CORE: a Complex Event Recognition Engine

Peer-reviewed author version

Bucchi, Marco; Grez, Alejandro; Quintana, Andrés; Riveros, Cristian & VANSUMMEREN, Stijn (2022) CORE: a Complex Event Recognition Engine. In: Proceedings of the VLDB Endowment, 15 (9) , p. 1951 -1964.

DOI: 10.14778/3538598.3538615
Handle: http://hdl.handle.net/1942/39425

CORE: a Complex Event Recognition Engine

Marco Bucchi
PUC Chile
mabucchi@uc.cl

Alejandro Grez
PUC Chile and IMFD Chile
ajgrez@uc.cl

Andrés Quintana
PUC Chile and IMFD Chile
afquintana@uc.cl

Cristian Riveros
PUC Chile and IMFD Chile
cristian.riveros@uc.cl

Stijn Vansummeren
UHasselt, Data Science Institute
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 automaton-based 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 state-of-the-art CER systems on real-world data. We show that (1) CORE's performance is stable with respect to both query and time window size, and (2) CORE outperforms the other systems by up to five orders of magnitude on different workloads.

PVLDB Reference Format:
Marco Bucchi, Alejandro Grez, Andrés Quintana, Cristian Riveros, and Stijn Vansummeren. CORE: a Complex Event Recognition Engine. PVLDB, 15(9): 1951 - 1964, 2022.
doi:10.14778/3538598.3538615

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/CORE-cer.

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 [43], network intrusion detection [41], industrial control systems [34] and real-time analytics [49]. 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 [10, 26, 30].

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 [8, 25, 40, 54, 56, 57]. Indeed, conceptually, evaluating a CER query requires maintaining or recomputing a set of partial matches, so that when a new event arrives all partial matches that—together with the newly arrived event–now form a complete answer can be found. Unfortunately, even for simple CER patterns, the set of partial matches quickly becomes polynomial in the number N of previously processed events (or, when time windows are used, the number of events in the current window). Even worse, under the so-called skip-till-any-match selection strategy [8], queries that include the iteration operator may have sets of partial matches that grow *exponentially* in N [8]. As a result, the arrival of each new event requires a computation that is super-linear in N, which is incompatible with the small latency requirement.

In recognition of the computational challenge of CER query evaluation, a plethora of research has proposed innovative evaluation methods [16, 26, 30]. These methods range from proposing diverse execution models [18, 28, 40, 53], including cost-based database-style query optimizations to trade-off between materialization and lazy computation [36, 37, 40]; to focusing on specific query fragments (e.g., event selection policies [8]) that somewhat limit the super-linear partial match explosion; to using load shedding [56] to obtain low latency at the expense of potentially missing matches; and to employing distributed computation [25, 39]. All of these still suffer, however, from a processing overhead per event 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 5. Unfortunately, for applications such as maritime monitoring [43], network intrusion [41] and fraud detection [15], long time windows are necessary and a solution based on new principles hence seems desirable.

In recent work [31, 33], a subset of the authors have proposed a theoretical algorithm for evaluating CER queries that circumvents the super-linear partial match problem in theory: 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

Proceedings of the VLDB Endowment, Vol. 15, No. 9 ISSN 2150-8097.
doi:10.14778/3538598.3538615

algorithm needs to at least inspect every input event and list the query answers.

Briefly, the evaluation algorithm consists of translating a CER query into a particular kind of automaton model, called a Complex Event Automaton (CEA). Given a CEA and an input event stream, the algorithm simulates the CEA on the stream, and records all CEA runs by means of a graph data structure. Interestingly, this graph succinctly encodes all the partial and complete matches. When a new event arrives, the graph can be updated in constant time to represent the new (partial) runs. Once a final state of the automaton is reached, a traversal over the graph allows to enumerate the complex events with output-linear delay.

To date, this approach to CER query evaluation has only been the subject of theoretical investigation. Because it is the only known algorithm that provably circumvents the super-linear partial match problem, however, the question is whether it can serve as the basis of a practical CER system. In this paper, we answer this question affirmatively by presenting CORE, a novel COmplex event Recognition Engine based on the principles of [31–33].

A key limitation of [31, 33] that we need to resolve in designing CORE is that time windows are not supported in [31, 33], neither semantically at the query language level, nor at the evaluation algorithm level. Indeed, the algorithm records *all* runs, no matter how long ago the run occurred in the stream and no matter whether the run still occurs in the current time window. As such, it also does not prune the graph to clear space for further processing, which quickly becomes a memory bottleneck. Another limitation is that the algorithm cannot deal with simple equijoins, such as requiring all matching events to have identical values in a particular attribute. Such equijoins are typically expressed by means of the so-called *partition-by* operator [30]. We lift both limitations in CORE.

Technically, to ensure that CER queries with time windows can still be processed in constant update per event and output-linear delay enumeration we need to be able to represent CEA runs in a *ranked order fashion*: during enumeration, runs that start later in the stream must be traversed before runs that start earlier, so that once a run does not satisfy a time window restriction all runs that succeed it in the ranked order will not satisfy the time window either, and can therefore be pruned. The challenge is to encode this ranking in the graph representation of runs while ensuring that we continue to be able to update it in constant time per new event. This requires a completely new kind of graph data structure for encoding runs, and a correspondingly new CEA evaluation algorithm. This new algorithm is compatible with the partition-by operator, as we will see. We stress that, by this new algorithm, the runtime of CORE's complexity per event is independent of the number of partial matches, as well as the length of the time window being used. It therefore supports long time windows by design.

Contributions. Our contributions are as follows.

(1) To be precise about the query language features supported by CORE's evaluation algorithm, we formally introduce CEQL, a functional query language that focuses on *recognizing* complex events. It supports common event recognition operators including sequencing, disjunction, filtering, iteration, and projection [31–33], extended with partition-by, and time-windows. Other features considered in the literature that focus on *processing* of complex events, such as aggregation [46, 47], integration of non-event data sources [57], and parallel or distributed [25, 42, 50] execution are currently not supported by CEQL, and left for future work.

(2) We present an evaluation algorithm for CEQL that is based on entirely new evaluation algorithm for CEA that deals with time windows and partitioning, thereby lifting the limitations of [31–33].

(3) We show that the algorithm is practical. We implement it inside of CORE, and experimentally compare CORE against state-of-the-art CER engines on real-world data. Our experiments show that CORE's performance is stable: the throughput is not affected by the size of the query or size of the time window. Furthermore, CORE outperforms existing systems by one to five orders of magnitude in throughput on different query workloads.

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. We present the CEA computation model in Section 3. The algorithm and its data structures are described in Section 4, which also discusses implementation aspects of CORE. We dedicate Section 5 to experiments and conclude in Section 6.

Because of space limitations, certain details, formal statements and proofs are omitted. A full version of this paper, whose Appendix contains those items, is available online [21].

Further Related Work. CER systems are usually divided into three approaches: automata-based, tree-based, and logic-based, with some systems (e.g., [4, 24, 56]) being hybrids. We refer to recent surveys [10, 16, 26, 30] of the field for in-depth discussion of these classes of systems. CORE falls within the class of automata-based systems [8, 24, 27, 28, 37, 42, 44, 46, 47, 50, 53, 54, 57]. These systems use automata as their underlying execution model. As already mentioned, 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.

Tree- and logic-based systems [4, 14, 17, 23, 36, 38, 40] typically evaluate queries by constructing and evaluating a tree of CER operators, much like relational database systems evaluate relational algebra queries. Cost models can be used to identify efficient trees, and stream characteristics monitored to re-optimize trees during processing when necessary [36]. 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 [19, 20, 20, 22, 35, 51], 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 [11–13, 29]. Those works, however, do not consider enumeration with time window constraints as we do here.

```
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 (BUY or SELL) as s;
      (BUY or SELL) as b
PARTITION BY [name], [volume]
WITHIN 1 minute
```

(Q_2)

```
SELECT MAX * FROM Stock
WHERE (BUY OR SELL) as l; (BUY OR SELL)+ as m; (BUY OR SELL) as h
FILTER l[price < 100] AND m[price >= 100]
   AND m[price <= 2000] AND h[price > 2000]
PARTITION BY [name]
```

(Q_3)

Figure 1: CEQL queries on a Stock stream.

2 CEQL SYNTAX AND SEMANTICS

CORE's query language is similar in spirit to existing languages for expressing CER queries (see, e.g., [10, 26, 30] for a language survey). In particular, CORE shares with these languages a common set of operators for expressing CER patterns, including, for example sequencing, disjunction, iteration (a.k.a. Kleene closure), and filtering, among others [10, 30]. While hence conceptually similar, it is important to note that existing system implementations and their evaluation algorithms differ in (1) the exact set of operators supported, (2) how these operators can be composed, and (3) even in the semantics ascribed to operators. Given these sometimes subtle (semantic) differences, we wish to be unambiguous about the class of queries supported by CORE's evaluation algorithm. In this section, we therefore formally define the syntax and semantics of CEQL, the query language that is fully implemented by CORE.

CEQL is based on *Complex Event Logic* (CEL for short)—a formal logic that is built from the above-mentioned common operators without restrictions on composability, and whose expressiveness and complexity have been studied in [31–33]. CEQL extends CEL by adding support for time windows as well as the partition-by event correlation operator. We first introduce CEQL by means of examples, and then proceed with the formal syntax and semantics.

2.1 CEQL by Example

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. 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 in the stream. This can be any unary Complex Event Logic (CEL) expression [33]. 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. 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 `BUY` or `SELL` 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`) and iteration (also known as Kleene closure, denoted `+`). These may be freely nested in the `WHERE` clause. Query Q_2 illustrates the use of disjunction. Query Q_3 illustrates the use of iteration. In Q_3, 100 and 2000 are two values representing a lower and upper limit price, respectively. Q_3 looks for an upward trend: a sequence of `BUY` or `SELL` events pertaining to the same stock symbol where the sale price is initially below 100 (captured by the `l` variable), then between 100 and 2000 (captured by `m`), then above 2000 (`h`). Importantly, because of the Kleene closure iteration operator, variable `m` captures all sales of the stock in the [100, 2000] price range in such a trend. The `MAX` operator in the `SELECT` clause is an example of a selection strategy [8, 30, 33]: it ensures that within a trend `m` is bound to a maximal sequence of events in the [100, 2000] price range. If this policy were not specified, CEQL would adopt the skip-till-any-match policy [8, 30] by default, which also returns complex events with `m` containing only subsets of this maximal sequences.

2.2 CEQL Syntax and Semantics

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 $t(a) = \texttt{NULL}$.

A *stream* is a possibly infinite sequence $S = t_0 t_1 t_2 \ldots$ 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 $\{i, \ldots, j\}$. Intuitively, given a stream $S = t_0 t_1 \ldots$ the interval $[i, j]$ of C represents the subsequence $t_i t_{i+1} \ldots t_j$

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 $\{i, \ldots, j\}$ 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 $\{i, \ldots, j\}$ assigned to $X \in \mathbf{X}$ 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) [33], 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

[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_1 AND 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).

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

Figure 2: The semantics of a CEL formulas.

CEL[2], which is inductively defined in Table 2. Concretely, given a stream $S = t_0 t_1 t_2 \ldots$ (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 a selection strategy, then a CEL projection is applied, followed by removing certain valuations from the set. We refer the interested reader to [33] 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

[2] Note that the semantics used in this paper is an extension of the semantics of CEL in [33] since we also consider the time interval as part of the complex event.

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.

3 QUERY COMPILATION

At the heart of evaluating a CEQL query lies the problem of evaluating the query's SELECT-WHERE-WITHIN clauses on either a single stream, or multiple different substreams thereof (for a query with PARTITION BY). In this section and the next, we discuss how to do so efficiently, focusing on evaluation over a single stream. To this end, let Q be a CEQL query without PARTITION BY and with only one stream mentioned in the FROM clause.

In CORE we first compile the SELECT-WHERE part of Q into a *Complex Event Automaton (CEA for short)* [31, 33], 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 $[\![Q]\!](S)$. This evaluation algorithm is described in Section 4. 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 $p \xrightarrow{\sigma} q$ 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* ($\bullet$) or *unmarking* ($\circ$). 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 1. *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*

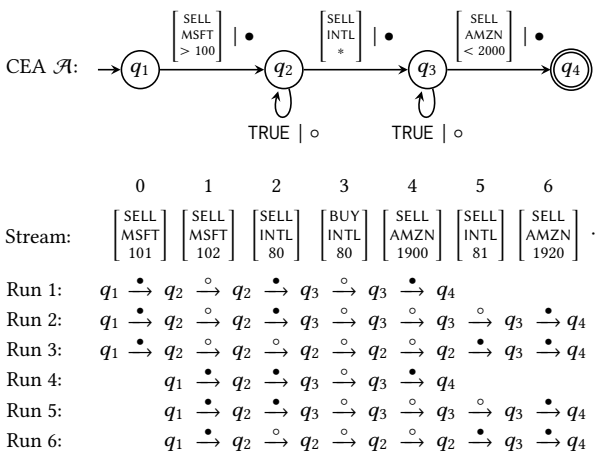

Figure 3: A CEA representing Q_1 from Figure 1 and some of its runs on an example stream.

is a sequence $\rho := q_i \xrightarrow{P_i/m_i} q_{i+1} \xrightarrow{P_{i+1}/m_{i+1}} \cdots \xrightarrow{P_j/m_j} q_{j+1}$ 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 $C_\rho := ([i, j], \{k \mid i \leq k \leq j \wedge m_k = \bullet\})$. 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 [31, 33] 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 4. 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 [31, 33]. 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:*

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

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

[3] We remark that any selection policy mentioned in the SELECT clause can also be expressed using CEA, see [31, 33].

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 [31, 33], 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 4.4 we discuss the internal implementation of CORE and how this exponential factor impacts system performance.

4 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

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

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 [52]) 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* [33, 51]. This means that the time required to print the first complex event of $[\![\mathcal{A}]\!]^{\epsilon}_{j}(S)$ 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 4.1, and operations on it in Section 4.2. The evaluation algorithm is given in Section 4.3 and aspects of its implementation in Section 4.4.

4.1 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

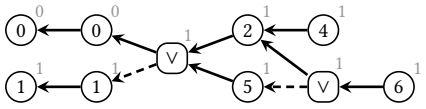

Figure 4: An example tECS. Union nodes are labeled by $\vee$ while non-union nodes are depicted as circles. Left and right children of union nodes are indicated by dashed and solid edges, respectively. The maximum-start of each node is at its top-right in grey.

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. A *full-path* in $\mathcal{E}$ is a path $\bar{p} = \mathsf{n}_1, \mathsf{n}_2, \ldots, \mathsf{n}_k$ such that n_k is a bottom node. Each full-path $\bar{p}$ 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 2. *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 $[\![\mathcal{A}]\!]^{\epsilon}_{j}(S)$ at every j. To that end, it will be necessary to enumerate, for certain nodes n in $\mathcal{E}$, the set

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

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

Example 3. *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* $[\![4]\!]^{6}_{\mathcal{E}}(4)$ *and* $[\![6]\!]^{6}_{\mathcal{E}}(6)$*, which contains all complex events output at position* 4 *and* 6*, respectively.*

A straightforward algorithm for enumerating $[\![\mathsf{n}]\!]^{\epsilon}_{\mathcal{E}}(j)$ 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 $\max(\mathsf{n}) = \max\big(\{i \mid (i, D) \in [\![\mathsf{n}]\!]_{\mathcal{E}}\}\big)$. 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

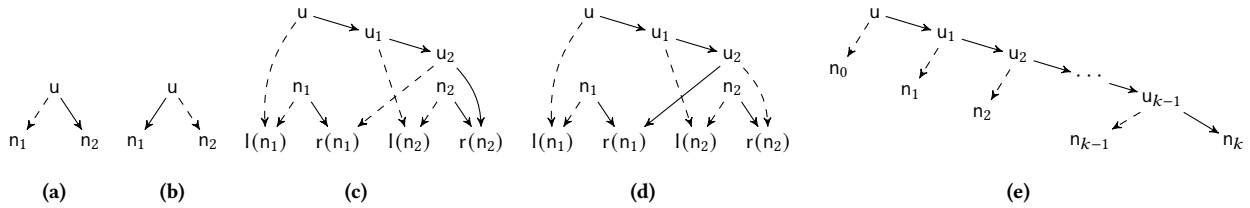

Figure 5: Gadgets for the implementation of the union method. The u nodes are union nodes, where the dashed and bold arrows symbolize the left and right nodes, respectively.

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, $[\![\mathsf{n}]\!]^{\epsilon}_{\mathcal{E}}(j)$ is non-empty and we perform the DFS-based enumeration. However, when we traverse a union node u we always visit left(u) before right(u). Moreover, we only visit right(u) if $j - \max(\text{right}(\mathsf{u})) \leq \epsilon$. Otherwise, right(u) and its descendants do not contribute to $[\![\mathsf{n}]\!]^{\epsilon}_{\mathcal{E}}(j)$, 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* odepth(n) of a node n recursively as follows: if n is a non-union node, then odepth(n) = 0; otherwise, odepth(n) = odepth(left(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 odepth(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 $[\![\bar{p}]\!]_{\mathcal{E}} \neq [\![\bar{q}]\!]_{\mathcal{E}}$.

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.*

4.2 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:

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

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 $\max(\mathsf{n}_1) = \max(\mathsf{n}_2)$. 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 $\max(\text{right}(\mathsf{n}_1)) \geq \max(\text{right}(\mathsf{n}_2))$, three new union nodes, u, u_1, and u_2 are added, and connected as shown in Figure 5(c). Finally, if $\max(\text{right}(\mathsf{n}_1)) < \max(\text{right}(\mathsf{n}_2))$, 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 $\mathsf{ul} = \mathsf{n}_0, \mathsf{n}_1, \ldots, \mathsf{n}_k$ such that (1) n_0 is non-union, (2) $\max(\mathsf{n}_0) \geq \max(\mathsf{n}_i)$ and (3) $\max(\mathsf{n}_j) > \max(\mathsf{n}_{j+1})$, 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 $\mathsf{ul} = \mathsf{n}_0, \ldots, \mathsf{n}_k$ in-place by inserting a safe node n such that $\max(\mathsf{n}) \leq \max(n_0)$. Specifically, if there is $i > 0$ such that $\max(\mathsf{n}_i) = \max(\mathsf{n})$, then it replaces n_i in ul by the result of calling union(n_i, n). This hence also updates $\mathcal{E}$.

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.

```
1:  procedure Evaluation(A, S, ε)
2:      j ← −1
3:      T ← ∅
4:      while t ← yield(S) do
5:          j ← j + 1
6:          T′ ← ∅
7:          ul ← new-ulist(new-bottom(j))
8:          ExecTrans(q0, ul, t, j)
9:          for p ∈ ordered-keys(T) do
10:             ExecTrans(p, T[p], t, j)
11:         T ← T′
12:         Output(j, ε)
13: procedure ExecTrans(p, ul, t, j)
14:     n ← merge(ul)
15:     if q ← Δ(p, t, •) then
16:         n′ ← extend(n, j)
17:         ul′ ← new-ulist(n′)
18:         Add(q, n′, ul′)
19:     if q ← Δ(p, t, ∘) then
20:         Add(q, n, ul)
21:     return
22: procedure Add(q, n, ul)
23:     if q ∈ keys(T′) then
24:         T′[q] ← insert(T′[q], n)
25:     else
26:         T′[q] ← ul
27:     return
28:
29: procedure Output(j, ε)
30:     for p ∈ keys(T) do
31:         if p ∈ F then
32:             n ← merge(T[p])
33:             Enumerate(n, j)
```

Otherwise, we discern two cases. If $\max(\mathsf{n}) = \max(\mathsf{n}_0)$, then n is inserted at position 1 in $\mathtt{ul}$. Otherwise, n is inserted between n_i and n_{i+1} with $i > 0$ such that $\max(\mathsf{n}_i) > \max(\mathsf{n}) > \max(\mathsf{n}_{i+1})$. The last method, merge($\mathtt{ul}$), takes a union-list $\mathtt{ul}$ 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 $\mathsf{u} = \mathsf{n}_0$. Otherwise, we add k union nodes $\mathsf{u}, \mathsf{u}_1, \ldots, \mathsf{u}_{k-1}$ 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 $\mathtt{ul}$.

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] ← $\mathtt{ul}$ for inserting or updating it with a union-list $\mathtt{ul}$. 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., 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 [9], 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.

4.3 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 4, its goal is to enumerate, at every position j in the stream, the set $[\![\mathcal{A}]\!]^{\epsilon}_j(S)$ of complex events produced by accepting runs terminating at j that satisfy the time-bound ϵ. It does so by

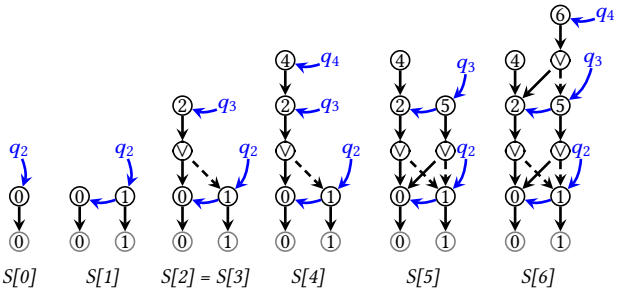

Figure 6: Illustration of Algorithm 1 on the CEA $\mathcal{A}$ and stream S of Figure 3.

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 $i \leq j$ 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 $[\![\mathcal{A}]\!]^{\epsilon}_{j}(S)$, 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 $\mathrm{T}'[q]$. In lines 19–20, if there is a non-marking transition reaching q from p, we add `n` directly to $\mathrm{T}'[q]$ 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 $[\![\mathcal{A}]\!]^{\epsilon}_{j}(S)$. 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 4.2, 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 $[\![\mathcal{A}]\!]^{\epsilon}_{j}(S)$ 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. 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 4.3 update T, T', and $\mathcal{E}$ by means of the methods of Section 4.2. All of these either take constant time, or time linear in the size of the union list being manipulated. We can show 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.

4.4 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 [33].

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 5).

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.

5 EXPERIMENTS

In this section, we compare CORE against four leading CER systems: SASE [53], Esper [4], FlinkCEP [5], and OpenCEP [6]. 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 surveyed the literature for other systems to compare against but found ourselves limited to these baselines, as explained in the online appendix [21].

Setup. We compare against SASE v.1.0, Esper v.8.7.0, FlinkCEP v.1.12.2, and OpenCEP (commit `e320ad8`). All systems are implemented in Java except for OpenCEP which uses Python 3.9.0. We run 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. Java and Python virtual machines are restarted with freshly allocated memory before each run.

We compare systems on throughput and memory consumption. All reported numbers are averages over 3 runs. We measure throughput, expressed as the number of events processed per second (e/s), as follows. We first load the input stream completely in main memory to avoid measuring the data loading time. We then start the timer and allow systems to read and process events as fast as they can. After 30 seconds, we disallow reading further events and stop the timer when the last read event has been fully processed, with a timeout of 1 minute. We report the average throughput, expressed as the total number of processed events divided by the total running time. Runs that time-out have a value of "aborted" for the average, and will not be plotted. Recognized complex events are logged to main memory. We adopt the consumption policy [26, 30] that forgets all events read so far when a complex event is found. We adopt this policy for all systems because it is the only one supported by Esper and SASE. We measure memory consumption every 10000 events, and report the average value. Before measuring memory consumption, we always first call the garbage collector. 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 complex events. When this was not the case, we explicitly mention this difference below.

CORE and SASE are single-core, sequential programs. To ensure fair comparison, all of the systems are therefore run in a single-core, sequential setup. Esper, FlinkCEP, and OpenCEP 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 the competition by 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 orders of magnitude more processors before meeting COREs throughput. Therefore, we do not consider a setup with parallelization.

All the experiments are reproducible. The data and scripts can be found in [1].

Datasets. We run our experiments over three real datasets: (1) *the stock market dataset* [7] containing buy and sell events of stocks in a single market day; (2) *the smart homes dataset* [2] containing power measurements of smart plugs deployed in different households; and (3) *the taxi trips dataset* [3] recording taxi trips events in New York. Each original dataset includes several millions of real-world events and have already been used in the past to compare CER systems (e.g. [44–47]). For each dataset, we run experiments on a prefix of the full stream consisting of about one million events. Full details on the datasets, the prefix considered, and the queries that we run are given in the online appendix [1].

Sequence Queries With Output. We start by considering sequence queries, which have been used for benchmarking in CER before (see, for example, [28, 48, 53–55]). Specifically, for each dataset we fix a workload of queries P_n of the following form, each detecting sequences of n events.

```
Pn := SELECT * FROM Dataset
      WHERE A1 ; A2 ; ... ; An
      FILTER A1[filter_1] AND ... AND An[filter_n]
      WITHIN T
```

Here, `A1`, . . ., `An` and `filter_1`, . . ., `filter_n` are dataset-dependent, as detailed in the online appendix. We consider sequences of length $n = 3$, 6, 9, 12, and 24. In this experiment, we use a fixed time window `T` of 10 seconds for the stock market and smart homes dataset, and of 2.7 hours for the taxi trip dataset, which is enough to find several outputs. While P_{12} and P_{24} usually do not produce outputs, they are nevertheless useful to see how systems scale to long sequence queries.

In the following analysis, when the processing of an input event triggers one or more complex events to be recognized, we will refer to this triggering input event as a *pattern occurrence*. Each pattern occurrence may generate multiple complex event outputs. In our experiments, when a pattern occurrence is found, we only enumerate the first thousand corresponding complex events outputs. An exception is FlinkCEP where, for implementation reasons, we only generate the first such output. Note that this favors FlinkCEP since it needs to produce less output. For OpenCEP we use the so-called ANY evaluation strategy, which is the only one that allows enumerating all matches. Unfortunately, it does not produce the same outputs as CORE, SASE, and FlinkCEP. Despite this, the number of results produced by OpenCEP is similar to other systems, making the comparison reasonable.

Figure 7 (left), plots the measured throughput and memory consumption (in log scale) as a function of the sequence length n. CORE's throughput is in the range of $[10^5, 10^6]$ e/s. This throughput fluctuates due to variations in the number of complex events found, which must all be reported before an event can be declared fully processed. For example, for the smart homes dataset, queries P_9 and P_{12} are very output intensive (i.e., around 1 million pattern occurrences in total), and the fluctuations are most noticeable for this dataset. Instead, the stocks and taxi trips datasets have less pattern occurrences (i.e., around a few thousand) and we can see that CORE's throughput is more stable. Note in particular that CORE's throughput is only linearly affected by n in these cases.

We next compare to the other systems. For $n = 3$, the throughput of most baselines is one order of magnitude (OOM) lower than CORE, except for Esper which has comparable throughput on stock and smart homes. However, as n grows, the throughput of all systems except CORE degrades exponentially. For the smart homes dataset, the throughput of FlinkCEP is more stable, although up to

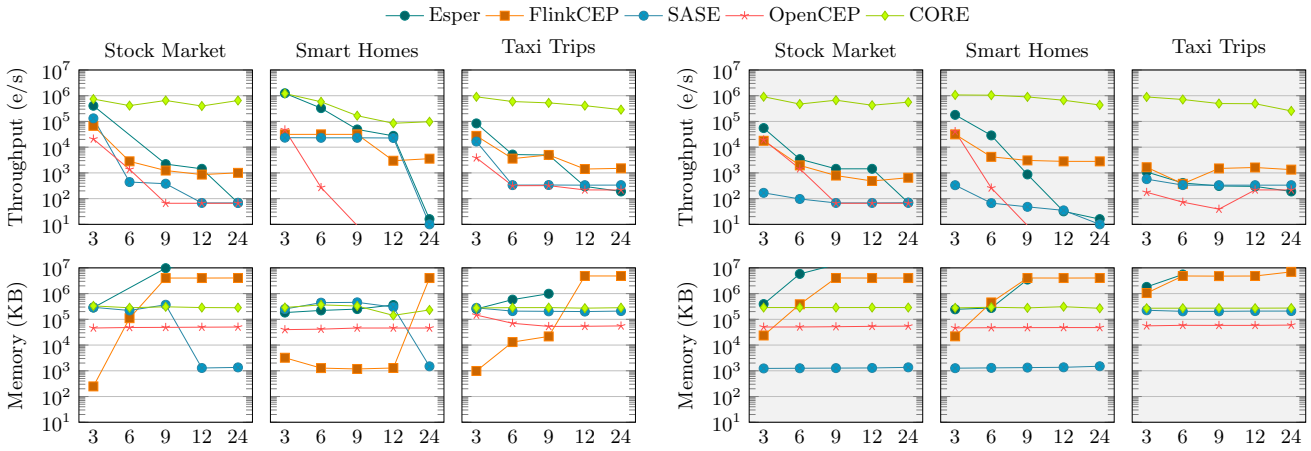

Figure 7: Throughput (higher is better) and memory consumption (lower is better) as a function of sequence length n, on queries with output (left) and queries without outputs (right, in gray).

2 OOM lower than CORE. Recall, however, that FlinkCEP produces only a single complex event per pattern occurrence, while all other systems enumerate up to one thousand complex events per occurrence. In contrast to the baselines, CORE's throughput is stable, only affected by the high number of complex events found, and degrades only linearly in n on stocks and taxis. As a consequence, on these datasets, CORE's throughput is 1 to 5 OOM higher than the baselines for large values of n.

The memory used by CORE is high (~300MB) but stable in n. By contrast, the memory consumption of Esper and FlinkCEP can grow exponentially in n. OpenCEP and SASE[4] are special cases whose memory consumption is stable and comparable to CORE.

Sequence Queries Without Output. In practice, we may expect CER systems to look for unusual patterns in the sense that the number of complex events found is small. Our next experiment captures this setting. For each query P_n we create a variant P'_n by adding, to the sequence pattern of P_n, an additional event that never occurs in the stream. These variant queries hence never produce outputs. Since systems do not know this, however, they must inherently look for partial matches that satisfy the original sequence query P_n. Because no time is spent enumerating complex events when processing P'_n, this experiment can hence also be viewed as a way of measuring only the update performance of P_n. Note that, even though no complex events are found, systems may still materialize a large number of partial matches.

Figure 7 (right) plots the results (in log scale) on P'_n for $n = 3$, 6, 9, 12, 24. We see that CORE's throughput is comparable to the throughput on P_n (Figure 7, left), except for the smart homes dataset, where the throughput has improved. Overall, CORE's throughput is on the order of 10^6 e/s, decreasing mostly linearly with n. By contrast, other systems have lower throughput, by 1 to 5 OOM, and this throughput decreases much more rapidly in n. This is most observable for Esper and OpenCEP, whose performance drops exponentially in n. For the taxi dataset, the throughput of all systems

[4]The memory consumption of SASE drops for P_{12} and P_{24}. This is because in these cases the number of events that SASE can successfully process in full is significantly less than for smaller values of n, and also less than other systems .

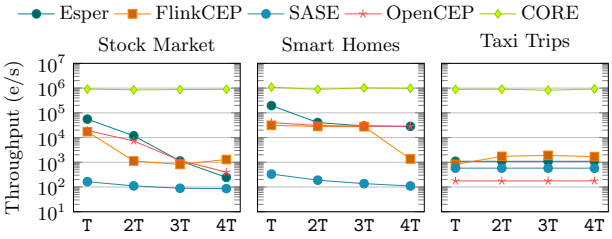

Figure 8: Throughput as a function of time-window size.

reduces to less 10^4 e/s compared to the original P_n setting, most probably because of a high number of partial matches that are maintained but never completed. CORE is not affected by the number of partial matches, which explains its stable performance.

Notice that CORE does not use *lazy evaluation* [36, 37, 40] as some other systems do: it eagerly updates its internal data structure on each input event.A CER system that uses lazy evaluation could perform well when there is no output (i.e., for P'_n) but badly when several pattern occurrences appear (i.e., for P_n). One can see CORE as the best of both worlds. It does work event after event, and results are ready for enumeration at each pattern occurrence.

Memory consumption of most systems remains high, with the memory usage of Esper and FlinkCEP increases exponentially with n. CORE's memory consumption is stable, like the consumption of SASE[5] and OpenCEP.

Varying the Time-Window Size. We next measure the effect of varying the time window size on processing performance. We do so by fixing query P'_3, and varying the window size from T to 4T where T is the original window size. We fix P'_3 because this allows us to measure the effect of update processing only, and because performance on P'_n is highest for all systems when $n = 3$.

The results (log scale) are in Figure 8. We see that CORE outperforms other systems by at least one OOM for size T and by 2 to 4 OOM for 4T. Note that for stock market and smart homes dataset we are still using relatively small time windows of a few hundreds

[5]Memory consumption for SASE is lower compared to Figure 7 (left) because the number of events that SASE can successfully process in full is significantly less.

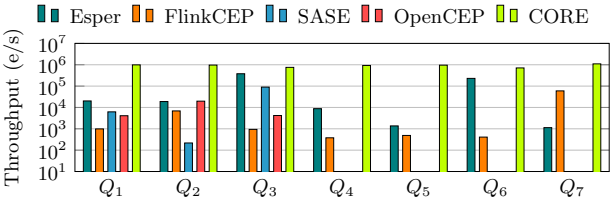

Figure 9: Throughput of evaluating queries Q_1 to Q_7, which include non-sequence operators, over stock market data.

events; in practice windows may be significantly larger (e.g., taxi trips dataset). We also observe that the throughput of other systems may degrade exponentially as the size of the time window grows. Indeed, this is clear for Esper, FlinkCEP, and OpenCEP on stock market dataset, where the throughput is around 10^5 e/s at `T` but less than 10^3 e/s at `4T`. In contrast, CORE consistently maintains its performance at 10^6 e/s, and it is not affected by the window size, as the theoretical analysis predicted.

In the online appendix [21], we report similar experimental results even when all systems are allowed to use a selection strategy heuristic to aid in faster processing.

Other Operators. For the last experiment, we consider a diverse workload of queries where other operators like disjunction, iteration, or partition-by are used. For these queries, we report only on the stock market dataset. 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`

SASE and OpenCEP do not support disjunction, and we hence omit Q_4–Q_7 for them. 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 we show the throughput (log-scale), grouped per query. The results confirm our observations from the previous experiments. In particular, CORE's throughput is stable over all queries (i.e., 10^6 e/s), in contrast to the baselines, which are not stable. In particular for every baseline system there is at least one query where CORE' exhibits 2 OOM or more higher throughput.

Q_2 and Q_5 add filter clauses to Q_1 and Q_4 respectively. If we contrast system performance on Q_2 and Q_5 with that on Q_1 and Q_4, respectively, then we see that adding such filters reduces performance of some baselines, most notably SASE and Esper. CORE does not suffer from these problems due to its evaluation algorithm.

Q_3 and Q_6 add a partition-by clause to Q_1 and Q_4, respectively. If we contrast system performance on Q_3 and Q_6 with that on Q_1 and Q_4, respectively, then we see that partition-by aids the performance of systems like Esper and SASE but slightly decreases the throughput of CORE. This is because CORE evaluate the partition-by clause by running several instances of the main algorithm, one for each partition, which slightly diminishes the throughput. Nevertheless, CORE still outperforms the baselines.

Limitations of CORE. As the previous experiments show, CORE outperforms other systems by several orders of magnitudes on different query and data workloads. We finish this section by a discussion of the limitations of CORE.

In particular, we see two limitations compared to other approaches. First, CORE's asymptotically optimal performance comes from representing partial matches succinctly in a compact structure, and enumerating complex events from this structure whenever a pattern occurrence is found. The enumeration is with output-linear delay, meaning that the time spent to enumerate complex event C is asymptotically $O(|C|)$, i.e., linear in C. This asymptotic analysis hides a constant factor. While our experiments show that this constant is negligible in practice, enumeration from the data structure may take longer than directly fetching the complex event from an uncompressed representation, as most other systems do. This difference may be important in situations where a user wants to access outputs multiple times.

Second, CORE compiles CEQL queries into non-deterministic CEA, which need to be (on-the-fly) determinized before execution. In the worst case, this determinized CEA can be of size exponential in the size of the query. Since the time needed to process a new event tuple, while constant in data complexity, does depends on the CEA size (Section 4.3), this may also affect processing performance. While we did not observe this blowup on the queries in this section, CEQL queries with complicated nesting of iteration and disjunction can theoretically exhibit such behavior. Unfortunately, no baseline system (except CORE) currently allows such nesting, and we are therefore unable to experimentally compare CORE in such a setting.

6 CONCLUSIONS AND FUTURE WORK

We introduced CORE, the first CER system whose evaluation algorithm guarantees constant time per event, followed by output-linear delay enumeration. We showed experimentally that this algorithm provides stable performance, being unaffected by the size of the stream, query, or time window, and leading to a throughput up to five orders of magnitudes higher than the state of the art.

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 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.

ACKNOWLEDGMENTS

The authors are grateful to Martín Ugarte for stimulating discussions that lie at the basis of this work. A. Grez, A. Quintana, and C. Riveros were supported by ANID - Millennium Science Initiative Program - Code `ICN17_002`. S. Vansummeren was supported by the Bijzonder Onderzoeksfonds (BOF) of Hasselt University (Belgium) under Grant No. `BOF20ZAP02`.

REFERENCES

[1] [n.d.]. CORE Website. https://github.com/CORE-cer. Accessed on 2022-03-15.

[2] [n.d.]. DEBS 2014 Grand Challenge: Smart homes. https://debs.org/grand-challenges/2014/. Accessed on 2022-03-11.

[3] [n.d.]. DEBS 2015 Grand Challenge: Taxi Trips. https://debs.org/grand-challenges/2015/. Accessed on 2022-03-11.

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

[5] [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.

[6] [n.d.]. OpenCEP. https://research.redhat.com/blog/research_project/complex-event-processing-2/. Accessed on 2022-03-11.

[7] [n.d.]. Stock market data traces. https://davis.wpi.edu/datasets/Stock_Trace_Data/. Accessed on 2022-03-11.

[8] 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.

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

[21] Marco Bucchi, Alejandro Grez, Andrés Quintana, Cristian Riveros, and Stijn Vansummeren. 2022. CORE: a COmplex event Recognition Engine. https://arxiv.org/abs/2111.04635. Extended paper version.

[22] 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.

[23] 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.

[24] 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.

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

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

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

[35] 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.

[36] Ilya Kolchinsky and Assaf Schuster. 2018. Efficient Adaptive Detection of Complex Event Patterns. *Proc. VLDB Endow.* 11, 11 (2018), 1346–1359. https://doi.org/10.14778/3236187.3236190

[37] Ilya Kolchinsky, Izchak Sharfman, and Assaf Schuster. 2015. Lazy evaluation methods for detecting complex events. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems, DEBS '15, Oslo, Norway, June 29 - July 3, 2015*, Frank Eliassen and Roman Vitenberg (Eds.). ACM, 34–45. https://doi.org/10.1145/2675743.2771832

[38] 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.

[39] 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.

[40] 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.

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

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

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

[50] 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.

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

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

[53] 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.

[54] 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.

[55] 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.

[56] 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.

[57] 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.