CORE: a COmplex event Recognition Engine

Marco Bucchi
PUC Chile
mabucchi@uc.cl

Alejandro Grez
PUC Chile
ajgrez@uc.cl

Andrés Quintana
PUC Chile
afquintana@uc.cl

Cristian Riveros
PUC Chile
cristian.riveros@uc.cl

Stijn Vansummeren
UHasselt – Hasselt University
stijn.vansummeren@uhasselt.be

Abstract

Complex Event Recognition (CER) systems are a prominent technology for finding user-defined query patterns over large data streams in real time. CER query evaluation is known to be computationally challenging, since it requires maintaining a set of partial matches, and this set quickly grows super-linearly in the number of processed events. We present CORE, a novel COmplex event Recognition Engine that focuses on the efficient evaluation of a large class of complex event queries, including time windows as well as the partition-by event correlation operator. This engine uses a novel evaluation algorithm that circumvents the super-linear partial match problem: under data complexity, it takes constant time per input event to maintain a data structure that compactly represents the set of partial matches and, once a match is found, the query results may be enumerated from the data structure with output-linear delay. We experimentally compare CORE against three state-of-the-art CER systems on both synthetic and real-world data. We show that (1) CORE's performance is not affected by the length of the stream, size of the query, or size of the time window, and (2) CORE outperforms the other systems by up to three orders of magnitude on different query workloads.

1 Introduction

Complex Event Recognition (CER for short), also called Complex Event Processing, has emerged as a prominent technology for supporting streaming applications like maritime monitoring [41], network intrusion detection [38], industrial control systems [32] and real-time analytics [47]. CER systems operate on high-velocity streams of primitive events and evaluate expressive event queries to detect *complex events*: collections of primitive events that satisfy some pattern. In particular, CER queries match incoming events on the basis of their content; where they occur in the input stream; and how this order relates to other events in the stream [7, 24, 28]. In this respect, CER queries distinguish themselves from streaming queries supported by engines such as Flink [19] or Spark [12] in that CER queries include regular expression operators like sequencing, disjunction and iteration to express requirements on arrival order, which are not supported by stream processing engines.

CER systems hence aim to detect situations of interest, in the form of complex events, in order to give timely insights for implementing reactive responses to them when necessary. As such, they strive for low latency query evaluation. CER query evaluation, however, is known to be computationally challenging [5, 23, 37, 53, 55, 56]. Indeed, conceptually, evaluating a CER query requires maintaining a set of partial matches, and this set quickly grows super-linearly in the number of processed events, as illustrated next.

Example 1. *Consider that we have a stream* Stock *that is emitting* BUY *and* SELL *events of particular stocks. The events carry the stock name, the volume bought or sold, the price, and a timestamp. The following query (see Section 2 for an introduction to CEQL) retrieves all complex events where a Microsoft stock is sold, followed (not necessarily contiguously) by the trade of Oracle, followed by the trade of a Cisco stock, followed by the sale of an Amazon stock, subject to certain price limits.*

```
SELECT * FROM Stock
WHERE (SELL as ms; (BUY OR SELL) as or; (BUY OR SELL) as cs; SELL as am)
FILTER ms[name="MSFT"] AND ms[price > 26.0]
   AND or[name = "ORCL"] AND or[price < 11.14]
   AND cs[name="CSCO"] AND am[name="AMZN"] AND am[price >= 18.97]
WITHIN 30 minutes.
```

Observe that, within a given time window, the number of partial matches that consist of a Microsoft sale followed by an Oracle trade followed by a Cisco trade may easily be cubic in the number of events in the window. Contemporary CER engines either materialize this set of partial matches to be able to quickly determine, when an Amazon sale occurs, whether the partial match constitutes an answer—or they lazily compute this set of partial matches when an Amazon sale occurs (for the same reason). In both cases, an $\Omega(N^3)$ operation is required to complete the complex event recognition, with N the number of previous events in the window. Even worse, under the so-called skip-till-any-match selection strategy [5], queries that include the iteration operator may have sets of partial matches that grow exponentially *in N [5]. This is clearly detrimental to latency.*

In recognition of the computational challenge of CER query evaluation, a plethora of research has proposed innovative evaluation methods [14, 24, 28]. These methods range from proposing diverse execution models [16, 26, 37, 52], including cost-based database-style query optimizations to trade-off between materialization and lazy computation [37]; to focusing on specific query fragments (e.g., event selection policies [5]) that somewhat limit the super-linear partial match explosion; to using load shedding [55] to obtain low latency at the expense of potentially missing matches; and to employing distributed computation [23, 36]. All of these still suffer, however, from a processing overhead that is super-linear in N. As such, their scalability is limited to CER queries over a short time window, as we show in Section 6. Unfortunately, for applications such as maritime monitoring [41], network intrusion [38] and fraud detection [13], long time windows are necessary.

In this paper, we present CORE, a novel COmplex event Recognition Engine that focuses on the efficient evaluation of a large class of complex event queries. In particular, CORE uses a novel evaluation algorithm that circumvents the super-linear partial match problem: under data complexity the algorithm takes *constant* time per input event to maintain a data structure that compactly represents the set of partial and full matches in a size that is at most linear in N. Once a match is found, complex event(s) may be enumerated from the data structure with *output-linear delay*, meaning that the time required to output recognized complex event C is linear in the size of C. This complexity is asymptotically optimal since any evaluation algorithm needs to at least inspect every input tuple and list the query answers. We stress that, in particular, the runtime of CORE's complexity is independent of the number of partial matches, as well as the length of the time window being used.

Contributions. Our contributions are as follows.

(1) We introduce CEQL, a functional CER query language built around common CER operators, including sequencing, disjunction, filtering, iteration, projection, partition-by, and time-windows. We base CEQL on an time-interval extension of CEL, a formal logic for CER proposed in [29–31].

(2) CORE's evaluation algorithm is based on a computational model called Complex Event Automata (CEA). A subset of the authors introduced CEA in [29–31], where also a theoretical evaluation algorithm was presented that requires only constant time per event, followed by output-linear delay enumeration of the output. A key limitation, however, is that it cannot deal with time windows. As such, it needs to memorize all previously seen events — no matter how long ago — and can never prune its internal state to clear space for further processing. In addition, the algorithm cannot deal with the partition-by operator that requires all matching events to have identical values in a particular attribute. The current paper presents an entirely new evaluation algorithm for CEA that deals with time windows and partitioning, thereby lifting these limitations.

(3) We show that the algorithm is practical. We experimentally compare CORE against state-of-the-art CER engines on both synthetic and real-world data. Over synthetic data streams, our experiments show that CORE's performance is stable: the throughput is not affected by the length of the stream, size of the query, or size of the time window, working for queries and time windows of arbitrary length. Furthermore, CORE outperforms existing systems by two to three orders of magnitude on different query workloads. This trend is confirmed over real data streams, giving evidence in practice of the algorithm's advantages.

The structure of the paper is as follows. We finish this section by discussing further related work not already mentioned above. We continue by introducing CEQL in Section 2, and define its syntax and semantics in Section 3. We present the CEA computation model in Section 4. The algorithm and its data structures are described in Section 5, which also discusses implementation aspects of CORE. We dedicate Section 6 to experiments and conclude in Section 7.

Because of space limitations, certain details, formal statements and proofs are deferred the Appendix.

Further Related Work CER systems are usually divided into three approaches: automata-based, tree-based, and logic-based, with some systems (e.g., [2, 22, 55]) being hybrids. We refer to recent surveys [7, 14, 24, 28] of the field for in-depth discussion of these classes of systems. CORE falls within the class of automata-based systems [5, 22, 25, 26, 40, 42, 44, 45, 48, 52, 53, 56]. These systems use automata as their underlying execution model. As illustrated by Example 1, these systems either materialize a super-linear number of partial matches, or recompute them on the fly. Conceptually, almost all of these works propose a method to reduce the materialization/recomputation cost by representing partial matches in a more compact manner. CORE is the first system to propose a representation of partial matches with formal, proven, and optimal performance guarantees: linear in the number of seen events, with constant update cost, and output-linear enumeration delay. CEQL focuses on the recognition of complex events, and supports all common CER operators in this respect (see Section 3). Other features considered in the literature, orthogonal to recognition, such as aggregation [44, 45], integration of non-event data sources [56], and parallel or distributed [23, 40, 48] execution are left for future work.

Tree- and logic-based systems [2, 11, 15, 21, 35, 37] typically evaluate queries by constructing and evaluating a tree of CER operators, much like relational database systems evaluate relational algebra queries. These evaluation trees do no have the formal, optimal performance guarantees offered by CORE.

Query evaluation with bounded delay has been extensively studied for relational database queries [17, 18, 18, 20, 34, 49], where it forms an attractive evaluation method, especially when query output risks being much bigger than the size of the input. CORE applies this methodology to the CER domain which differs from the relational setting in the choice of query operators, in particular the presence of operators like sequencing and Kleene star (iteration). In this respect, CORE's evaluation algorithm is closer the work on query evaluation with bounded delay over words and trees [8–10, 27]. Those works, however, do not consider enumeration with time window constraints as we do here, nor have they been the subject of implementation in a concrete system. In particular, the compact data structure that underlies CEQL's evaluation algorithm is inspired on the Enumerable Compact Set of [39].

2 CEQL by example

Numerous languages for expressing CER queries have been proposed in the literature (see, e.g., [7, 24, 28] for a survey). While these languages often exhibit subtle semantic differences, they are based on a common set of operators for expressing patterns, including, for example sequencing, disjunction, iteration, and filtering, among others [7, 28]. CORE's query language, dubbed CEQL (for Complex Event Query Language), is a practical CER query language based on *Complex Event Logic* (CEL for short)—a formal logic that is built from these common operators and whose expressiveness and complexity have been studied in [29–31]. CEQL extends CEL by adding support for time windows as well as the partition-by event correlation operator. In this section, we introduce CEQL by means of examples. A formal definition is given in Section 3.

We continue our running example of detecting patterns of interest in a stream of stock ticks presented in Example 1. Suppose that we are interested in all triples of SELL events where the first is a sale of Microsoft over 100 USD, the second is a sale of Intel (of any price), and the third is a sale of Amazon below 2000 USD. Query Q_1 in Figure 1 expresses this in CEQL. In Q_1, the FROM clause indicates the streams to read events from, while the WHERE clause indicates the pattern of atomic events that need to be matched

```
SELECT * FROM Stock
WHERE SELL as msft; SELL as intel; SELL as amzn
FILTER msft[name="MSFT"] AND msft[price > 100]
    AND intel[name="INTC"]
   AND amzn[name="AMZN"] AND amzn[price < 2000]
```

(Q_1)

```
SELECT b FROM Stock
WHERE SELL as s; BUY as b
PARTITION BY [name], [volume]
WITHIN 1 minute
```

(Q_2)

```
SELECT MAX *  FROM Stock
WHERE SELL as low; SELL+ as s1; SELL as high; SELL+ as s2; SELL as end
FILTER low[price < 100] AND s1[price >= 100] AND s1[price <= 2000]
   AND high[price > 2000] AND s2[price >= 100] AND s2[price <= 2000]
   AND end[price < 100]
PARTITION BY [name]
```

(Q_3)

Figure 1: CEQL queries on a Stock stream.

in the stream. This can be any unary Complex Event Logic (CEL) expression [31]. In Q_1, the CEL expression

```
SELL as msft; SELL as intel; SELL as amzn
```

indicates that we wish to see three SELL events and that we will refer to the first, second and third events by means of the variables msft, intel and amzn, respectively. In particular, the semicolon operator (;) indicates sequencing among events. It is important to note that sequencing in CORE is non-contiguous. As such, the msft event needs not be followed immediately by the intel event—there may be other events in between, and similarly for amzn. The FILTER clause requires the msft event to have MSFT in its name attribute, and a price above 100. It makes similar requirements on the intel and amzn events.

The conditions in a CEQL FILTER clause can only express predicates on single events. Correlation among events, in the form of equi-joins, is supported in CEQL by the PARTITION BY clause. This feature is illustrated by query Q_2 in Figure 1, which detects all pairs of SELL and subsequent BUY events of the same stock and the same volume. In particular, there, the PARTITION BY clause requests that all matched events have the same values in the name and volume attributes. The WITHIN clause specifies that the matched pattern must be detected within 1 minute. In CORE, each event is assigned the time at which it arrives to the system, so we do not assume that events include a special attribute representing time, as some other systems do. Finally, the SELECT clause ensures that, from the matched pair of events, only the event in variable b is returned.

In general, the pattern specified in the WHERE clause in a CEQL query may include other operators such as disjunction (denoted OR, see Example 1) and iteration (also known as Kleene closure, denoted +). These may be freely nested in the WHERE clause. Query Q_3 illustrates the use of iteration. In query Q_3, 100 and 2000 are two values representing a lower and upper limit price, respectively. Q_3 is a segmentation query: it looks for sequences of SELL events pertaining to the same stock symbol where the sale price is initially below 100 (captured by the low variable), then between 100 and 2000 (captured by s1), then above 2000 (high), then again between 100 and 2000 (u2), to end below 100 (end). Importantly, because of the Kleene closure iteration operator, s1 (and later, s2) capture all sales of the stock in the $[100, 2000]$ price range. The MAX operator in the SELECT clause is an example of a selection strategy [5, 28, 31]: it ensures that s1 and s2 are bound to maximal sequences of events that satisfy the pattern. If this policy were not specified, CEQL would adopt the skip-till-any-match policy [5, 28] by default, which also returns complex events with s1 and s2 containing only subsets of this maximal sequences.

3 CEQL syntax and semantics

In this section, we give the formal syntax and semantics of CEQL. We start by defining CORE's event model.

Events, complex events, and valuations. We assume given a set of *event types* $\mathbf{T}$ (consisting, e.g., of the event types BUY and SELL in our running example), a set of *attribute names* $\mathbf{A}$ (e.g., name, price, etc) and a set of *data values* $\mathbf{D}$ (e.g. integers, strings, etc.). A *data-tuple* t is a partial mapping that maps attribute names from $\mathbf{A}$ to data values in $\mathbf{D}$. Each data-tuple is associated to an event type. We denote by $t(a) \in \mathbf{D}$ the value of the attribute $a \in \mathbf{A}$ assigned by t, and by $t(\text{type}) \in \mathbf{T}$ the event type of t. If t is not defined on attribute a, then we write $t(a) = \text{NULL}$.

A *stream* is a possibly infinite sequence $S = t_0 t_1 t_2 \dots$ of data-tuples. Given a set $D \subseteq \mathbb{N}$, we define the set of data tuples $S[D] = \{t_i \mid i \in D\}$. A *complex event* is a pair $C = ([i, j], D)$ where $i \leq j \in \mathbb{N}$ and D is a subset of $\{i, \dots, j\}$. Intuitively, given a stream $S = t_0 t_1 \dots$ the interval $[i, j]$ of C represents the subsequence $t_i t_{i+1} \dots t_j$ of S where the complex event C happens and $S[D]$ represents the data-tuples from S that are relevant for C. We write $C(\text{time})$ to denote the time-interval $[i, j]$, and $C(\text{start})$ and $C(\text{end})$ for i and j, respectively. Furthermore, we write $C(\text{data})$ to denote the set D.

To define the semantics of CEQL, we will also need the following notion. Let $\mathbf{X}$ be a set of *variables*, which includes all event types, $\mathbf{T} \subseteq \mathbf{X}$. A *valuation* is a pair $V = ([i, j], \mu)$ with $[i, j]$ a time interval as above and μ a mapping that assigns subsets of $\{i, \dots, j\}$ to variables in $\mathbf{X}$. Similar to complex events, we write $V(\text{time})$, $V(\text{start})$, and $V(\text{end})$ for $[i, j]$, i, and j, respectively, and $V(X)$ for the subset of $\{i, \dots, j\}$ assigned to X by μ.

We write C_V for the complex event that is obtained from valuation V by forgetting the variables in V, and retaining only its positions: $C_V(\text{time}) = V(\text{time})$ and $C_V(\text{data}) = \bigcup_{X \in \mathbf{X}} V(X)$. The semantics of CEQL will be defined in terms of valuations, which are subsequently transformed into complex events in this manner.

Predicates A (unary) *predicate* is a possibly infinite set P of data-tuples. For example, P could be the set of all tuples t such that $t(\text{price}) \geq 100$. A data-tuple t *satisfies* predicate P, denoted $t \models P$, if, and only if, $t \in P$. We generalize this definition from data-tuples to sets by taking a "for all" extension: a set of data-tuples T satisfies P, denoted by $T \models P$, if, and only if, $t \models P$ for all $t \in T$.

CEQL. Syntactically, a CEQL query has the form:

```
SELECT          [selection-strategy] < list-of-variables >
FROM            < list-of-streams >
WHERE           < CEL-formula >
[PARTITION BY   < list-of-attributes >]
[WITHIN         < time-value >]
```

Specifically, the WHERE clause consists of a formula in Complex Event Logic (CEL) [31], whose abstract syntax is given by the following grammar:

$$\varphi := R \mid \varphi \text{ AS } X \mid \varphi \texttt{ FILTER } X[P] \mid \varphi \texttt{ OR } \varphi \mid \varphi \,;\, \varphi \mid \varphi+ \mid \pi_L\ (\varphi).$$

In this grammar, R is a event type in $\mathbf{T}$, X is a variable in $\mathbf{X}$, P is a predicate, and L is a subset of variables in $\mathbf{X}$.[1]

The semantics of CEQL is now as follows. Conceptually, a CEQL query first evaluates its FROM clause, then its PARTITION BY clause, and subsequently its WHERE, SELECT, and WITHIN clauses (in that order). The FROM clause merely specifies the list of streams registered to the system from which events should be inspected. All these streams are logically merged into a single stream S that is processed by the subsequent clauses. The PARTITION BY clause, if present, logically partitions this stream into multiple substreams $S_1, S_2, \ldots$, and executes the WHERE-SELECT-WITHIN clauses on each substream separately. The union of the outputs generated for each substream constitute the final output. Concretely, every S_i is a maximal subsequence of S such that for every pair of tuples t and t' occurring in S_i, and for every attribute a mentioned in the PARTITION BY clause, it holds that $t(a) \neq \texttt{NULL}$, $t'(a) \neq \texttt{NULL}$, and $t(a) = t'(a)$. As such, all tuples in S_i share the same value in every attribute of the PARTITION BY clause.

The semantics of the WHERE-SELECT-WITHIN clauses is as follows. CEQL's WHERE clause is derived from the semantics of CEL[2], which is inductively defined in Table 2. Concretely, given a stream $S = t_0t_1t_2\ldots$ (or one of the substreams S_i if the query has a PARTITION BY clause), a CEL formula φ evaluates to a set of valuations, denoted $⦃\varphi⦄(S)$. The base case is when φ is an event type R. In that case $⦃\varphi⦄(S)$ contains all valuations whose time-interval is a single position i, such that the data-tuple t_i at position i in S is of type R. Furthermore, the valuation is such that variable R (recall that $\mathbf{T} \subseteq \mathbf{X}$) stores only position i and all other variables are empty. The AS clause is a variable assignment that takes an existing valuation $V \in ⦃\varphi⦄(S)$ and extends it by gathering all positions $\cup_Y V(Y)$ in variable X, keeping all other variables as in V. The filter clause FILTER $X[P]$ retains only those valuations for which the content of variable X satisfies predicate P, and the OR clause takes the union of two sets of valuations. The sequencing operator uses the time-interval for capturing all pairs of valuations in which the first is chronologically followed by the second. Specifically, $⦃\varphi_1\,;\,\varphi_2⦄(S)$ takes $V_1 \in ⦃\varphi_1⦄(S)$ and $V_2 \in ⦃\varphi_2⦄(S)$ such that V_2 is after V_1 (i.e., $V_1(\text{end}) < V_2(\text{start})$) and joins them into one valuation V, where the time interval is given by the start of V_1 and the end of V_2. The semantics of iteration $\varphi+$ is defined as the application of sequencing (;) one or more times over the same formula. The projection π_L modifies valuations by setting all variables that are not in L to empty.

The WHERE part of a CEQL query hence returns a set of valuations when evaluated over a stream. The SELECT clause, if it does not mention a selection strategy, corresponds to a projection in CEL, and hence operates on this set accordingly. If it does specify

$$\begin{aligned}
⦃R⦄(S) &= \{V \mid V(\text{time}) = [i,i] \ \wedge\ t_i(\text{type}) = R \\
&\qquad \wedge\ V(R) = \{i\} \ \wedge\ \forall X \neq R.\, V(X) = \emptyset\ \} \\
⦃\varphi \text{ AS } X⦄(S) &= \{V \mid \exists V' \in ⦃\varphi⦄(S).\ V(\text{time}) = V'(\text{time}) \\
&\qquad \wedge\ V(X) = \cup_Y V'(Y) \\
&\qquad \wedge\ \forall Z \neq X.\, V(Z) = V'(Z)\ \} \\
⦃\varphi \texttt{ FILTER } X[P]⦄(S) &= \{V \mid V \in ⦃\varphi⦄(S) \wedge V(X) \models P\ \} \\
⦃\varphi_1 \texttt{ OR } \varphi_2⦄(S) &= ⦃\varphi_1⦄(S)\ \cup\ ⦃\varphi_2⦄(S) \\
⦃\varphi_1\,;\,\varphi_2⦄(S) &= \{V \mid \exists V_1 \in ⦃\varphi_1⦄(S), V_2 \in ⦃\varphi_2⦄(S). \\
&\qquad V_1(\text{end}) < V_2(\text{start}) \\
&\qquad \wedge\ V(\text{time}) = [V_1(\text{start}), V_2(\text{end})] \\
&\qquad \wedge\ \forall X.\, V(X) = V_1(X) \cup V_2(X)\ \} \\
⦃\varphi+⦄(S) &= ⦃\varphi⦄(S)\ \cup\ ⦃\varphi\,;\,\varphi+⦄(S) \\
⦃\pi_L\,(\varphi)⦄(S) &= \{V \mid \exists V' \in ⦃\varphi⦄(S).\ V(\text{time}) = V'(\text{time}) \\
&\qquad \wedge \forall X \in L.\, V(X) = V'(X) \\
&\qquad \wedge\ \forall X \notin L.\, V(X) = \emptyset\}
\end{aligned}$$

Figure 2: The semantics of a CEL formulas.

a selection strategy, then a CEL projection is applied, followed by removing certain valuations from the set. We refer the interested reader to [31] for a definition and discussion of selection strategies. Finally, if ϵ is a time-interval, then the WITHIN clause operate on the resulting set of valuations as follows:

$$⦃\varphi \texttt{ WITHIN } \epsilon⦄(S) \ = \ \{V \in ⦃\varphi⦄(S) \mid V(\text{end}) - V(\text{start}) \leq \epsilon\}.$$

Complex Event Semantics. The semantics defined above is one where CEL and CEQL queries return valuations. In CER systems, it is customary, however, to return complex events instead. The complex event semantics of CEL and CEQL is obtained by first evaluating the query under the valuation semantics, and then removing variables altogether. That is, if φ is a CEL formula or CEQL query, its complex event semantics $[\![\varphi]\!](S)$ is defined by $[\![\varphi]\!](S) := \{C_V \mid V \in ⦃\varphi⦄(S)\}$. For the rest of this paper, we will be interested in efficiently computing the complex event semantics $[\![\varphi]\!](S)$. We stress, however, that our techniques can be extended to also efficiently compute the valuation semantics instead.

4 Query compilation

As explained in Section 3, evaluating a CEQL query essentially amounts to evaluating the query's SELECT-WHERE-WITHIN clauses on either a single stream, or multiple different substreams thereof (for a query with PARTITION BY). We will therefore focus in this section and the next on the efficient evaluation of SELECT-WHERE-WITHIN on a single stream, or, equivalently, on the evaluation of a CEQL query Q without PARTITION BY and with only one stream mentioned in the FROM clause.

Concretely, in CORE we first compile the SELECT-WHERE part of Q into a *Complex Event Automaton (CEA for short)* [29, 31], which is a form of finite state automaton that produces complex events. CORE's evaluation algorithm is then defined in terms of CEA: it takes as input a CEA $\mathcal{A}$, the (optional) time window ϵ specified in the WITHIN clause of Q, and a stream S, and uses this to compute $[\![Q]\!](S)$. The evaluation algorithm is described in Section 5. Here, we introduce CEA.

Roughly speaking, a CEA is similar to a standard finite state automaton. The difference is that a standard finite state automaton processes finite strings and also has transitions of the form $p \xrightarrow{\sigma} q$

[1]Observe that CEL includes FILTER, there is hence no separate FILTER clause in CEQL. For convenience, in CEQL queries we use φ FILTER θ_1AND θ_2 in the WHERE clause as a shorthand for (φ FILTER θ_1) FILTER θ_2, and φ FILTER θ_1 OR θ_2 as shorthand for (φ FILTER θ_1) OR (φ FILTER θ_2).
[2]Note that the semantics used in this paper is an extension of the semantics of CEL in [31] since we also consider the time interval as part of the complex event.

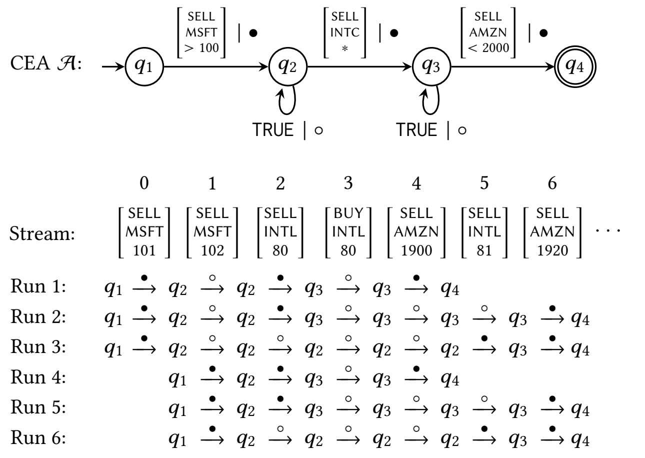

CEA $\mathcal{A}$:
SELL MSFT > 100 | •
SELL INTC * | •
SELL AMZN < 2000 | •
TRUE | ∘
TRUE | ∘
Stream:
0 SELL MSFT 101
1 SELL MSFT 102
2 SELL INTL 80
3 BUY INTL 80
4 SELL AMZN 1900
5 SELL INTL 81
6 SELL AMZN 1920
Run 1: $q_1 \xrightarrow{\bullet} q_2 \xrightarrow{\circ} q_2 \xrightarrow{\bullet} q_3 \xrightarrow{\circ} q_3 \xrightarrow{\bullet} q_4$
Run 2: $q_1 \xrightarrow{\bullet} q_2 \xrightarrow{\circ} q_2 \xrightarrow{\bullet} q_3 \xrightarrow{\circ} q_3 \xrightarrow{\circ} q_3 \xrightarrow{\circ} q_3 \xrightarrow{\bullet} q_4$
Run 3: $q_1 \xrightarrow{\bullet} q_2 \xrightarrow{\circ} q_2 \xrightarrow{\circ} q_2 \xrightarrow{\circ} q_2 \xrightarrow{\circ} q_2 \xrightarrow{\bullet} q_3 \xrightarrow{\bullet} q_4$
Run 4: $q_1 \xrightarrow{\bullet} q_2 \xrightarrow{\bullet} q_3 \xrightarrow{\circ} q_3 \xrightarrow{\bullet} q_4$
Run 5: $q_1 \xrightarrow{\bullet} q_2 \xrightarrow{\bullet} q_3 \xrightarrow{\circ} q_3 \xrightarrow{\circ} q_3 \xrightarrow{\circ} q_3 \xrightarrow{\bullet} q_4$
Run 6: $q_1 \xrightarrow{\bullet} q_2 \xrightarrow{\circ} q_2 \xrightarrow{\circ} q_2 \xrightarrow{\circ} q_2 \xrightarrow{\bullet} q_3 \xrightarrow{\bullet} q_4$

Figure 3: A CEA representing Q_1 from Figure 1 and some of its runs on an example stream.

with q, p states and σ a symbol from some finite alphabet, whereas a CEA processes possibly unbounded streams of data-tuples and has transitions of the form $p \xrightarrow{P/m} q$ with p and q states, P a predicate and m an action, which can be *marking* (•) or *unmarking* (∘). The semantics of such a transition $p \xrightarrow{P/m} q$ is that, when a new tuple t arrives in the stream and the CEA is in state p, if t satisfies P then the CEA moves to state q and applies the action m: if m is a marking action then the event t will be part of the output complex event once a final state is reached, otherwise it will not.

Example 2. *In Figure 3 we show a CEA $\mathcal{A}$ that represents query Q_1 from Figure 1. There, we depict predicates by listing, in array notation, the event type, the requested value of the name attribute, and the constraint on the price attribute. The initial state is q_1 and there is only one final state: q_4. The figure also shows an example stream S, and several runs of $\mathcal{A}$ on S. Every run shown is accepting (i.e., ends in an accepting state of the automaton), and as such returns a complex event. The time of this complex event is the interval $[i, j]$ with i the position where the run starts, and j the position where the run ends. The data of this complex event consists of all positions marked by the run. For example, the complex event C_1 output by run 1 is $([0, 4], \{0, 2, 4\})$; the complex event C_2 output by run 2 is $([0, 6], \{0, 2, 6\})$, and so on.*

Formally, a *Complex Event Automaton (CEA)* is a tuple $\mathcal{A} = (Q, \Delta, q_0, F)$ where Q is a finite set of states, $\Delta \subseteq Q \times \mathbf{P} \times \{\bullet, \circ\} \times (Q \setminus \{q_0\})$ is a finite transition relation, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of final states. We will denote transitions in Δ by $q \xrightarrow{P/m} q'$. A *run of $\mathcal{A}$ over stream S from positions i to j* is a sequence $\rho := q_i \xrightarrow{P_i/m_i} q_{i+1} \xrightarrow{P_{i+1}/m_{i+1}} \dots \xrightarrow{P_j/m_j} q_{j+1}$ such that q_i is the initial state of $\mathcal{A}$ and for every $k \in [i, j]$ it holds that $q_k \xrightarrow{P_k/m_k} q_{k+1} \in \Delta$ and $t_k \models P_k$. A run ρ is *accepting* if $q_{j+1} \in F$. An accepting run ρ of $\mathcal{A}$ over S from i to j naturally defines the complex event $C_\rho := ([i, j], \{k \mid i \le k \le j \wedge m_k = \bullet\})$. If position i and j are clear from the context, we say that ρ is a run of $\mathcal{A}$ over S. Finally, we define the semantics of $\mathcal{A}$ over a stream S as $[\![\mathcal{A}]\!](S) := \{C_\rho \mid \rho \text{ is an accepting run of } \mathcal{A} \text{ over } S\}$.

We note that in this paper, because we consider complex events with time intervals, a run may start at an arbitrary position i in the stream, which differs from the semantics of CEA considered in [29, 31] where complex events do not have time intervals and runs always start at the beginning of the stream. It is also important to note that in the definition above no transition can re-enter the initial state q_0; this will be important for defining the time-interval of the output complex events in Section 5. This requirement on the initial state is without loss of generality, since any incoming transitions into the initial state q_0 may be removed without modifying semantics by making a copy q_0' of q_0 (also copying its outgoing transitions) and rewrite any transition into q_0 to go to q_0' instead.

The usefulness of CEA comes from the fact that CEL can be translated into CEA [29, 31]. Because the SELECT-WHERE part of a CEQL query is in essence a CEL formula, this reduces the evaluation problem of the SELECT-WHERE-WITHIN part of CEQL query into the evaluation problem for CEA, in the following sense.[3]

Theorem 1. *For every CEL formula φ we can construct a CEA $\mathcal{A}$ of size linear in φ such that for every ϵ:*

$$[\![\varphi \texttt{ WITHIN } \epsilon]\!](S) = \{C \mid C \in [\![\mathcal{A}]\!](S) \wedge C(\text{end}) - C(\text{start}) \le \epsilon\}.$$

Our evaluation algorithm will compute the right-hand side in this equation. It requires, however, that the input CEA $\mathcal{A}$ is *I/O-deterministic*: for every pair of transitions $q \xrightarrow{P_1/m_1} q_1$ and $q \xrightarrow{P_2/m_2} q_2$ from the same state q, if $P_1 \cap P_2 \neq \emptyset$ then $m_1 \neq m_2$. In other words, an event t may trigger both transitions at the same time (i.e., $t \models P_1$ and $t \models P_2$) only if one transition marks the event, but the other does not. In [29, 31], it was shown that any CEA can be I/O-determinized. The determinization method we use is based on the classical subset construction of finite state automata, thus possibly adding an exponential blow-up in the number of states. To avoid this exponential blow-up in practice, in CORE we determinize the CEA not all at once, but on the fly while the stream is being processed. Importantly, we cache the previous states that we have computed. In Section 5.4 we discuss the internal implementation of CORE and how this exponential factor impacts system performance.

5 Evaluation algorithm

In this section, we present an efficient evaluation algorithm that, given a CEA $\mathcal{A}$, time window ϵ, and stream S, computes the set

$$[\![\mathcal{A}]\!]^\epsilon(S) := \{C \mid C \in [\![\mathcal{A}]\!](S) \wedge C(\text{end}) - C(\text{start}) \le \epsilon\}.$$

In fact, our algorithm will compute this set *incrementally*: at every position j in the stream, it outputs the set

$$[\![\mathcal{A}]\!]^\epsilon_j(S) := \{C \in [\![\mathcal{A}]\!]^\epsilon \mid C(\text{end}) = j\}.$$

The algorithm works by incrementally maintaining a data structure that *compactly* represents partial outputs (i.e., fragments of S that later may cause a complex event to be output). Whenever a new tuple arrives, it takes *constant time* (in data complexity [51]) to update the data structure. Furthermore, from the data structure, we may at each position j enumerate the complex events of $[\![\mathcal{A}]\!]^\epsilon_j(S)$ one by one, without duplicates, and with *output-linear delay* [31, 49]. This means that the time required to print the first complex event

[3]We remark that any selection policy mentioned in the SELECT clause can also be expressed using CEA, see [29, 31].

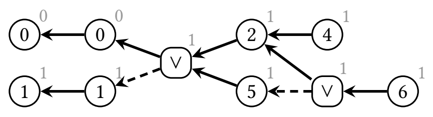


Figure 4: An example tECS. Union nodes are labeled by ∨ while non-union nodes are depicted as circles. Left and right children of union nodes are indicated by dashed and solid edges, respectively. The maximum-start of each node is at its top-right in grey.

of $[\![\mathcal{A}]\!]_j^\epsilon(S)$ from the data structure, or any of the following ones, is linear in the size of complex event being printed. Note in particular that the data complexity of our algorithm is asymptotically optimal: any evaluation algorithm needs to at least inspect every input tuple and list the query answers. Also note that, because it takes constant time to update the data structure with a new input event, the size of our data structure is at most linear in the number of seen events.

We first define the data structure in Section 5.1, and operations on it in Section 5.2. The evaluation algorithm is given in Section 5.3 and aspects of its implementation in Section 5.4.

5.1 The data structure

Our data structure is called a *timed Enumerable Compact Set* (tECS). Figure 4 gives an example. Specifically, a tECS is a directed acyclic graph (DAG) $\mathcal{E}$ with two kinds of nodes: union nodes and non-union nodes. Every union node u has exactly two children, the left child $\mathsf{left}(\mathsf{u})$ and the right child $\mathsf{right}(\mathsf{u})$, which are depicted by dashed and solid edges in Figure 4, respectively. Every non-union node n is labeled by a stream position (an element of $\mathbb{N}$) and has at most one child. If non-union node n has no child it is called a *bottom node*, otherwise it is an *output node*. We write $\mathsf{pos}(\mathsf{n})$ for the label of non-union node n and $\mathsf{next}(\mathsf{o})$ for the unique child of output node o. To simplify presentation in what follows, we will range over nodes of any kind by n; over bottom, output, and union nodes by b, o, and u, respectively.

A tECS represents sets of complex events or, more precisely, sets of *open* complex events. An *open complex event* is a pair (i, D) where $i \in \mathbb{N}$ and D is a finite subset of $\{i, i+1, \ldots\}$. An open complex event is almost a complex event, with a start time i and set of positions D, but where the end time is missing: if we choose $j \geq \max(D)$, then $([i, j], D)$ is a complex event. Intuitively, when processing a stream, the open complex events represented by a tECS are partial results that may later become full complex events.

The representation is as follows. Let $\bar{p} = \mathsf{n}_1, \mathsf{n}_2, \ldots, \mathsf{n}_k$ be a *full-path* in $\mathcal{E}$ such that n_k is a bottom node. Then $\bar{p}$ represents the open complex event $[\![\bar{p}]\!]_{\mathcal{E}} = (i, D)$ where $i = \mathsf{pos}(n_k)$ is the label of the bottom node n_k, and D is the set of labels of the other non-union nodes in $\bar{p}$. For instance, for the full-path $\bar{p} = 4, 2, \vee, 1, 1$ in Figure 4, we have $[\![\bar{p}]\!]_{\mathcal{E}} = (1, \{1, 2, 4\})$. Given a node n, the set $[\![\mathsf{n}]\!]_{\mathcal{E}}$ of open complex events represented by n consists of all open complex events $[\![\bar{p}]\!]_{\mathcal{E}}$ with $\bar{p}$ a full-path in $\mathcal{E}$ starting at n.

Example 3. *In Figure 4 we have* $[\![4]\!]_{\mathcal{E}} = \{(0, \{0, 2, 4\}), (1, \{1, 2, 4\})$ *and* $[\![6]\!]_{\mathcal{E}} = \{(0, \{0, 2, 6\}), (0, \{0, 5, 6\}), (1, \{1, 2, 6\}), (1, \{1, 5, 6\})$*.*

Remember that our purpose in constructing $\mathcal{E}$ is to be able to enumerate the set $[\![\mathcal{A}]\!]_j^\epsilon(S)$ at every j. To that end, it will be necessary to enumerate, for certain nodes n in $\mathcal{E}$, the set

$$[\![\mathsf{n}]\!]_{\mathcal{E}}^\epsilon(j) \ := \ \{\, ([i, j], D) \mid (i, D) \in [\![\mathsf{n}]\!]_{\mathcal{E}} \wedge j - i \leq \epsilon \,\},$$

i.e., all open complex events represented by n that, when closed with j, are within a time window of size ϵ.

Example 4. *The set of all complex events output by the accepting runs of CEA $\mathcal{A}$ in Figure 3 can be retrieved from the tECS of Figure 4 by enumerating $[\![4]\!]_{\mathcal{E}}^6(4)$ and $[\![6]\!]_{\mathcal{E}}^6(6)$, which contains all complex events output at position 4 and 6, respectively.*

A straightforward algorithm for enumerating $[\![\mathsf{n}]\!]_{\mathcal{E}}^\epsilon(j)$ is to perform a depth-first search (DFS) starting at n. During the search we maintain the full-path $\bar{p}$ from n to the currently visited node m. Every time we reach a bottom node, we check whether $[\![\bar{p}]\!]_{\mathcal{E}}$ satisfies the time window, and, if so, output it. There are two problems with this algorithm. First, it does not satisfy our delay requirements: the DFS may spend unbounded time before reaching a full-path $\bar{p}$ that satisfies the time window and actually generates output. Second, it may enumerate the same complex event multiple times. This happens if there are multiple full-paths from n that represent the same open complex event. We therefore impose three restrictions on the structure of a tECS.

The first restriction is that $\mathcal{E}$ needs to be time-ordered, which is defined as follows. For a node n define its *maximum-start*, denoted $\max(\mathsf{n})$, as $\max(\mathsf{n}) = \max\left(\{i \mid (i, D) \in [\![\mathsf{n}]\!]_{\mathcal{E}}\}\right)$. A tECS is *time-ordered* if (1) every node n carries $\max(\mathsf{n})$ as an extra label (so that it can be retrieved in $O(1)$ time) and (2) for every union node u it holds that $\max(\mathsf{left}(\mathsf{u})) \geq \max(\mathsf{right}(\mathsf{u}))$. For instance, the tECS of Figure 4 is time-ordered. The DFS-based enumeration algorithm described above can be modified to avoid needless searching on a time-ordered tECS: before starting the search, first check that $j - \max(n) \leq \epsilon$. If so, $[\![\mathsf{n}]\!]_{\mathcal{E}}^\epsilon(j)$ is non-empty and we perform the DFS-based enumeration. However, when we traverse a union node u we always visit $\mathsf{left}(\mathsf{u})$ before $\mathsf{right}(\mathsf{u})$. Moreover, we only visit $\mathsf{right}(\mathsf{u})$ if $j - \max(\mathsf{right}(\mathsf{u})) \leq \epsilon$. Otherwise, $\mathsf{right}(\mathsf{u})$ and its descendants do not contribute to $[\![\mathsf{n}]\!]_{\mathcal{E}}^\epsilon(j)$, and can be skipped.

The second restriction is that $\mathcal{E}$ needs to be k*-bounded* for some $k \in \mathbb{N}$, which is defined as follows. Define the *(left) output-depth* $\mathsf{odepth}(\mathsf{n})$ of a node n recursively as follows: if n is a non-union node, then $\mathsf{odepth}(\mathsf{n}) = 0$; otherwise, $\mathsf{odepth}(\mathsf{n}) = \mathsf{odepth}(\mathsf{left}(\mathsf{n})) + 1$. The output depth tell us how many union nodes we need to traverse to the left before we find a non-union node that, therefore, produces part of the output. Then, $\mathcal{E}$ is k*-bounded* if $\mathsf{odepth}(\mathsf{n}) \leq k$ for every node n. For instance, the tECS of Figure 4 is 1-bounded. The k-boundedness restriction is necessary because even though we know that, on a time-ordered $\mathcal{E}$, we will find a complex event to output by consistently visiting left children of union nodes, starting from n, there may be an unbounded number of union nodes to visit before reaching a bottom node. In that case, the length of the corresponding full-path $\bar{p}$ risks being significantly bigger than the size of $[\![\bar{p}]\!]_{\mathcal{E}}$, violating the output-linear delay.

The third restriction on $\mathcal{E}$, needed to ensure that we may enumerate without duplicates, is for it to be *duplicate-free*. Here, $\mathcal{E}$ is duplicate-free if all of its nodes are duplicate-free, and a node n is duplicate-free if for every pair of distinct full-paths $\bar{p}$ and $\bar{q}$ that start at n we have $[\![\bar{p}]\!]_{\mathcal{E}} \neq [\![\bar{q}]\!]_{\mathcal{E}}$.

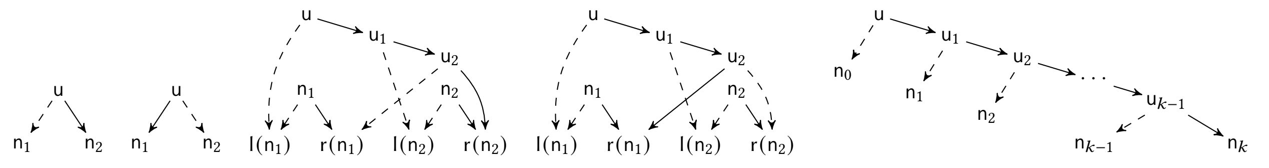


Figure 5: Gadgets for the implementation of the union method. The u nodes are union nodes, where the dashed and bold arrows symbolize the left and right nodes, respectively.

Theorem 2. *Fix k. For every k-bounded and time-ordered tECS $\mathcal{E}$, and for every duplicate-free node* n *of $\mathcal{E}$, time-window bound ϵ, and position j, the set $[\![\mathsf{n}]\!]^{\epsilon}_{\mathcal{E}}(j)$ can be enumerated with output-linear delay and without duplicates.*

Appendix B.1 gives the pseudocode of the enumeration algorithm, and shows its correctness.

5.2 Methods for managing the data structure

The evaluation algorithm will build the tECS $\mathcal{E}$ incrementally: it starts from the empty tECS and, whenever a new tuple arrives on the stream S, it modifies $\mathcal{E}$ to correctly represent the relevant open complex events. To ensure that we may enumerate $[\![\mathcal{A}]\!]^{\epsilon}_{j}(S)$ from $\mathcal{E}$ by use of Theorem 2, $\mathcal{E}$ will always be time-ordered, k-bounded for $k = 3$, and duplicate-free. We next discuss the operations for modifying a tECS $\mathcal{E}$ required by the evaluation algorithm.

It is important to remark that, in order to ensure that newly created nodes are 3-bounded, many of these operations expect their argument nodes to be *safe*. Here, a node is *safe* if it is a non-union node or if both odepth(n) = 1 and odepth(right(n)) ≤ 2. All of our operations themselves return safe nodes, as we will see.

Operations on tECS. We consider the following three operations:

$$\mathsf{b} \longleftarrow \text{new-bottom}(i) \qquad \mathsf{o} \longleftarrow \text{extend}(\mathsf{n}, j) \qquad \mathsf{u} \longleftarrow \text{union}(\mathsf{n}_1, \mathsf{n}_2)$$

where $i, j \in \mathbb{N}$, n, n_1 and n_2 are nodes in $\mathcal{E}$, and b, o, and u are the bottom, output, and union nodes, respectively, created by these methods. The first method, new-bottom(i) simply adds a new bottom node b labeled by i to $\mathcal{E}$. The second method, extend(n, j) adds a new output node o to $\mathcal{E}$ with pos(o) = j and next(o) = n. The third method, union($\mathsf{n}_1, \mathsf{n}_2$) returns a node u such that $[\![\mathsf{u}]\!]_{\mathcal{E}} = [\![\mathsf{n}_1]\!]_{\mathcal{E}} \cup [\![\mathsf{n}_2]\!]_{\mathcal{E}}$. This method requires a more detailed discussion.

Specifically, union requires that its inputs n_1 and n_2 are safe and that max(n_1) = max(n_2). Under these requirements, union($\mathsf{n}_1, \mathsf{n}_2$) operates as follows. If n_1 is non-union then a new union node u is created which is connected to n_1 and n_2 as shown in Figure 5(a). If n_2 is non-union, then u is created as shown in Figure 5(b). When n_1 and n_2 are both union nodes we distinguish two cases. If max(right(n_1)) $\geq$ max(right(n_2)), three new union nodes, u, u_1, and u_2 are added, and connected as shown in Figure 5(c). Finally, if max(right(n_1)) < max(right(n_2)), three new union nodes are added, but connected as we show in Figure 5(d). In all cases, $[\![\mathsf{u}]\!]_{\mathcal{E}} = [\![\mathsf{n}_1]\!]_{\mathcal{E}} \cup [\![\mathsf{n}_2]\!]_{\mathcal{E}}$ and the newly created union nodes are time-ordered. Furthermore, the reader is invited to check that, because n_1 and n_2 are safe, u is also safe, and the output-depth of the newly created union nodes is a most 3.

Because also the nodes created by new-bottom and extend are safe, time-ordered, and have output-depth at most 3, it follows that any tECS that is created using only these three methods is time-ordered and 3-bounded. Moreover, all of these methods output safe nodes and take constant time.

Union-lists and their operations. To incrementally maintain $\mathcal{E}$, the evaluation algorithm will also need to manipulate *union-lists*. A union-list is a non-empty sequence ul of safe nodes of the form ul = $\mathsf{n}_0, \mathsf{n}_1, \ldots, \mathsf{n}_k$ such that (1) n_0 is non-union, (2) max(n_0) $\geq$ max(n_i) and (3) max(n_j) > max(n_{j+1}), for every $i \leq k$ and $1 \leq j < k$. In other words, a union-list is a non-empty sequence of safe nodes sorted decreasingly by maximum-start.

We require three operations on union-lists, all of which take safe nodes as arguments. The first method, new-ulist(n), creates a new union-list containing the single non-union node n. The second method, insert(ul, n), mutates union-list ul = $\mathsf{n}_0, \ldots, \mathsf{n}_k$ in-place by inserting a safe node n such that max(n) $\leq$ max(n_0). Specifically, if there is $i > 0$ such that max(n_i) = max(n), then it replaces n_i in ul by the result of calling union(n_i, n). This hence also updates $\mathcal{E}$. Otherwise, we discern two cases. If max(n) = max(n_0), then n is inserted at position 1 in ul. Otherwise, n is inserted between n_i and n_{i+1} with $i > 0$ such that max(n_i) > max(n) > max(n_{i+1}). The last method, merge(ul), takes a union-list ul and returns a node u such that $[\![\mathsf{u}]\!]_{\mathcal{E}} = [\![\mathsf{n}_0]\!]_{\mathcal{E}} \cup \ldots \cup [\![\mathsf{n}_k]\!]_{\mathcal{E}}$. Specifically, if $k = 0$, then u = n_0. Otherwise, we add k union nodes u, $\mathsf{u}_1, \ldots, \mathsf{u}_{k-1}$ to $\mathcal{E}$, and connect them as shown in Figure 5 (e). It is important to observe that, because n_0 is a non-union node, odepth(u) ≤ 1. Moreover, because all n_i are safe, odepth(u_i) ≤ 2. As a result, u is safe. Furthermore, all of the new union nodes are time-ordered and are 3-bounded. This, combined with the properties of new-bottom, extend, and union described above implies that any tECS that is created using only these three methods plus merge is time-ordered and 3-bounded. Furthermore, all of these methods retrieve safe nodes, and their outputs are hence valid inputs to further calls. Finally, we remark that all methods on union-lists take time linear in the length of ul.

Hash tables. In order to incrementally maintain $\mathcal{E}$, the evaluation algorithm will also need to manipulate hash tables that map CEA states to union-lists of nodes. If T is such a hash table, then we write T[q] for the union-list associated to state q and T[q] $\leftarrow$ ul for inserting or updating it with a union-list ul. We use the method keys(T) to iterate through all the current states of T and write $q \in$ keys(T) for checking if q is already a key in T or not. For technical reasons, we also consider a method called ordered-keys(T) that iterates over keys of T *in the order in which they have been inserted into T*. If a key is inserted and then later is inserted again (i.e.,

Algorithm 1 Evaluation of an I/O-deterministic CEA $\mathcal{A} = (Q, \Delta, q_0, F)$ over a stream S given a time-bound ϵ.

```
procedure Evaluation($\mathcal{A}$, $S$, $\epsilon$)
    $j \leftarrow -1$
    $\mathrm{T} \leftarrow \emptyset$
    while $t \leftarrow \mathrm{yield}(S)$ do
        $j \leftarrow j + 1$
        $\mathrm{T}' \leftarrow \emptyset$
        $\mathtt{ul} \leftarrow \mathtt{new\text{-}ulist}(\mathtt{new\text{-}bottom}(j))$
        ExecTrans($q_0$, ul, $t$, $j$)
        for $p \in \mathrm{ordered\text{-}keys}(\mathrm{T})$ do
            ExecTrans($p$, $\mathrm{T}[p]$, $t$, $j$)
        $\mathrm{T} \leftarrow \mathrm{T}'$
        Output($j$, $\epsilon$)

procedure ExecTrans($p$, ul, $t$, $j$)
    $\mathtt{n} \leftarrow \mathtt{merge}(\mathtt{ul})$
    if $q \leftarrow \Delta(p, t, \bullet)$ then
        $\mathtt{n}' \leftarrow \mathtt{extend}(\mathtt{n}, j)$
        $\mathtt{ul}' \leftarrow \mathtt{new\text{-}ulist}(\mathtt{n}')$
        Add($q$, n′, ul′)
    if $q \leftarrow \Delta(p, t, \circ)$ then
        Add($q$, n, ul)
    return

procedure Add($q$, n, ul)
    if $q \in \mathrm{keys}(\mathrm{T}')$ then
        $\mathrm{T}'[q] \leftarrow \mathtt{insert}(\mathrm{T}'[q], \mathtt{n})$
    else
        $\mathrm{T}'[q] \leftarrow \mathtt{ul}$
    return

procedure Output($j$, $\epsilon$)
    for $p \in \mathrm{keys}(\mathrm{T})$ do
        if $p \in F$ then
            $\mathtt{n} \leftarrow \mathtt{merge}(\mathrm{T}[p])$
            Enumerate(n, $j$)
```

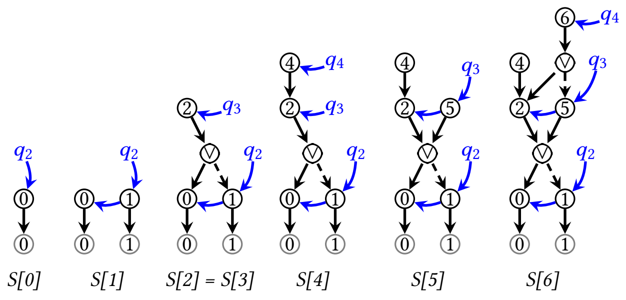


Figure 6: Illustration of Algorithm 1 on the CEA $\mathcal{A}$ and stream S of Figure 3.

an update), then it is the time of first insertion that counts for the iteration order. One can easily implement ordered-keys(T) by maintaining a traditional hash table together with a linked list that stores keys sorted in insertion order. Compatible with the RAM model of computation [6], we assume that hash table lookups and insertion take constant time, while iteration over their keys by means of keys(T) and ordered-keys(T) is with constant delay.

5.3 The evaluation algorithm

CORE's main evaluation algorithm is presented in Algorithm 1. It receives as input an I/O deterministic CEA $\mathcal{A} = (Q, \Delta, q_0, F)$, a stream S, and a time-bound ϵ. As already mentioned at the beginning of Section 5, its goal is to enumerate, at every position j in the stream, the set $[\![\mathcal{A}]\!]_j^\epsilon(S)$ of complex events produced by accepting runs terminating at j that satisfy the time-bound ε. It does so by maintaining (1) a tECS $\mathcal{E}$ to represents all open complex events up to the current position j and (2) the set of *active states* of $\mathcal{A}$. Here, a state $q \in Q$ is *active* at stream position j if there is some (not necessarily accepting) run of $\mathcal{A}$ that starts at position $i \le j$ which is in state q at position j. Specifically, in order to incrementally maintain the tECS, Algorithm 1 will link active states to the set of open complex events that they generate. Towards that goal, it uses a hash table T that maps active states of Q to a union-list of nodes.

We next explain how Algorithm 1 works. During our discussion, the reader may find it helpful to refer to Figure 6, which illustrates Algorithm 1 as it evaluates the CEA of Figure 3 over the stream S of Figure 3. Each subfigure depicts the state after processing $S[j]$. The tECS is denoted in black, while the hash table T that links the active states to union-lists is illustrated in blue. For each position, the set ordered-keys(T) will be ordered top down. (E.g., for $S[4]$ this will be q_4, q_3, q_2 while for $S[3]$ this will be q_3, q_2.)

Algorithm 1 consists of four procedures, of which Evaluation is the main one. It starts by initializing the current stream position j to -1 and the hash table T to empty (lines 2–3). Then, for every tuple in the stream, it executes the while loop in lines 4–12. Here, we assume that yield(S) returns the next unprocessed tuple t from S. For every such tuple, j is updated, and the hash table T′ is initialized to empty (lines 5-6). Intuitively, in lines 6–10, the hash table T will hold the states that are active at position $j - 1$ (plus corresponding union-lists) while T' will hold the states that are active at position j. In particular, T' is computed from T in lines 7–10. Specifically, lines 7–8 take into account that a new run may start at any position in the stream, and hence in particular at the current position j. For this purpose, the algorithm creates a new union-list starting at position j (line 7) and executes all transitions of initial state q_0 by calling ExecTrans (line 8), whose operation is explained below. Subsequently, lines 9–10 take into account that a state q is active at position j if there is a state p active at position $j - 1$ and a transition $p \xrightarrow{P/m} q$ of $\mathcal{A}$ with $t \models P$. As such, we iterate through all active states of T and execute all of their transitions (line 9-10). Once this is done, we swap the content of T with T' to prepare of the next iteration. We also call the Output method (lines 29-33) who is in charge of enumerating all complex events in $[\![\mathcal{A}]\!]_j^\epsilon(S)$, and whose operation is explained below.

The procedure ExecTrans is the workhorse of Algorithm 1. It receives an active state p, a union-list ul, the current tuple t, and the current position j. The union-list ul encodes all open complex events of runs that have reached p. ExecTrans first merges ul into a single node n (line 14). Then it executes the marking (line 15) and non-marking transitions (line 19) that can read t while in state p. Specifically, we write $q \leftarrow \Delta(p, t, m)$ to indicate that there is $p \xrightarrow{P/m} q$ in Δ with $t \models P$. Given that $\mathcal{A}$ is I/O-deterministic, there exists at most one such state q and, if there is none, we interpret $q \leftarrow \Delta(p, t, m)$ to be false. In lines 15–18, if there is a marking transition reaching q from p, then we extend all open complex events represented by n with the new position j (line 16) and add them to T′[q]. In lines 19–20, if there is a non-marking transition reaching q from p, we add n directly to T′[q] without extending it.

To add the open complex events to $\mathrm{T}'[q]$, we use the method Add (lines 22-27). This method checks whether it is the first time that we reach q on the j-th iteration or not. Specifically, if $q \in \mathrm{keys}(\mathrm{T}')$, then we have already reached q on the j-th iteration and therefore we insert n in the list $\mathrm{T}'[q]$ (lines 23-24). Instead, if it is the first time that we reach q on the j-iteration, then we initialize the q entry of T' with the union-list representation of n.

The Output procedure (lines 29-33) is in charge of enumerating all complex events in $[\![\mathcal{A}]\!]_j^{\epsilon}(S)$. Given that, when it is called, T contains all active states at position j, it suffices to iterate over $p \in \mathrm{keys}(\mathrm{T})$ and check whether p is a final state or not (lines 30-31). If p is final, then we merge the union-list at $\mathrm{T}[p]$ into a node n and call Enumerate(n, j), where Enumerate is the enumeration algorithm of Theorem 2.

Recall that by Theorem 2 if the tECS is k-bounded, time-ordered, and duplicate-free then the set $[\![\mathsf{n}]\!]_{\mathcal{E}}(j)$ can be enumerated with output-linear delay. Because Algorithm 1 builds $\mathcal{E}$ only through the methods of Section 5.2, we are guaranteed it is 3-bounded and time-ordered. Moreover, we can show that, because $\mathcal{A}$ is I/O-deterministic, $\mathcal{E}$ will also be duplicate-free. From this, we can derive the following correctness statement of Algorithm 1.

Theorem 3. *After the j-th iteration of Evaluation, the Output method enumerates the set $[\![\mathcal{A}]\!]_j^{\epsilon}(S)$ with output-linear delay.*

We note that the order in which we iterate over active states and execute transitions in lines 7–11 is important for the correctness of the algorithm (as explained in detail in the Appendix). In particular, because we first execute transitions of initial state q_0 and then process all states according to their insertion-order, we can prove that states are processed following a decreasing order of the max-start of active states. From this, we also derive that every call to $\mathsf{insert}(T'[q], n)$ in line 24 is legal: when it is called we have that $\max(T'[q]) \geq \max(n)$, as is required by the definition of insert.

Let us now analyze the update-time. When a new tuple arrives, lines 5–11 of Algorithm 5.3 update T, T', and $\mathcal{E}$ by means of the methods of Section 5.2. All of these either take constant time, or time linear in the size of the union list being manipulated. We can show (see the Appendix) that, for every position j, the length of every union list is bounded by the number of active states (i.e., the number of keys in T). Then, because in each invocation of lines 5–11 we iterate over all transitions in the worst case, and because executing a transition takes time proportional to the length of union-list, which is at most the number of states, we may conclude that the time for processing a new tuple is $O(|Q| \cdot |\Delta|)$. This is constant in data complexity.

5.4 Implementation aspects of CORE

We review here some implementation aspects of CORE that we did not cover by the algorithm or previous sections.

The system receives a CEQL query and a stream, reading it tuple by tuple. From the query, CORE collects all atomic predicates (e.g., `price > 100`) into a list, call it $P_1, \ldots, P_k$. For each tuple t of the stream, the system evaluates t over $P_1, \ldots, P_k$ by building a bit vector $\vec{v}_t$ of k entries such that $\vec{v}_t[i] = 1$ if, and only if, $t \models P_i$, for every $i \leq k$. Then, CORE uses $\vec{v}_t$ as the internal representation of t for optimizing the evaluation of complex predicates (e.g., conjunctions or disjunctions of atomic predicates) and for the determinization procedure (see below). Furthermore, CORE evaluates each predicate once, improving the performance over costly attributes (e.g., text).

As already mentioned, CORE compiles the CEQL query into a non-deterministic CEA $\mathcal{A}$ (Theorem 1). For the evaluation of $\mathcal{A}$ with Algorithm 1, CORE runs a determinization procedure *on-the-fly*: for a state p in the determinization of $\mathcal{A}$ and the bit vector $\vec{v}$, the states $q_\bullet := \Delta(p, \vec{v}, \bullet)$ and $q_\circ := \Delta(p, \vec{v}, \circ)$ are computed in linear time over $|\mathcal{A}|$. Moreover, we cache $q_\bullet$ and $q_\circ$ in main memory and use a fast-index to recover $\Delta(p, \vec{v}, \bullet)$ and $\Delta(p, \vec{v}, \circ)$ whenever is needed again. Although the determinization of $\mathcal{A}$ could be of exponential size in the worst case, this rarely happens in practice. Note that the determinization process depends on the selection strategy which can also computed on-the-fly by following the constructions in [31].

For reducing memory usage when dealing with time windows, the system manages the memory itself with the help of Java weak references. Nodes in the tECS data structure are weakly referenced, while the strong references are stored in a list, ordered by creation time. When the system goes too long without any outputs, it will remove the strong references from nodes that are now outside the time window, allowing Java's garbage collector to reclaim that memory without the need to modify the tECS data structure. Although this memory management could break the constant time update and output-linear delay, it takes constant *amortized* time and works well in practice (see Section 6).

For evaluating the PARTITION BY clause, CORE partitions the stream by the corresponding PARTITION BY attributes, running one instance of the algorithm for each partition. This process is done by hashing the corresponding attribute values, assigning them their own runs, or creating new ones if they don't exist.

We implemented CORE in Java. Its code is open-source and available at [1] under the GNU GPLv3 license.

6 Experiments

In this section, we compare CORE against three leading CER systems: SASE [52], Esper [2], and FlinkCEP [3]. These all provide a CER query language with features like pattern matching, windowing, and partition-by based correlation, whose semantics is comparable to CORE. We have also run experiments to compare against TESLA/TRex [22, 23] and Siddhi [4, 50]. Unfortunately, while both systems offer a CER query language, their event recognition features differ from CORE's and, as such, they are not optimized for the use cases that we aim to test here. For example, they do not have dedicated support for count-based time windows.[4] When we expressed simple sequence queries with time windows by means of other language features (e.g., by using inequality predicates) in their query languages, the performance was so deficient (i.e., a throughput of less than ten events per second), that we decided not to further report the results of TESLA/TRex and Siddhi.

We do not compare experimentally to various other proposals for evaluating CER queries, for the following reasons. First, we do not compare against systems without a publicly available implementation (e.g., ZStream [37], NextCEP [48], DistCED [40], SAP

[4]Although Siddhi has an operator for time windows, on can only use it in the presence of aggregation. As such, one cannot retrieve the complex events found, only aggregations thereof.

ESP [54][5]) or whose implementation was discontinued in the last decade (e.g., Cayuga [26]). Second, we do not compare against proposals that do not offer a CER query language or whose implementation does not allow running the queries considered here (e.g. CET [42], GRETA [44]). Similarly, we do not consider proposals whose system optimization focus is on additional CER features [33, 55, 56], such as load shedding [55] or interacting with external, non-stream data sources [56], for which the comparison would not be fair. Finally, we do not compare against data streams management systems like Spark and Flink whose SQL-based query languages do not natively support basic CER operators, like sequencing and iteration.

Setup. We compare CORE against SASE v.1.0, Esper v.8.7.0, and FlinkCEP v.1.12.2. We run our experiments on a server equipped with an 8-core AMD Ryzen 7 5800X processor running at 3.8GHz, 64GB of RAM, Windows 10 operating system, OpenJDK Runtime 17+35-2724, and the OpenJDK 64-Bit Server Virtual Machine build 17+35-2724. The Virtual Machine is restarted with 1GB of freshly allocated memory before each repetition of each experiment.

We compare systems with respect to their throughput and memory consumption. All reported numbers are averages taken over three repetitions of each experiment. We measure the throughput of each system, expressed as the number of events processed per second (e/s), as follows. To avoid measuring the data loading time of each system, we first load the input stream completely in main memory. We then start the timer, and count how many events each system can process within 30 seconds. We report the average throughput, expressed as the total number of processed events divided by 30 seconds. In all experiments, the input stream is larger than what can be processed within 30 seconds in any system.

Recognized complex events are logged to a file on disk. Because a single input event may fire many complex events, we only enumerate the first ten results. The only exception is FlinkCEP where, due to implementation reasons, we only print the first output. Note that, in principle, this hence favors FlinkCEP over the other systems since it needs to produce less outputs. For all systems, we use the consumption policy [24, 28] that forgets all events read so far when a complex event is found. We adopt this consumption policy for all systems because it is the only one supported by Esper and SASE.

We calculate memory usage by means of JVM system calls, after first calling the garbage collector. We measure the memory consumption every 10000 events, and report the average value. The experiments that measure memory consumption are run separately from the experiments that measure throughput.

For the sake of consistency, we have verified that all systems produced the same set of matched complex events. When this was not the case, we explicitly mention this difference below.

CORE's current implementation and SASE are single-core, sequential programs. To ensure fair comparison, all of the systems are therefore run in a single-core, sequential setup. Both Esper and FlinkCEP may exploit parallelism in a multi-core setup and support work in a distributed environment. While this may improve their performance, we stress that in many of the experiments below, CORE outperforms Esper and FlinkCEP by several orders of magnitude. As such, even if we assume that these systems have perfect linear scaling in the number of added processors (which is unlikely in practice), they would need several orders of magnitude more processors before meeting COREs throughput. Therefore, we do not consider a setup with parallelization.

To the best of our knowledge, today there does not exist a standard benchmark for complex event recognition. For this reason, we first experiment with a set of queries over synthetic data, and subsequently with a set of queries over real data, taken from the stock domain. All the experiments are reproducible. The data and scripts can be found in [1].

Sequence queries with output. We start by considering sequence queries, which have been used for benchmarking in CER before (see, for example, [26, 46, 52–54]). Specifically, we consider sequence patterns of length n occurring within time window T of the form:

```
SELECT * FROM RandomStream
WHERE A1 ; A2 ; ... ; An
WITHIN T
```

where A1 to An are n events of different types. RandomStream is a stream of $n+6$ possible event types, namely, A1 to An plus B1 to B6. The input stream is randomly generated, such that each event type appears with uniform probability $\frac{1}{n+6}$. We use B1 to B6 to introduce noise, in the form of irrelevant events. Although systems can easily discard these events as non-relevant to the query, they affect the probability that a complex event occurs in the time window.

In Figure 7, we display the results of evaluating sequence queries of length $n = 3, 5, 7, 9$ and a fixed time window size $\mathtt{T} = 100$ events. The top-left plot shows the throughput, in logarithmic scale, grouped by system. CORE's throughput is in the order of 10^6 e/s, while that of Esper and FlinkCEP is one to three orders of magnitude (OOM) lower. When $n = 3$ or $n = 5$, SASE's throughput is higher than CORE. However, as n grows, SASE's performance degrades exponentially. In contrast, CORE's throughput is stable, and degrades only linearly in n. For $n = 9$ CORE outperforms SASE by 6x, Esper by 33x, and FlinkCEP by 500x.

The top-right and bottom-left of Figure 7 separate the throughput in update throughput (updates per second) and enumeration throughput (outputs per second). On the one hand, the throughput of all systems improves when we remove the enumeration time. Nevertheless, the degrading behavior of SASE is more evident in this plot. The enumeration throughput of CORE is comparable (even slightly higher) to that of Esper and FlinkCEP, while being lower than that of SASE. Since CORE stores the output compactly in the tECS data structure (see Section 5), it needs to "decompress" each result before enumeration. Instead, SASE maintains each output explicitly, making the enumeration procedure more direct and faster. Although the output-linear delay approach forces CORE to pay this cost, it pays off regarding the overall throughput.

The bottom-right of Figure 7 shows the memory consumption for each query and system. The memory used by CORE is high (~200MB) but stable in n. Instead, the memory consumption of Esper, FlinkCEP[6] and SASE grows exponentially in n.

[5]SAP offers 30-day trial versions of its products, however, SAP ESP itself is no longer available from its website.

[6]The memory consumption of FlinkCEP drops for $n = 7, 9$. This is because in these cases FlinkCEP processes significantly less events compared to other systems, and compared to lower values of n.

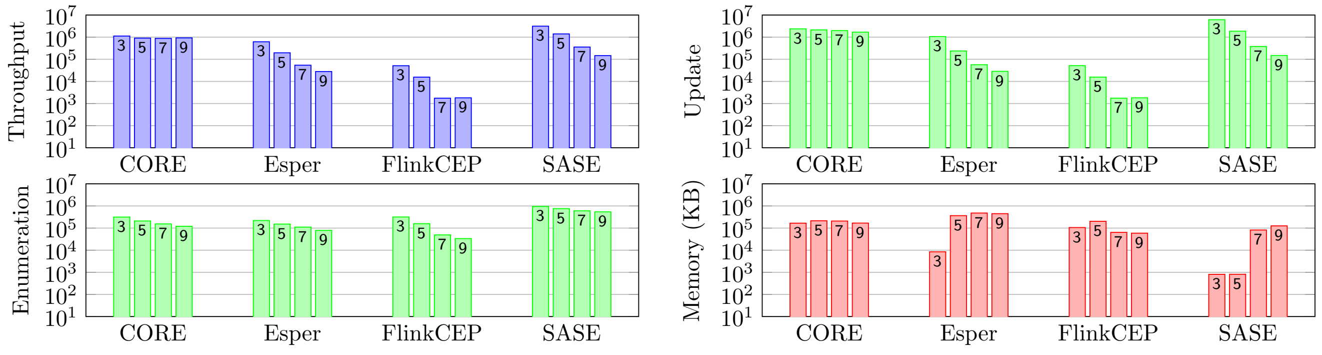


Figure 7: The throughput (top-left), update throughput (top-right), enumeration throughput (bottom-left), and memory consumption (bottom-right) of evaluating sequence queries of length 3, 5, 7 and 9 with windows length of 100 events.

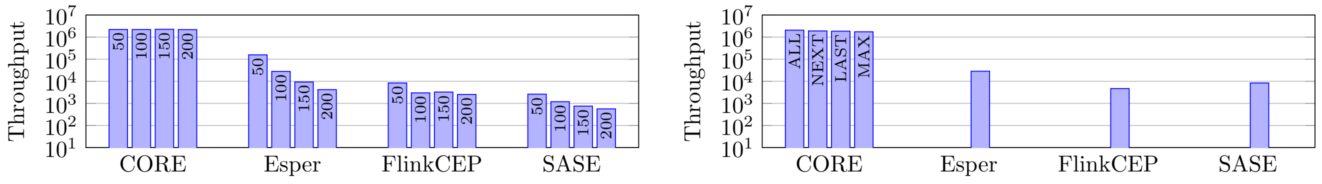


Figure 8: At the left, the throughput of sequence query A1;A2;A3 with time windows of 50, 100, 150, and 200 events, respectively, and without outputs. At the right, the throughput of the same sequence query with time window 100 under selection strategies.

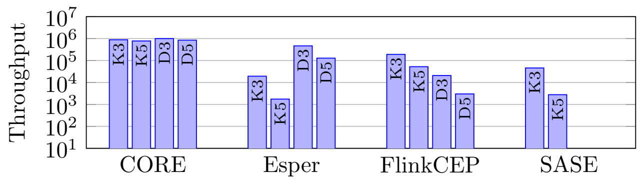
Sequence queries without output. The previous experiment considers sequence queries in the setting where the sought pattern occurs frequently in the randomly generated input stream. Because we adopt the consumption policy that forgets all input events seen so far once a matched complex event is found, this implies that, on average, the different systems need to remember only a limited set of partial answers. In practice, however, we may expect CER systems to look for unusual events, which means that the number of partial answers that need to be remembered may be significantly larger. Our next experiment captures this setting. We fix the sequence pattern A1; A2; A3 (i.e., $n = 3$, the case with the highest throughput for all systems in the previous experiment) and vary the window size from T = 50 to T = 200. We generate a random input stream with event types A1, A2, B1, . . ., B6, each with uniform probability. The event type A3 that fires a complex event is not included. We are therefore at the extreme point where systems look for an unusual complex event that never appears.

In Figure 8 (left) we show the throughput (log scale) for each window size, grouped by system. We see that CORE outperforms other systems by at least one OOM when T = 50 and by three OOM when T = 200 (more than 3800 times faster than SASE). This is in stark contrast with the frequent-match setting of the previous experiment, where SASE outperforms CORE. Note that we are still using relatively small time windows, of at most 200 events; in practice windows may be significantly larger. We also observe that the performance of other systems degrades exponentially in the time size T. Indeed, this is clear for Esper, where the throughput is more than 10^5 e/s when T = 50 but less than 10^4 e/s when T = 200. In contrast, CORE is stable.

Selection strategies. Many CER systems offer so-called selection strategies [24, 28, 52, 53]. A selection strategy can be seen as a heuristic for evaluating a query, where the system is asked to return only a specific subset of all matched complex event. Since this subset is often easier to recognize, it improves performance. In the next experiment, we compare all systems in the presence of selection strategies. Unfortunately, each system has its own algorithm for selection strategy and, thus, it is not possible to guarantee that everyone generates the same outputs. To solve this, we keep the approach of the previous experiment, namely, a sequence query A1;A2;A3 and T = 100, where we hide the last event A3. In this setting, there is no output, and we argue that all systems are hence performing the same task; namely, we test the case where the selection strategy found no result (i.e., an unusual event). Nevertheless, the systems are still free to adopt their performance-improving heuristics, consistent with their selection strategy.

The throughput (log scale) of the four systems with selection strategies is shown in Figure 8 (right). CORE implements four different selection strategies: ALL (no selection strategy), NEXT, LAST, and MAX (see [31] for the semantics of each selection strategy). All are implemented at the automata level, doing a sophisticated determinization procedure to filter outputs, namely, the algorithm is the same (see Section 5), but the underlying automaton is different. We used the default selection strategy for the other systems.

From Figure 8 we can conclude that the use of a selection strategy improves the performance of Esper, FlinkCEP, and SASE. The last system is the one that has a better improvement, going from 10^3 e/s (without selection strategy) to 10^4 e/s (with selection strategy). Despite this improvement, CORE is two OOM above other systems (i.e., 10^6 e/s) in all selection strategies. We can conclude then that the advantage of CORE is in the evaluation algorithm rather than in the use of selection strategies.



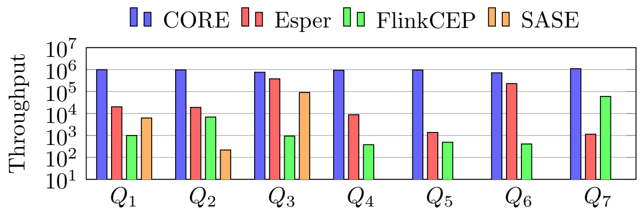


Figure 9: At the left: the throughput of evaluating queries with iteration ($K3$ and $K5$) and disjunction ($D3$ and $D5$) with a time window of 100 events. At the right: The throughput of evaluating queries Q_1 to Q_7 over stock market stream.

Other operators. We next consider queries with iteration and disjunction. For testing iteration, we consider the patterns

K3 := `A1;A2+;A3` and K5 := `A1;A2+;A3;A4+;A5`

which are the natural extensions of sequence queries with $n = 3$ and $n = 5$. Similarly, for disjunction, we use queries:

D3 := `A1;(A2 OR A2');A3` and
D5 := `A1;(A2 OR A2');A3;(A4 OR A4');A5`.

We set T = 100, and we randomly generate the input stream to contain query's event types plus B1 to B6 (i.e., noise) with uniform probability. Note that this experiment produces outputs similar to the first experiment ("sequence queries with output").

Figure 9 (left) shows the throughput (log-scale) per query, grouped by system. There is no result on D3 and D5 for SASE, given that it does not support queries with disjunction. We first observe that CORE outperforms all other systems by 2 or 3 OOM on queries with iteration and disjunction. Furthermore, CORE's performance is stable, maintaining a throughput of over one million events per second on all queries. The throughput of other systems, in contrast, degrades significantly as we increase the query length. All of this is consistent with the observations of the previous experiments. Second, when we compare Figure 9 with Figure 7 we see that adding iteration or disjunction affects the throughput of all systems except CORE, which remains stable around 10^6 e/s. For example, the throughput of Esper and SASE on the sequence query `A1;A2;A3` is around 10^6 e/s, but drops two OOM to 10^4 e/s when adding an iteration (K3). Note that this analysis does not consider the time windows size, which we know also negatively impacts the throughput of all systems except CORE.

Stock market data. We next compare systems on real-world data consisting of stock market events.[7] This dataset has already been used in the past to compare CER systems [42–45]. Similar to Example 1, the stream contains BUY and SELL events of stocks in a single market day, ordered by timestamp. Each event also has the stock name, volume and price (see Example 1). We test seven queries containing different features and measure the throughput of each system. Given space restrictions, we present the full CEQL definition of each query in the online appendix [1], and limit ourselves here to the the following simplified description.

Q_1	:=	`SELL ; BUY ; BUY ; SELL`
Q_2	:=	Q_1 + `FILTER`
Q_3	:=	Q_1 + `PARTITION BY`
Q_4	:=	`SELL ; (BUY OR SELL) ; (BUY OR SELL) ; SELL`
Q_5	:=	Q_4 + `FILTER`
Q_6	:=	Q_4 + `PARTITION BY`
Q_7	:=	`SELL ; (BUY OR SELL)+ ; SELL`

As we already mentioned, SASE does not support disjunction, and we hence omit Q_4–Q_7 for SASE. Queries Q_3 and Q_6 use the partition-by clause. Unfortunately, every system gave different outputs when we tried partition-by queries. Therefore, for Q_3 and Q_6 we cannot guarantee query equivalence for all systems. In all other cases, the results provided by each system are the same.

In Figure 9 (right) we show the throughput (log-scale), grouped per query. The figure confirms our observations of the previous experiments. Over real data, CORE's throughput is stable (i.e., 10^6 e/s) and approximately two OOM faster than other systems, which are not stable. Also, the presence of certain operators (e.g., filters or disjunction) may decrease performance, except for CORE. For example, disjunction decreases the performance of Esper (e.g., compare Q_1 and Q_4). Interestingly, adding filters reduces the performance of some systems like, for example, SASE on Q_1 and Q_2, or Esper on Q_4 and Q_5. CORE does not suffer from these problems due to its automaton-based evaluation algorithm. Finally, we can see that partition-by aids the performance of systems like Esper and SASE but slightly decreases the throughput of CORE and FlinkCEP (see Q_3 and Q_6). For CORE, we evaluate the partition-by clause by running several instances of the main algorithm, one for each partition, which diminishes the throughput. Nevertheless, CORE still outperforms the competition.

7 Conclusions and future work

We introduced CORE, a CER system whose evaluation algorithm guarantees constant time per event, followed by output-linear delay enumeration. To the best of our knowledge, CORE is the first system that supports both guarantees. We showed through experiments that the evaluation algorithm provides stable performance for CORE, which is not affected by the size of the stream, query, or time window. This property means a throughput up to three orders of magnitudes higher than leading systems in the area.

CORE provides a novel query evaluation approach; however, there is space for several improvements. A natural problem is to extend CEQL to allow time windows or partition-by operators inside the WHERE clause, which will increase the expressive power of CEQL. We currently do not know how to extend the evaluation algorithm for such queries while maintaining the performance

[7]https://davis.wpi.edu/datasets/Stock_Trace_Data/

guarantees. Other relevant features to include in CORE are aggregation, integration of non-event data sources, or the algorithm's parallelization, among others, which we leave as future work.

References

[1] [n.d.]. CORE Website. https://github.com/CORE-cer. Accessed on 2021-10-30.

[2] [n.d.]. Esper Enterprise Edition website. http://www.espertech.com/. Accessed on 2021-10-30.

[3] [n.d.]. FlinkCEP - Complex event processing for Flink. https://ci.apache.org/projects/flink/flink-docs-release-1.13/docs/libs/cep/. Accessed on 2021-10-30.

[4] [n.d.]. Siddhi: Cloud Native Stream Processor. https://siddhi.io/. Accessed on 2021-10-30.

[5] Jagrati Agrawal, Yanlei Diao, Daniel Gyllstrom, and Neil Immerman. 2008. Efficient pattern matching over event streams. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. Association for Computing Machinery, New York, NY, USA, 147–160.

[6] Alfred V Aho and John E Hopcroft. 1974. *The design and analysis of computer algorithms*. Pearson Education India.

[7] Elias Alevizos, Anastasios Skarlatidis, Alexander Artikis, and Georgios Paliouras. 2017. Probabilistic complex event recognition: A survey. *ACM Computing Surveys (CSUR)* 50, 5 (2017), 1–31.

[8] Antoine Amarilli, Pierre Bourhis, Louis Jachiet, and Stefan Mengel. 2017. A Circuit-Based Approach to Efficient Enumeration. In *ICALP 2017-44th International Colloquium on Automata, Languages, and Programming*. 1–15.

[9] Antoine Amarilli, Pierre Bourhis, Stefan Mengel, and Matthias Niewerth. 2019. Enumeration on trees with tractable combined complexity and efficient updates. In *Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. 89–103.

[10] Antoine Amarilli, Pierre Bourhis, Stefan Mengel, and Matthias Niewerth. 2021. Constant-delay enumeration for nondeterministic document spanners. *ACM Transactions on Database Systems (TODS)* 46, 1 (2021), 1–30.

[11] Darko Anicic, Paul Fodor, Sebastian Rudolph, Roland Stühmer, Nenad Stojanovic, and Rudi Studer. 2010. A rule-based language for complex event processing and reasoning. In *International Conference on Web Reasoning and Rule Systems*. Springer, 42–57.

[12] Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, and Matei Zaharia. 2018. Structured streaming: A declarative api for real-time applications in apache spark. In *Proceedings of the 2018 International Conference on Management of Data*. 601–613.

[13] Alexander Artikis, Nikos Katzouris, Ivo Correia, Chris Baber, Natan Morar, Inna Skarbovsky, Fabiana Fournier, and Georgios Paliouras. 2017. A prototype for credit card fraud management: Industry paper. In *Proceedings of the 11th ACM international conference on distributed and event-based systems*. 249–260.

[14] Alexander Artikis, Alessandro Margara, Martin Ugarte, Stijn Vansummeren, and Matthias Weidlich. 2017. Complex event recognition languages: Tutorial. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*. 7–10.

[15] Alexander Artikis, Marek Sergot, and Georgios Paliouras. 2014. An event calculus for event recognition. *IEEE Transactions on Knowledge and Data Engineering* 27, 4 (2014), 895–908.

[16] Alexander Artikis, Anastasios Skarlatidis, François Portet, and Georgios Paliouras. 2012. Logic-based event recognition. *The Knowledge Engineering Review* 27, 4 (2012), 469–506.

[17] Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. 2007. On acyclic conjunctive queries and constant delay enumeration. In *International Workshop on Computer Science Logic*. Springer, 208–222.

[18] Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. 2017. Answering conjunctive queries under updates. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. 303–318.

[19] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36, 4 (2015), 28–38.

[20] Nofar Carmeli and Markus Kröll. 2021. On the Enumeration Complexity of Unions of Conjunctive Queries. *ACM Transactions on Database Systems (TODS)* 46, 2 (2021), 1–41.

[21] Federico Chesani, Paola Mello, Marco Montali, and Paolo Torroni. 2010. A logic-based, reactive calculus of events. *Fundamenta Informaticae* 105, 1-2 (2010), 135–161.

[22] Gianpaolo Cugola and Alessandro Margara. 2010. TESLA: a formally defined event specification language. In *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*. 50–61.

[23] Gianpaolo Cugola and Alessandro Margara. 2012. Complex event processing with T-REX. *Journal of Systems and Software* 85, 8 (2012), 1709–1728.

[24] Gianpaolo Cugola and Alessandro Margara. 2012. Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys (CSUR)* 44, 3 (2012), 1–62.

[25] Alan Demers, Johannes Gehrke, Mingsheng Hong, Mirek Riedewald, and Walker White. 2005. *A general algebra and implementation for monitoring event streams*. Technical Report. Cornell University.

[26] Alan Demers, Johannes Gehrke, Mingsheng Hong, Mirek Riedewald, and Walker White. 2006. Towards expressive publish/subscribe systems. In *International Conference on Extending Database Technology*. Springer, 627–644.

[27] Fernando Florenzano, Cristian Riveros, Martín Ugarte, Stijn Vansummeren, and Domagoj Vrgoč. 2020. Efficient enumeration algorithms for regular document spanners. *ACM Transactions on Database Systems (TODS)* 45, 1 (2020), 1–42.

[28] Nikos Giatrakos, Elias Alevizos, Alexander Artikis, Antonios Deligiannakis, and Minos Garofalakis. 2020. Complex event recognition in the big data era: a survey. *The VLDB Journal* 29, 1 (2020), 313–352.

[29] Alejandro Grez, Cristian Riveros, and Martín Ugarte. 2019. A formal framework for complex event processing. In *22nd International Conference on Database Theory (ICDT 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 5:1–5:18.

[30] Alejandro Grez, Cristian Riveros, Martín Ugarte, and Stijn Vansummeren. 2020. On the expressiveness of languages for complex event recognition. In *23rd International Conference on Database Theory (ICDT 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 15:1–15:17.

[31] Alejandro Grez, Cristian Riveros, Martin Ugarte, and Stijn Vansummeren. 2021. A Formal Framework for Complex Event Recognition. *ACM Transactions on Database Systems (TODS)* (2021). https://doi.org/10.1145/3485463 To appear.

[32] Mikell P Groover. 2007. *Automation, production systems, and computer-integrated manufacturing*. Prentice Hall.

[33] Martin Hirzel. 2012. Partition and compose: Parallel complex event processing. In *DEBS*. 191–200.

[34] Muhammad Idris, Martín Ugarte, Stijn Vansummeren, Hannes Voigt, and Wolfgang Lehner. 2020. General dynamic Yannakakis: conjunctive queries with theta joins under updates. *The VLDB Journal* 29, 2 (2020), 619–653.

[35] Mo Liu, Elke Rundensteiner, Kara Greenfield, Chetan Gupta, Song Wang, Ismail Ari, and Abhay Mehta. 2011. E-cube: multi-dimensional event sequence analysis using hierarchical pattern query sharing. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. 889–900.

[36] Ruben Mayer, Muhammad Adnan Tariq, and Kurt Rothermel. 2017. Minimizing communication overhead in window-based parallel complex event processing. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*. 54–65.

[37] Yuan Mei and Samuel Madden. 2009. Zstream: a cost-based query processor for adaptively detecting composite events. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. 193–206.

[38] Biswanath Mukherjee, L Todd Heberlein, and Karl N Levitt. 1994. Network intrusion detection. *IEEE network* 8, 3 (1994), 26–41.

[39] Martín Muñoz and Cristian Riveros. 2022. Streaming query evaluation with constant delay enumeration over nested documents. In *Proceedings of the 25th International Conference on Database Theory (accepted). Also available in arXiv preprint arXiv:2010.06037*.

[40] Peter R Pietzuch, Brian Shand, and Jean Bacon. 2003. A framework for event composition in distributed systems. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer, 62–82.

[41] Manolis Pitsikalis, Alexander Artikis, Richard Dreo, Cyril Ray, Elena Camossi, and Anne-Laure Jousselme. 2019. Composite event recognition for maritime monitoring. In *Proceedings of the 13th ACM International Conference on Distributed and Event-based Systems*. 163–174.

[42] Olga Poppe, Chuan Lei, Salah Ahmed, and Elke A Rundensteiner. 2017. Complete event trend detection in high-rate event streams. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 109–124.

[43] Olga Poppe, Chuan Lei, Lei Ma, Allison Rozet, and Elke A Rundensteiner. 2021. 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*. 1452–1464.

[44] Olga Poppe, Chuan Lei, Elke A Rundensteiner, and David Maier. 2017. GRETA: graph-based real-time event trend aggregation. *Proceedings of the VLDB Endowment* 11, 1 (2017), 80–92.

[45] Olga Poppe, Chuan Lei, Elke A Rundensteiner, and David Maier. 2019. Event trend aggregation under rich event matching semantics. In *Proceedings of the 2019 International Conference on Management of Data*. 555–572.

[46] Medhabi Ray, Chuan Lei, and Elke A Rundensteiner. 2016. Scalable pattern sharing on event streams. In *Proceedings of the 2016 international conference on management of data*. 495–510.

[47] BS Sahay and Jayanthi Ranjan. 2008. Real time business intelligence in supply chain analytics. *Information Management & Computer Security* 16 (2008), 28–48.

[48] Nicholas Poul Schultz-Møller, Matteo Migliavacca, and Peter Pietzuch. 2009. Distributed complex event processing with query rewriting. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*. 1–12.

[49] Luc Segoufin. 2013. Enumerating with constant delay the answers to a query. In *Proceedings of the 16th International Conference on Database Theory*. 10–20.

[50] Sriskandarajah Suhothayan, Kasun Gajasinghe, Isuru Loku Narangoda, Subash Chaturanga, Srinath Perera, and Vishaka Nanayakkara. 2011. Siddhi: A second look at complex event processing architectures. In *Proceedings of the 2011 ACM workshop on Gateway computing environments*. 43–50.

[51] Moshe Y Vardi. 1982. The complexity of relational query languages. In *Proceedings of the fourteenth annual ACM symposium on Theory of computing*. 137–146.

[52] Eugene Wu, Yanlei Diao, and Shariq Rizvi. 2006. High-performance complex event processing over streams. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. 407–418.

[53] Haopeng Zhang, Yanlei Diao, and Neil Immerman. 2014. On complexity and optimization of expensive queries in complex event processing. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 217–228.

[54] Shuhao Zhang, Hoang Tam Vo, Daniel Dahlmeier, and Bingsheng He. 2017. Multi-query optimization for complex event processing in SAP ESP. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. IEEE, 1213–1224.

[55] Bo Zhao, Nguyen Quoc Viet Hung, and Matthias Weidlich. 2020. Load shedding for complex event processing: Input-based and state-based techniques. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 1093–1104.

[56] Bo Zhao, Han van der Aa, Thanh Tam Nguyen, Quoc Viet Hung Nguyen, and Matthias Weidlich. 2021. EIRES: Efficient Integration of Remote Data in Event Stream Processing. In *Proceedings of the 2021 International Conference on Management of Data*. 2128–2141.

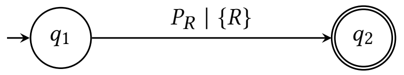


Figure 10: A VCEA for the atomic formula R. Here, $P_R = \{t \mid t(\text{type}) = R\}$.

A Proofs of Section 3

A.1 Proof of Theorem 1

Theorem 1. *For every CEL formula φ we can construct a CEA $\mathcal{A}$ of size linear in φ such that for every ϵ:*

$$[\![\varphi \texttt{ WITHIN } \epsilon]\!](S) \;=\; \{C \mid C \in [\![\mathcal{A}]\!](S) \wedge C(\text{end}) - C(\text{start}) \leq \epsilon\}.$$

Proof. This result follows almost directly from the proof for Theorem 6.2 in [31], which states that for every (unary) CEL formula φ there is an equivalent CEA $\mathcal{A}$. However, there are certain subtleties that need to be addressed so that this result translates into ours. For completeness, we present here the construction of [31], modified to fit our setting with intervals. However, it is worth noting that the only three differences in the construction are in the cases $\varphi = R$, $\varphi = \psi_1 \,;\, \psi_2$ and $\varphi = \psi+$, in order to handle the fact that now a run may start at an arbitrary position i in the stream.

In [31], they use an extended model of CEA, which replaces the $\bullet, \circ$ marks in the transitions with subsets of variables. Formally, a *valuation complex event automaton* (VCEA) is a tuple $\mathcal{A} = (Q, \Delta, I, F)$, where Q is the set of states, $I, F \subseteq Q$ are the set of initial and final states, respectivelly, and Δ is the transition relation $\Delta \subseteq Q \times \mathbf{P} \times 2^{\mathbf{X}} \times Q$. The definitions of run and accepting run remain the same as for CEA, i.e., a run is a sequence $\rho \;:=\; q_i \xrightarrow{P_i/L_i} q_{i+1} \xrightarrow{P_{i+1}/L_{i+1}} \ldots \xrightarrow{P_j/L_j} q_{j+1}$ such that $q_i \in I$ and for every $k \in [i, j]$ it holds that $q_k \xrightarrow{P_k/L_k} q_{k+1} \in \Delta$ and $t_k \models P_k$; and ρ is *accepting* if $q_{j+1} \in F$. Now ρ defines a valuation $V_\rho \;:=\; ([i, j], \mu)$, where $\mu(X) = \{k \mid X \in L_k\}$, and we define the valuation semantics of $\mathcal{A}$ over a stream S as $[\![\mathcal{A}]\!](S) := \{V_\rho \mid \rho$ is an accepting run of $\mathcal{A}$ over $S\}$. Now, we say that a CEL formula φ is equivalent to a VCEA if $[\![\varphi]\!](S) = [\![\mathcal{A}]\!](S)$ for every stream S.

Consider any CEL formula φ. We can construct a VCEA $\mathcal{A}_\varphi$ that is equivalent to φ by induction on φ, as follows.

- If $\varphi = R$, then $\mathcal{A}_\varphi$ is defined as depicted in Figure 10, i.e. $\mathcal{A}_\varphi = (\{q_1, q_2\}, \{(q_1, P_R, \{R\}, q_2)\}, \{q_1\}, \{q_2\})$, where P_R is the predicate containing all tuples with type R, namely, $P_R := \{t \mid t(\text{type}) = R\}$.
- If $\varphi = \psi$ AS X, then $\mathcal{A}_\varphi = (Q_\psi, \Delta_\varphi, I_\psi, F_\psi)$ where Δ_φ is the result of adding variable X to all marking transitions of Δ_ψ, i.e., $\Delta_\varphi = \{(p, P, L, q) \in \Delta_\psi \mid L = \emptyset\} \cup \{(p, P, L, q) \mid \exists L' \neq \emptyset$ such that $(p, P, L', q) \in \Delta_\psi \wedge L = L' \cup \{X\}\}$.
- If $\varphi = \psi$ `FILTER` $X[P]$ for some variable X and predicate P, then $\mathcal{A}_\varphi = (Q_\psi, \Delta_\varphi, I_\psi, F_\psi)$ where Δ_φ is defined as $\{(p, P', L, q) \in \Delta_\psi \mid X \notin L\} \cup \{(p, P \wedge P', L, q) \mid (p, P', L, q) \in \Delta_\psi \wedge X \in L\}$.
- If $\varphi = \psi_1$ `OR` ψ_2, then $\mathcal{A}_\varphi$ is the automata union between $\mathcal{A}_{\psi_1}$ and $\mathcal{A}_{\psi_2}$: $\mathcal{A}_\varphi = (Q_{\psi_1} \cup Q_{\psi_2}, \Delta_{\psi_1} \cup \Delta_{\psi_2}, I_{\psi_1} \cup I_{\psi_2}, F_{\psi_1} \cup F_{\psi_2})$. Here, we assume w.l.o.g. that $\mathcal{A}_{\psi_1}$ and $\mathcal{A}_{\psi_2}$ have disjoint sets of states.
- If $\varphi = \psi_1 \,;\, \psi_2$, then $\mathcal{A}_\varphi = (Q_{\psi_1} \cup Q_{\psi_2}, \Delta_\varphi, I_{\psi_1}, F_{\psi_2})$ where $\Delta_\varphi = \Delta_{\psi_1} \cup \Delta_{\psi_2} \cup \{(p, \text{TRUE}, \emptyset, p) \mid p \in I_{\psi_2}\} \cup \{(p, P, L, q) \mid q \in I_{\psi_2} \wedge \exists q' \in F_{\psi_1}.(p, P, L, q') \in \Delta_{\psi_1}\}$. Here, we assume w.l.o.g. that $\mathcal{A}_{\psi_1}$ and $\mathcal{A}_{\psi_2}$ have disjoint sets of states.
- If $\varphi = \psi+$, then $\mathcal{A}_\varphi = (Q_\psi \cup \{q\}, \Delta_\varphi, I_\psi, F_\psi)$ where q is a fresh state, $\Delta_\varphi = \Delta_\psi \cup \{(p, P, L, q) \mid \exists q' \in F_\psi.(p, P, L, q') \in \Delta_\psi\} \cup \{(q, P, L, p) \mid \exists q' \in I_\psi.(q', P, L, p) \in \Delta_\psi\}$.
- If $\varphi = \pi_L(\psi)$ for some $L \subseteq \mathbf{L}$, then $\mathcal{A}_\varphi = (Q_\psi, \Delta_\varphi, I_\psi, F_\psi)$ where Δ_φ is the result of intersecting the labels of each transition in Δ_ψ with L. Formally, that is $\Delta_\varphi = \{(p, P, L \cap L', q) \mid (p, P, L', q) \in \Delta_\psi\}$.

Note that in the VCEA model we replaced q_0 with a subset of states I. This is not a problem, since one can modify the final automaton to have only one initial state by adding a fresh state q_0 and replicating every transition coming from an initial state to go from q_0, i.e., adding transitions $\{(q_0, P, L, q) \mid \exists p \in I.(p, P, L, q) \in \Delta\}$. This also assures that q_0 has no incoming transitions. From now on, we assume that $\mathcal{A}_\varphi$ has only one initial state q_φ.

Now that we have a VCEA $\mathcal{A}_\varphi = (Q_\varphi, \Delta_\varphi, q_\varphi, F_\varphi)$ that is equivalent to φ, we can construct a CEA $\mathcal{A}$ by changing the sets in the transitions with $\bullet, \circ$ marks. More precisely, we define $\mathcal{A} = (Q_\varphi, \Delta, q_\varphi, F_\varphi)$ where $\Delta = \{(p, P, \bullet, q) \mid (p, P, L, q) \in \Delta_\varphi \wedge L \neq \emptyset\} \cup \{(p, P, \circ, q) \mid (p, P, L, q) \in \Delta_\varphi \wedge L = \emptyset\}$. One can verify that this CEA satisfies the theorem. □

B Proofs of Section 5

B.1 Proof of Theorem 2

Theorem 2. *Fix k. For every k-bounded and time-ordered tECS $\mathcal{E}$, and for every duplicate-free node n of $\mathcal{E}$, time-window bound ϵ, and position j, the set $[\![\mathsf{n}]\!]^{\epsilon}_{\mathcal{E}}(j)$ can be enumerated with output-linear delay and without duplicates.*

Proof. Consider a k-bounded and time-ordered tECD $\mathcal{E}$, a duplicate-free node n of $\mathcal{E}$, a time-window bound ϵ and a position j. We provide Algorithm 2 and show that: (1) it enumerates the set $[\![\mathsf{n}]\!]^{\epsilon}_{\mathcal{E}}(j)$, and (2) it does so with output-linear delay. To simplify presentation in what follows, we denote the sets of bottom, output and union nodes by N_B, N_O and N_U, respectively.

Algorithm 2 Enumeration of $[\![\mathsf{n}]\!]^{\epsilon}_{\mathcal{E}}(j)$.

```
procedure Enumerate($\mathcal{E}$, n, $\epsilon$, $j$)
    st ← new-stack()
    $\tau \leftarrow j - \epsilon$
    if max(n) $\geq \tau$ then
        push(st, (n, $\emptyset$))
    while (n′, $P$) ← pop(st) do
        while true do
            if n′ $\in N_B$ then
                output([pos(n′), $j$], $P$)
                break
            else if n′ $\in N_O$ then
                $P \leftarrow P \cup$ {pos(n′)}
                n′ ← next(n′)
            else if n′ $\in N_U$ then
                if max(right(n′)) $\geq \tau$ then
                    push(st, (right(n′), $P$))
                n′ ← left(n′)
```

Algorithm 2 uses a stack st with the typical stack operations: new-stack() to initialize an empty stack, push(st, e) to add an element e at the beginning of st, and pop(st) to get the first element in st. Moreover, when st is empty, $e \leftarrow$ pop(st) is interpreted as false. We assume that each of the previous operations can be performed in constant time.

Recall that $\mathcal{E}$ encodes a directed acyclic graph (DAG) $G_{\mathcal{E}} = (N, E)$ where N are the vertices, and edges goes from any union node u to left(u) and right(u), and from any output node o to next(o). For every node $\mathsf{n}' \in N$ reachable from n and a threshold $\tau \geq 0$, let paths$_{\geq \tau}(\mathsf{n}')$ be all paths of $G_{\mathcal{E}}$ that start at n' and end at some bottom node b with pos(b) $\geq \tau$. It is clear that there exists a bijection between $[\![\mathsf{n}']\!]^{\epsilon}_{\mathcal{E}}(j)$ and paths$_{\geq j-\epsilon}(\mathsf{n}')$, namely, for every complex event within a time window of size ϵ there exists exactly one path that reaches a bottom node b with pos(b) $\geq j - \epsilon$ (this is true because, since n is duplicate-free, so must be n'). Then, Algorithm 2 receives as input a tECD $\mathcal{E}$, a node n, a time-window bound ϵ and a position$j$, and traverses $G_{\mathcal{E}}$ in a DFS manner and following the left-to-right order, i.e., for every union node u, the paths of left(u) are traversed before the ones of right(u). Each iteration of the **while** of line 6 traverses a new path starting from the point it branches from the previous path. For this, the stack st is used to store the node and partial complex event of that branching point. Then, the **while** of line 7 traverses through the nodes of the next path, following the left direction whenever a union node is reached and adding the right node to the stack whenever needed. Moreover, by checking for every node n' its value max(n') before adding it to the stack, it makes sure of only going through paths in paths$_{\geq j-\epsilon}(\mathsf{n}')$.

A simpler recursive algorithm could compute the same results. However, if done in a naive way, the output-linear delay might not be assured because when going back from a path to compute the next one, the number of backtracking steps might be as long as the longest path of $G_{\mathcal{E}}$. To avoid this, Algorithm 2 uses the stack st which allows it to jump immediately to the node that continues the next path. Here we assume that storing P in the stack takes constant time. We can realize this assumption by representing P as a linked list of positions, where the list is ordered by the last element added and there is a link to the previous element. Then for storing P we can save in st a pointer to the last element added.

We first prove that Algorithm 2 enumerates the set $[\![\mathsf{n}]\!]^{\epsilon}_{\mathcal{E}}(j)$, provided that $\mathcal{E}$ is time-ordered and n a duplicate-free node.

Lemma 1. *Let $\mathcal{E}$ be a time-ordered tECD, n a duplicate-free node of $\mathcal{E}$ and ϵ a time-window. Then, Algorithm 2 enumerates $[\![\mathsf{n}]\!]^{\epsilon}_{\mathcal{E}}(j)$ without duplicates.*

Proof. For the rest of this proof fix a tECD $\mathcal{E}$ and a threshold $\tau = j - \epsilon$. For every node $\mathtt{m}$ of tECD, let $l_{\mathtt{m}}$ be the longest path from $\mathtt{m}$ to a bottom node. We say a pair $(\mathtt{m}, R)$ of a node and a set of positions is *visited* if at some iteration of the **while** in line 7, variables n' and P take the values $\mathtt{m}$ and R, respectively. Moreover, we say $(\mathtt{m}, R)$ is *left* when the pair $(\mathtt{m}', R')$ is popped from the stack, where $(\mathtt{m}', R')$ is the top of the stack when $(\mathtt{m}, R)$ was visited (or when the algorithm ends, if the stack was empty). The *enumeration period* of $(\mathtt{m}, R)$ is the period between it being visited and it being left. Intuitively, in the enumeration period of $(\mathtt{m}, R)$ is when the subtree rooted at $\mathtt{m}$ is enumerated.

Now, we prove that for every visited pair $(\mathtt{m}, R)$, the set $[\mathtt{m}](R, j) = \{([i, j], D \cup R) \mid (i, D) \in [\![\mathtt{m}]\!]_{\mathcal{E}} \wedge i \geq \tau\}$ is output in its enumeration period without duplicates. If this is the case, then it is clear that Lemma 1 holds, since, upon calling Enumerate($\mathcal{E}$, n, ϵ, j) the pair $(\mathsf{n}, \emptyset)$ is the first one visited, and the enumerated set corresponds to $[\mathsf{n}](\emptyset, j) = \{([i, j], D) \mid (i, D) \in [\![\mathsf{n}]\!]_{\mathcal{E}} \wedge i \geq \tau\} = [\![\mathsf{n}]\!]^{\epsilon}_{\mathcal{E}}(j)$.

We prove the above by strong induction over $l_{\mathtt{m}}$. The base case is when $l_{\mathtt{m}} = 0$, which means that $\mathtt{m}$ must be itself a bottom node. In that case, it enters in the **if** of line 8, outputs the corresponding complex event $([\mathsf{pos}(\mathtt{m}), j], R)$, clearly without duplicates. Note that $[\![\mathtt{m}]\!]_{\mathcal{E}} = \{\mathsf{pos}(\mathtt{m}), \emptyset\}$ and that $\max(\mathtt{m}) \geq \tau$, so $\mathsf{pos}(\mathtt{m}) \geq \tau$ and the statement holds.

Now for the inductive step, consider $l_{\mathtt{m}} > 0$, which means that $\mathtt{m}$ is either an output node or a union node. In the former case, after entering the **if** of line 11, P and $\mathtt{n}'$ are updated to $R \cup \{\mathsf{pos}(\mathtt{m})\}$ and $\mathsf{next}(\mathtt{m})$, respectively, and the new iteration of the **while** begins with those values. Then, since $l_{\mathsf{next}(\mathtt{m})} = l_{\mathtt{m}} - 1$, by induction hypothesis the set $[\mathsf{next}(\mathtt{m})](R \cup \{\mathsf{pos}(\mathtt{m})\}, j)$ is enumerated, but by definition $[\![\mathtt{m}]\!]_{\mathcal{E}} = \{(i, D' \cup \{\mathsf{pos}(\mathtt{m})\}) \mid (i, D') \in [\![\mathsf{next}(\mathtt{m})]\!]_{\mathcal{E}}\}$, so

$$[\mathsf{next}(\mathtt{m})](R \cup \{\mathsf{pos}(\mathtt{m})\}, j) \;=\; \{([i,j], D' \cup R \cup \{\mathsf{pos}(\mathtt{m})\}) \mid (i, D') \in [\![\mathsf{next}(\mathtt{m})]\!]_{\mathcal{E}} \wedge i \geq \tau\} \;=\; [\mathtt{m}](R, j).$$

Therefore, the enumerated set corresponds to $[\mathtt{m}](R, j)$. Also, by induction hypothesis this set is enumerated without duplicates and the statement holds.

If $\mathtt{m}$ is a union node, the **if** of line 14 is entered, which first pushes the pair $(\mathsf{right}(\mathtt{m}), R)$ if $\max(\mathsf{right}(\mathtt{m})) \geq \tau$ and then updates $\mathtt{n}'$ to $\mathsf{left}(\mathtt{m})$ before continuing with the next iteration. Since $l_{\mathsf{left}(\mathtt{m})}$ and $l_{\mathsf{right}(\mathtt{m})}$ are not greater than $l_{\mathtt{m}} - 1$, by induction hypothesis, both sets $[\mathsf{left}(\mathtt{m})](R, j)$, $[\mathsf{right}(\mathtt{m})](R, j)$ are enumerated without duplicates (note that after enumerating $[\mathsf{left}(\mathtt{m})](R, j)$, the pair $(\mathsf{right}(\mathtt{m}), R)$ is popped). Because $[\![\mathtt{u}]\!]_{\mathcal{E}} = [\![\mathsf{left}(\mathtt{u})]\!]_{\mathcal{E}} \uplus [\![\mathsf{right}(\mathtt{u})]\!]_{\mathcal{E}}$, then this enumerated set corresponds to $[\mathsf{left}(\mathtt{m})](R, j) \cup [\mathsf{right}(\mathtt{m})](R, j) = [\mathtt{m}](R, j)$. Also, since $\mathtt{n}'$ is duplicate-free, this set is enumerated without duplicates and the statement holds.

It has been proved that the set $[\![\mathtt{n}]\!]^{\epsilon}_{\mathcal{E}}(j)$ is enumerated without duplicates. □

Now that the correctness of the algorithm has been proved, we proceed to show that the enumeration is performed with output-linear delay.

Lemma 2. *Fix k. Let $\mathcal{E}$ be a k-bounded and time-ordered tECD, $\mathtt{n}$ a node of $\mathcal{E}$ and ϵ a time-window. Then, Algorithm 2 enumerates $[\![\mathtt{n}]\!]^{\epsilon}_{\mathcal{E}}(j)$ with output-linear delay.*

Proof. Fix $\mathcal{E}$ and τ. We showed that Algorithm 2 traverses all paths of $\mathsf{paths}_{\geq\tau}(\mathtt{n})$. Moreover, the order in which paths are traversed is completely determined by the order of the union nodes: for each union node $\mathtt{u}$, the paths to its left are traversed first, and then the ones to its right. Formally, for every node $\mathtt{n}'$ define the leftmost path from $\mathtt{n}'$ as $\pi_{\swarrow}(\mathtt{n}') := \mathtt{n}_0 \to \mathtt{n}_1 \to \ldots \to \mathtt{n}_l$ such that $\mathtt{n}_0 = \mathtt{n}'$ and, for every $i \leq l$:

- if $\mathtt{n}_i \in N_B$, then $i = l$,
- if $\mathtt{n}_i \in N_O$, then $\mathtt{n}_{i+1} = \mathsf{next}(\mathtt{n}_i)$, and
- if $\mathtt{n}_i \in N_U$, then $\mathtt{n}_{i+1} = \mathsf{left}(\mathtt{n}_i)$.

Consider a path $\pi := \mathtt{n}_0 \to \mathtt{n}_1 \to \ldots \to \mathtt{n}_l$, and let $j \leq l$ be the last position such that $\mathtt{n}_j$ is a union node, $\mathtt{n}_{j+1} = \mathsf{left}(\mathtt{n}_j)$ and $\max(\mathsf{right}(\mathtt{n}_j)) \geq \tau$. Then, let π^u be the path π up to position j, i.e., that stops at such union node.

Let $P = \{\pi_1, \pi_2, \ldots, \pi_m\}$ be the set of paths enumerated by Algorithm 2 in that order. Then, by analysing Algorithm 2, one can see that $\pi_1 = \pi_{\swarrow}(\mathtt{n})$ and, for every $i \leq m$, $\pi_i = \pi^u_{i-1} \cdot \pi_{\swarrow}(\mathsf{right}(\mathtt{u}))$, where $\mathtt{u}$ is the last node of π^u_{i-1}. In other words, it performs a greedy DFS from left to right: the first path to enumerate is $\pi_1 = \pi_{\swarrow}(\mathtt{n})$, then each π_i is the path of $\mathsf{paths}_{\geq\tau}(\mathtt{n})$ that branches from π_{i-1} to the right at the deepest level ($\mathtt{u}$) and from there follows the leftmost path. Moreover, to jump from π_{i-1} to π_i, the node popped by the stack is exactly $\mathtt{u}$, that is, the last node of π^u_{i-1}.

To show that the enumeration is done with output-linear delay, we study how long it takes between enumerating the complex events of π_{i-1} and π_i. Consider that π_{i-1} was just traversed and its complex event was output by line 9. Then, the **break** of line 10 is executed, breaking the **while** of line 7. Afterwards, either the stack is empty and the algorithm ends, or a pair $(\mathtt{m}, R)$ is popped from the stack, where $\mathtt{m}$ corresponds to the last node of π^u_{i-1}. From that point, it is straightforward to see that the number of iterations of the while of line 7 (each taking constant time) is equal to the number of nodes l in $\pi_{\swarrow}(\mathtt{m})$, so those nodes are traversed and the complex event of the path π_i is output. But, because $\mathcal{E}$ is k-bounded, then $l \leq k \cdot |O|$, where O is the complex event of π_i. Finally, the time taken is bounded by the size of the output, and the enumeration is performed with output-linear delay. □

By Lemmas 1 and 2, Theorem 2 immediately holds. □

B.2 Proof of Theorem 3

Theorem 3. *After the j-th iteration of Evaluation, the Output method enumerates the set $[\![\mathcal{A}]\!]^{\epsilon}_{j}(S)$ with output-linear delay.*

Proof. Fix a CEA $\mathcal{A} = (Q, \Delta, q_0, F)$ and a stream $S = t_0 t_2 \ldots$. Given a run of $\mathcal{A}$ over S of the form $\rho := q_i \xrightarrow{P_i/m_i} q_{i+1} \xrightarrow{P_{i+1}/m_{i+1}} \ldots \xrightarrow{P_j/m_j} q_{j+1}$, let $o(\rho) = (i, D)$ where $D = \{k \mid i \leq k \leq j \wedge m_k = \bullet\}$ be the open complex event associated to ρ. For any position j and a state $q \in Q$, let R^q_j be the set of runs that reach q after reading position j of the stream, that is, $R^q_j = \{\rho \mid \exists i.\, \rho \text{ is a run of } \mathcal{A} \text{ over } S \text{ from positions } i \text{ to } j\}$. Further, let $CE^q_j = \{o(\rho) \mid \rho \in R^q_j\}$ be the set of corresponding open complex events.

For a union-list $\mathtt{ul} = \mathtt{n}_0, \mathtt{n}_1, \ldots, \mathtt{n}_k$, of a tECD $\mathcal{E}$ define its set of open complex events as expected: $o(\mathtt{ul}) = [\![\mathtt{n}_0]\!]_{\mathcal{E}} \cup \ldots \cup [\![\mathtt{n}_k]\!]_{\mathcal{E}}$. Then, for a position j and a state $q \in Q$ let $ACE^q_j = o(\mathrm{T}[q])$ be the set of open complex events stored by $\mathrm{T}[q]$ after reading position j of the stream, i.e., after the $(j+1)$-th iteration of the **while** of line 4 of Algorithm 1.

Note that $[\![\mathcal{A}]\!]_j^{\epsilon}(S) = \{([i,j],D) \mid (i,D) \in \bigcup_{q\in F} CE_j^q \wedge j-i \leq \epsilon\}$. One can see that in Algorithm 1, if after the j-th iteration the underlying tECS $\mathcal{E}$ is time-ordered and k-bounded for some constant k and all nodes of $\mathcal{E}$ are duplicate-free, then by Theorem 2 procedure Output enumerates the set $\{([i,j],D) \mid (i,D) \in \bigcup_{q\in F} ACE_j^q \wedge j-i \leq \epsilon\}$ with output-linear delay. Therefore, if we prove that, for every j and q, (1) $CE_j^q = ACE_j^q$, (2) $\mathcal{E}$ is time-ordered and k-bounded and (3) all nodes in $\mathcal{E}$ are duplicate-free, then the theorem holds.

Before we start proving (1)-(3), let us first analyse procedure ExecTrans. At the beginning of iteration j, every ACE_j^q starts empty. Then, every time ExecTrans$(p, \mathtt{ul}, t, j)$ is called, it takes all open complex events in ACE_{j-1}^p and:

- for $q_\circ = \Delta(p,t,\circ)$, it makes $ACE_j^{q_\circ} \leftarrow ACE_j^{q_\circ} \cup ACE_{j-1}^p$; and
- for $q_\bullet = \Delta(p,t,\bullet)$, it makes $ACE_j^{q_\bullet} \leftarrow ACE_j^{q_\bullet} \cup (ACE_{j-1}^p \times \{j\})$, where $ACE_{j-1}^p \times \{j\} = \{(i, D\cup\{j\}) \mid (i,D) \in ACE_{j-1}^p\}$.

The two cases in Add just handle whether $ACE_j^{q_\bullet}$ or $ACE_j^{q_\circ}$ are empty at the moment or not.

We will prove (1) by simple induction over j. For the base case, consider the state before the first iteration, namely $j = -1$ and $\mathrm{T} = \emptyset$. It is easy to see that $CE_{-1}^q = ACE_{-1}^q = \emptyset$, so the property holds. Now for the inductive case, consider that $CE_j^q = ACE_j^q$. First, a new list with only a bottom node is created and stored in $\mathtt{ul}$ and then ExecTrans$(q_0, \mathtt{ul}, t, j)$ is called. This starts possibly two runs from the initial state q_0 with either a $\circ$ or $\bullet$-transition. Then, ExecTrans$(q_0, \mathrm{T}[p], t, j)$ is called for every state that is active before reading position j, i.e., for every p with $ACE_j^p \neq \emptyset$. Let us consider any $(i,D) \in ACE_{j+1}^q$. If it came from the first call to ExecTrans, then it means that $i = j$ and, either $D = \{j\}$ and there is a run (from $j+1$ to $j+1$) of the form $\rho = q_0 \xrightarrow{P/\bullet} q$, or $D = \emptyset$ and there is a run (from $j+1$ to $j+1$) $\rho = q_0 \xrightarrow{P/\circ} q$. Either way, the corresponding run assures that $(i,D) \in CE_{j+1}^q$. Now, if (i,D) came from one of the subsequent calls to ExecTrans, it means that there was some $(i,D') \in ACE_j^p$ such that either $D = D'$ and $q = \Delta(p, t_j, \circ)$, or $D = D' \cup \{j\}$ and $q = \Delta(p, t_j, \bullet)$. Call such transition e. By induction hypothesis, there is a run from i to j $\rho = q_i \xrightarrow{P_i/m_i} q_{i+1} \xrightarrow{P_{i+1}/m_{i+1}} \dots \xrightarrow{P_j/m_j} p$ with $o(\rho) = (i,D')$. Therefore, there is a run ρ' that extends ρ with e such that $o(\rho') = (i,D)$ and $\rho' \in R_{j+1}^q$, therefore $(i,D) \in ACE_{j+1}^q$. Then, $CE_{j+1}^q \subseteq ACE_{j+1}^q$. We can use the same analysis to prove that $ACE_{j+1}^q \subseteq CE_{j+1}^q$, thus $CE_{j+1}^q = ACE_{j+1}^q$ and the induction holds.

Now, we focus on proving (2). It has already been shown that every operation over tECS $\mathcal{E}$ used by Algorithm 1 preserves the time-ordered and k-boundedness for $k = 3$, provided that every time we call $\mathsf{union}(\mathsf{n}_1, \mathsf{n}_2)$, both n_1 and n_2 are safe nodes. In particular, since we only manage unions using union-lists, we only need to check that every time $\mathsf{new\text{-}ulist}(\mathsf{n})$ or $\mathsf{insert}(\mathtt{ul}, \mathsf{n})$ are called, the input node n is a safe one. This is easily verifiably by looking at lines 7, 17 and 24 of Algorithm 1 (for 17 and 24, recall that $\mathsf{merge}(\mathtt{ul})$ returns a safe node). Note that $\mathsf{merge}(\mathtt{ul})$ does not have any requirement, since its correct behaviour is ensured by using the other operations correctly. Therefore, at any moment in the evaluation, the underlying tECS $\mathcal{E}$ is time-ordered and 3-bounded. Note that $\mathsf{insert}(\mathtt{ul}, \mathsf{n})$ has another precondition: that $\max(\mathsf{n}_0) \geq \max(\mathsf{n})$, where n_0 is the first node of $\mathtt{ul}$. Since this does not relate directly with the correctness of the output of Algorithm 1, but with an overall correct behaviour of the algorithm, this property is discussed in the next section, along with another property that is needed to ensure constant update-time.

Next, we prove (3). By taking a look at Algorithm 1, one can see that the only ways it could add to $\mathcal{E}$ nodes with duplicates are with methods merge and insert, particularly in lines 14, 24 and 32. Fix a position j. Let All-N be the set all nodes in the union-list of some state at position j, i.e., All-N $= \{\mathsf{n} \mid \exists q. \exists i. T[q] = \mathsf{n}_0, \dots, \mathsf{n}_l \wedge \mathsf{n}_i = \mathsf{n}\}$, where $T[q]$ is its value at iteration j. If, for every two $\mathsf{n}_1, \mathsf{n}_2 \in$ All-N with $\mathsf{n}_1 \neq \mathsf{n}_2$ it holds that $[\![\mathsf{n}_i]\!]_{\mathcal{E}} \cap [\![\mathsf{n}_j]\!]_{\mathcal{E}} = \emptyset$, then all cases are solved and the algorithm creates no nodes with duplicates. One can see that each $[\![\mathsf{n}]\!]_{\mathcal{E}}$ actually stores the open complex events for a disjoint set of runs. Then, if there were some $\mathsf{n}_1, \mathsf{n}_2 \in$ All-N with $[\![\mathsf{n}_1]\!]_{\mathcal{E}} \cap [\![\mathsf{n}_2]\!]_{\mathcal{E}} \neq \emptyset$, it would mean that there are two different runs defining the same output, thus contradicting the fact that $\mathcal{A}$ is I/O-deterministic. Therefore, all nodes must define disjoint sets and Algorithm 1 only adds duplicate-free nodes to $\mathcal{E}$.

Finally, because (1)-(3) hold, also does the theorem. □

B.3 Special invariants of union-lists in Algorithm 1

In this section we discuss in more detail two invariants maintained by Algorithm 1 which are needed for its correct behaviour. Specifically, we prove that:

(1) every time a node n is added to some union-list $\mathtt{ul} = \mathsf{n}_0, \dots, \mathsf{n}_k$, it is the case that $\max(\mathsf{n}_0) \geq \max(\mathsf{n})$; and
(2) at any moment, the length of every union-list $\mathrm{T}[q]$ is smaller than the number of states of the automaton $\mathcal{A}$.

The former assures that the first node n_0 always remains a non-union node, while the latter is needed so that operations merge and insert take constant time (when $\mathcal{A}$ is fixed).

First, we focus on (1). This comes directly from the order in which we update the union-lists. As notation, for a non-empty union-list $\mathtt{ul} = \mathsf{n}_0, \dots, \mathsf{n}_k$, we write $\max(\mathtt{ul}) = \max(\mathsf{n}_0)$. Consider any iteration $j \geq 0$ and let $q_1, \dots, q_l$ be the order given by ordered-keys(T). One can prove that this order actually preserves the following property: $\max(\mathrm{T}[q_i]) \geq \max(\mathrm{T}[q_{i+1}])$ for every $i < l$. One can prove this by simple induction over j. As consequence, if ExecTrans$(q_i, \mathrm{T}[q_i], t, j)$ is called and we need to add some node n at some union-list $\mathtt{ul}$, it must be that $\mathtt{ul}$ was the result of some previous call ExecTrans$(q_k, \mathrm{T}[q_k], t, j)$ with $k < i$, and so $\max(\mathtt{ul}) \geq \max(\mathrm{T}[q_k]) \geq \max(\mathsf{n})$.

Now we prove (2). First, we show a property about the possible maximum-start values at any iteration of the algorithm. For $j \geq 0$, we define the set of maximum-starts at iteration j as $M_j = \{\max(\mathrm{T}[q]) \mid q \in \mathrm{keys}(\mathrm{T})\}$. The property we use is that the set M_j is closely restricted by the set M_{j-1} of the previous iteration. Specifically, one can see that M_j must always be a subset of M_{j-1}, with the possibility of adding the maximum-start j if a new run begins at that iteration, i.e., $M_j \subseteq (M_{j-1} \cup \{j\})$. This is because the maximum-start of every union-list $\mathrm{T}[q]$ at some iteration came either from extending the already existing runs of the previous iteration, therefore maintaining its maximum-start, or from starting a new run with maximum-start j.

Consider any union-list $\mathrm{T}[q]$ created at iteration j. From (1), we know that we will only add to $\mathrm{T}[q]$ nodes n with $\max(\mathsf{n}) \leq \max(\mathrm{T}[q])$. Moreover, it is always the case that $\max(\mathsf{n}) \in M_j$. This is clear for iteration j, but it actually holds for the following ones. Indeed, $\mathrm{T}[q]$ could actually receive insertions in some future iteration, call it k. However, since $\max(\mathrm{T}[q])$ does not change and the new maximum-start values in M_k are higher than $j \geq \max(\mathrm{T}[q])$, the actual maximum-start values that could be added to $\mathrm{T}[q]$ are still in $M_j \cap M_k \subseteq M_j$. Therefore, the size of $\mathrm{T}[q]$ is bounded by $|M_j| \leq |Q|$.

C Stock market queries

The specification of the stock market queries Q_1 to Q_7 is the following:

```
Q1 = SELECT * FROM S
     WHERE (SELL as msft; BUY as oracle; BUY as csco; SELL as amat)
     FILTER msft[name = 'MSFT'] AND oracle[name = 'ORCL'] AND
            csco[name = 'CSCO'] AND amat[name = 'AMAT']
     WITHIN 30000 [stock_time]

Q2 = SELECT * FROM S
     WHERE (SELL as msft; BUY as oracle; BUY as csco; SELL as amat)
     FILTER msft[name = 'MSFT'] AND msft[price > 26.0] AND
            oracle[name = 'ORCL'] AND oracle[price > 11.14] AND
            csco[name = 'CSCO'] AND amat[name = 'AMAT'] AND amat[price >= 18.92]
     WITHIN 30000 [stock_time]

Q3 = SELECT * FROM S
     WHERE (SELL as msft; BUY as oracle; BUY as csco; SELL as amat)
     FILTER msft[name = 'MSFT'] AND oracle[name = 'ORCL'] AND
            csco[name = 'CSCO'] AND amat[name = 'AMAT']
     PARTITION BY [volume]
     WITHIN 30000 [stock_time]
     CONSUME BY ANY

Q4 = SELECT * FROM S
     WHERE (SELL as msft; (BUY OR SELL) as oracle; (BUY OR SELL) as csco; SELL as amat)
     FILTER msft[name = 'MSFT'] AND oracle[name = 'ORCL'] AND
            csco[name = 'CSCO'] AND amat[name = 'AMAT']
     WITHIN 30000 [stock_time]

Q5 = SELECT * FROM S
     WHERE (SELL as msft; (BUY OR SELL) as oracle; (BUY OR SELL) as csco; SELL as amat)
     FILTER msft[name = 'MSFT'] AND msft[price > 26.0] AND
            oracle[name = 'ORCL'] AND oracle[price > 11.14] AND
            csco[name = 'CSCO'] AND amat[name = 'AMAT'] AND amat[price >= 18.92]
     WITHIN 30000 [stock_time]

Q6 = SELECT * FROM S
     WHERE (SELL as msft; (BUY OR SELL) as oracle; (BUY OR SELL) as csco; SELL as amat)
     FILTER msft[name = 'MSFT'] AND oracle[name = 'ORCL'] AND
            csco[name = 'CSCO'] AND amat[name = 'AMAT']
     PARTITION BY [volume]
     WITHIN 30000 [stock_time]
     CONSUME BY ANY
```

```
Q7 = SELECT * FROM S
     WHERE (SELL as msft; (BUY OR SELL)+ as qqq; SELL as amat)
     FILTER msft[name = 'MSFT'] AND qqq[name = 'QQQ'] and
            qqq[volume=4000] AND amat[name = 'AMAT']
     WITHIN 30000 [stock_time]
```

Next, we briefly explain each query. Q_1 is a sequence query looking for a sequence `SELL;BUY;BUY;SELL` of four major tech companies. Q_2 restricts Q_1 by filtering the price of each stock over a certain threshold (similar than in Example 1). Instead, Q_3 is a restricted version Q_1 with a partition-by clause over the stock volume. Q_4 is for testing disjunction by allowing Q_1 to test for `BUY OR SELL` in the two middle events. Q_5 and Q_6 are the analogs of Q_2 and Q_3, extending now Q_4 with filters and partition-by, respectively. Indeed, Q_5 contains the same features as Example 1. Finally, Q_7 combines disjunction and iteration, searching for a pattern of the form `SELL;(BUY OR SELL)+;SELL`.

For all queries, we used a time windows of 30 seconds (e.g., 30,000 milliseconds) over the `stock_time` attribute. We measure the throughput of the stock market stream, and it gives 4,803 e/s. Thus, using a time window of 30 seconds, we will have approximately 100 active events in the window, which is comparable with the experiments that we performed over synthetic data.