docs/libs/cep/

and state reduction, by using tree- and hash-based structures for internal representation. Lazy evaluation has also been explored in OpenCEP[10], to reduce computations and intermediate results, therefore reducing the detection latency of a match. Formal semantic extensions, such as Symbolic Register Transducers[24], provide declarative and compositional pattern semantics, enhancing the expressiveness of CEP engines. Handling efficiently kleene operators has motivated several optimizations, such as [9, 25, 11, 26, 27], for dealing with the exponential growth of internal state and intermediate results. Robust execution under real-world inconsistencies and possible disorders in the incoming stream has been explored in [28, 15, 29, 14], with probabilistic windowing, speculative processing and replay-based processing are exploited in combination, to achieve low-latency and correct detection of matches. More recent works, such as [30, 31, 32], aim to enhance the throughput of CEP systems, by exploiting parallelization techniques, load balancing, and query decomposition. Each of these works optimize a different aspect on the evaluation process of a CEP system, for a specific user-defined pattern query, yet, none of them provides a mechanism for online pattern extraction during the CEP execution process.

Sequential Pattern Mining (SPM) and Episode Mining (EM) aim to extract recurring temporal structures from event sequences. Surveys cover methods for serialized, parallel, and generalized episodes, often with gap, window, or other temporal constraints[33]. Most of these methods, such as [34], run offline, expect event ordering, and do not account for CEP-specific semantics such as selection and consumption policies, maximal matches, or robustness to disorder.

Prior works attempt to bridge this gap by learning CEP rules from historical traces. For example, iCEP[35] learns rule candidates from historical traces using constraint-based inference, DISCES[36] systematically explores the space of possible query designs through labeled historical sub-streams. CBDeclare [37], extends DECLARE with branched constraints capturing AND/OR/XOR relationships and while providing also matching discovery algorithms. A very recent work [38], moves closer to our setting by discovering CEP queries from operational data, using evolutionary computation. Although the approach detects candidate patterns using FlinkCEP, this is performed outside the CEP engine, requires a sampled historical dataset and expert-labeled sequences. In contrast, our work performs online discovery of candidate patterns within the CEP engine, exploring frequent pattern evolutions, consistent with original pattern's CEP semantics, maintaining the throughput characteristics of the underlying CEP engine.

Finally, there are methods that learn rule patterns from historical traces using classical ML models [39, 40], while more recent frameworks employ deep networks to extract CEP rules for IoT streams [41]. Reinforcement-learning techniques [42] adjust thresholds of existing CEP rules at runtime using deep Q-networks on top of FlinkCEP. In parallel, deep models such as [43] support event forecasting and runtime adaptation by coupling neural predictors with CEP engines [44, 45]. While ML and DL methods enhance CEP through forecasting and parameter adaptation, they do not explore new event-type sequences nor evolve the structure of existing patterns, as our proposal does.

6. Conclusions and Future Work

In this work, we presented an add-on mechanism for CEP systems that enables the online exploration of frequent variations and/or continuations of a user-defined pattern during query execution. Our approach employs a lightweight pattern-trie structure and we have also introduced three update algorithms: a brute-force method based on classical CEP detection and two optimized alternatives that exploit predecessor event counts and are an order of magnitude faster. Our design allows pattern exploration to be performed incrementally within the CEP engine. Our experimental evaluation on synthetic and real-world streams indicates that online exploration of pattern evolutions can be integrated into CEP execution without significant additional impact on execution time or throughput, even under large windows and high-throughput workloads. Future work includes extending the current framework to also consider rare pattern evolutions and to support pattern transformations beyond extensions and variations of the last event type.

Acknowledgments

The authors cordially thank Christos Balaktsis for valuable discussions and brainstorming during the development of this work.

Declaration on Generative AI

During the preparation of this work, the authors used OpenAI-GPT-5: Improve writing style and Grammar and spelling check. After using these, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] N. Giatrakos, E. Alevizos, A. Artikis, A. Deligiannakis, M. Garofalakis, Complex event recognition in the big data era: a survey, The VLDB Journal 29 (2020) 313–352.

[2] M. Fragkoulis, P. Carbone, V. Kalavri, A. Katsifodimos, A survey on the evolution of stream processing systems, The VLDB Journal 33 (2024) 507–541.

[3] I. Mavroudopoulos, K. Tsichlas, A. Gounaris, Sequential pattern detection: similarities and differences across various fields: I. mavroudopoulos et al., Data Mining and Knowledge Discovery 39 (2025) 38.

[4] L. Lan, R. Shi, B. Wang, L. Zhang, N. Jiang, A universal complex event processing mechanism based on edge computing for internet of things real-time monitoring, IEEE Access 7 (2019) 101865–101878.

[5] A. Dhillon, Mcep: a mobile device based complex event processing system for remote healthcare, in: 2018 IEEE International Conference on Internet of Things (iThings), Green Computing and Communications (GreenCom), Cyber, Physical and Social Computing (CPSCom) and Smart Data (SmartData), IEEE, 2018, pp. 203–210.

[6] P. Schneider, F. Xhafa, Anomaly detection and complex event processing over iot data streams: with application to EHealth and patient data monitoring, Academic Press, 2022.

[7] J. Roldán, J. Boubeta-Puig, J. L. Martínez, G. Ortiz, Integrating complex event processing and machine learning: An intelligent architecture for detecting iot security attacks, Expert Systems with Applications 149 (2020) 113251.

[8] M. Bucchi, A. Grez, A. Quintana, C. Riveros, S. Vansummeren, Core: a complex event recognition engine, arXiv preprint arXiv:2111.04635 (2021).

[9] S. Kyrama, A. Gounaris, Exploring alternatives of complex event processing execution engines in demanding cases, in: Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing, 2023, pp. 313–320.

[10] I. Kolchinsky, I. Sharfman, A. Schuster, Lazy evaluation methods for detecting complex events, in: Proc. of DEBS, 2015, pp. 34–45.

[11] O. Poppe, C. Lei, L. Ma, A. Rozet, E. A. Rundensteiner, To share, or not to share online event trend aggregation over bursty event streams, in: Proceedings of the 2021 International Conference on Management of Data, 2021, pp. 1452–1464.

[12] N. Rivetti, N. Zacheilas, A. Gal, V. Kalogeraki, Probabilistic management of late arrival of events, in: Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems, 2018, pp. 52–63.

[13] S. Kyrama, A. Gounaris, Handling out-of-order input arrival in cep engines on the edge combining optimistic, pessimistic and lazy evaluation, arXiv preprint arXiv:2507.01461 (2025).

[14] R. Trisminingsih, S. Bou, T. Amagasa, Efficient pattern matching over out-of-order event streams using sliding buffer, Journal of Information Processing 32 (2024) 963–972.

[15] S. Kyrama, A. Gounaris, Handling out-of-order input arrival in cep engines on the edge combining optimistic, pessimistic and lazy evaluation, arXiv preprint arXiv:2507.01461 (2025).

[16] A. Ziehn, P. M. Grulich, S. Zeuch, V. Markl, Bridging the gap: Complex event processing on stream processing systems., in: EDBT, 2024, pp. 447–460.
[17] C. Reiss, J. Wilkes, J. L. Hellerstein, Google cluster-usage traces: format+ schema, Google Inc., White Paper 1 (2011) 1–14.
[18] H. Zhang, Y. Diao, N. Immerman, On complexity and optimization of expensive queries in complex event processing, in: Proceedings of the 2014 ACM SIGMOD international conference on Management of data, 2014, pp. 217–228.
[19] A. J. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, W. M. White, et al., Cayuga: A general purpose event monitoring system., in: Cidr, volume 7, 2007, pp. 412–422.
[20] Y. Mei, S. Madden, Zstream: a cost-based query processor for adaptively detecting composite events, in: Proc. of SIGMOD, 2009, pp. 193–206.
[21] O. Poppe, C. Lei, S. Ahmed, E. A. Rundensteiner, Complete event trend detection in high-rate event streams, in: Proceedings of the 2017 ACM International Conference on Management of Data, 2017, pp. 109–124.
[22] M. Bucchi, A. Grez, A. Quintana, C. Riveros, S. Vansummeren, Core: a complex event recognition engine, arXiv preprint arXiv:2111.04635 (2021).
[23] G. Cugola, A. Margara, Complex event processing with t-rex, Journal of Systems and Software 85 (2012) 1709–1728.
[24] E. Alevizos, A. Artikis, G. Paliouras, Complex event recognition with symbolic register transducers, Proceedings of the VLDB Endowment 17 (2024) 3165–3177.
[25] A. Rozet, O. Poppe, C. Lei, E. A. Rundensteiner, Muse: Multi-query event trend aggregation, in: Proceedings of the 29th ACM International Conference on Information & Knowledge Management, 2020, pp. 2193–2196.
[26] O. Poppe, C. Lei, E. A. Rundensteiner, D. Maier, Event trend aggregation under rich event matching semantics, in: Proceedings of the 2019 International Conference on Management of Data, 2019, pp. 555–572.
[27] O. Poppe, A. Rozet, C. Lei, E. A. Rundensteiner, D. Maier, Sharon: Shared online event sequence aggregation, in: 2018 IEEE 34th International Conference on Data Engineering (ICDE), IEEE, 2018, pp. 737–748.
[28] N. Rivetti, N. Zacheilas, A. Gal, V. Kalogeraki, Probabilistic management of late arrival of events, in: Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems, 2018, pp. 52–63.
[29] C. Mutschler, M. Philippsen, Adaptive speculative processing of out-of-order event streams, ACM Transactions on Internet Technology (TOIT) 14 (2014) 1–24.
[30] S. Akili, S. Purtzel, M. Weidlich, Decopa: query decomposition for parallel complex event processing, Proceedings of the ACM on Management of Data 2 (2024) 1–26.
[31] M. Yankovitch, I. Kolchinsky, A. Schuster, Hypersonic: A hybrid parallelization approach for scalable complex event processing, in: Proceedings of the 2022 International Conference on Management of Data, 2022, pp. 1093–1107.
[32] K. Chapnik, I. Kolchinsky, A. Schuster, Darling: data-aware load shedding in complex event processing systems, Proceedings of the VLDB Endowment 15 (2021) 541–554.
[33] O. Ouarem, F. Nouioua, P. Fournier-Viger, A survey of episode mining, Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery 14 (2024) e1524.
[34] S. B. Gandreti, P. Sastry, Efficient depth-first search approach for mining frequent chain episodes, in: Proceedings of the 8th International Conference on Data Science and Management of Data (12th ACM IKDD CODS and 30th COMAD), 2024, pp. 66–74.
[35] A. Margara, G. Cugola, G. Tamburrelli, Learning from the past: automated rule generation for complex event processing, in: Proceedings of the 8th ACM international conference on distributed event-based systems, 2014, pp. 47–58.
[36] R. Sattler, S. Kleest-Meißner, S. Lange, M. L. Schmid, N. Schweikardt, M. Weidlich, Disces: systematic discovery of event stream queries, Proceedings of the ACM on Management of Data 3 (2025) 1–26.

[37] C. Balaktsis, I. Mavroudopoulos, M. Comuzzi, A. Gounaris, F. M. Maggi, Discovering comprehensive branched declarative process constraints, in: International Conference on Business Process Management, Springer, 2025, pp. 147–164.

[38] G. Appetito, E. Medvet, V. Gulisano, Automated discovery of cep applications with evolutionary computing, in: Proceedings of the 19th ACM International Conference on Distributed and Event-based Systems, 2025, pp. 33–38.

[39] N. Mehdiyev, J. Krumeich, D. Enke, D. Werth, P. Loos, Determination of rule patterns in complex event processing using machine learning techniques, Procedia Computer Science 61 (2015) 395–401.

[40] R. Mousheimish, Y. Taher, K. Zeitouni, Autocep: automatic learning of predictive rules for complex event processing, in: International conference on service-oriented computing, Springer, 2016, pp. 586–593.

[41] M. U. Simsek, F. Yildirim Okay, S. Ozdemir, A deep learning-based cep rule extraction framework for iot data., Journal of Supercomputing 77 (2021).

[42] A. Mdhaffar, G. Baklouti, Y. Rebai, M. Jmaiel, B. Freisleben, Rl4cep: reinforcement learning for updating cep rules, Complex & Intelligent Systems 11 (2025) 137.

[43] T. Xing, M. R. Vilamala, L. Garcia, F. Cerutti, L. Kaplan, A. Preece, M. Srivastava, Deepcep: Deep complex event processing using distributed multimodal information, in: 2019 IEEE international conference on smart computing (SMARTCOMP), IEEE, 2019, pp. 87–92.

[44] V. Stavropoulos, E. Alevizos, N. Giatrakos, A. Artikis, Optimizing complex event forecasting, in: Proceedings of the 16th ACM International Conference on Distributed and Event-Based Systems, 2022, pp. 19–30.

[45] M. Pitsikalis, E. Alevizos, N. Giatrakos, A. Artikis, Run-time adaptation of complex event forecasting, in: Proceedings of the 19th ACM International Conference on Distributed and Event-based Systems, 2025, pp. 9–20.