CORE: a COmplex event Recognition Engine

Marco Bucchi
PUC Chile
mabucchi@uc.cl

Alejandro Grez
PUC Chile
ajgrez@uc.cl

Andrés Quintana
PUC Chile
afquintana@uc.cl

Cristian Riveros
PUC Chile
cristian.riveros@uc.cl

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

Abstract

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

1 Introduction

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

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

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

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

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

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

In this paper, we present CORE, a novel COmplex event Recognition Engine that focuses on the efficient evaluation of a large class of complex event queries. In particular, CORE uses a novel evaluation

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

Contributions. Our contributions are as follows.

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

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

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

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

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

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

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

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

2 CEQL by example

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

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

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

(Q_1)

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

(Q_2)

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

(Q_3)

Figure 1: CEQL queries on a Stock stream.

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

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

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

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

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

3 CEQL syntax and semantics

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

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

A *stream* is a possibly infinite sequence $S = t_0 t_1 t_2 \ldots$ 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 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 >]