Distributed Complex Event Recognition

Master Thesis

Master in Innovation and Research in Informatics
Advanced Computing

Arnau Abella

January, 2022

FACULTAT D'INFORMÀTICA DE BARCELONA

UNIVERSITAT POLITÈCNICA DE CATALUNYA

Advisors:
Sergi Nadal, Universitat Politècnica de Catalunya

Stijn Vansummeren, UHasselt – Hasselt University

DISTRIBUTED COMPLEX EVENT RECOGNITION

The complete source code for this document is available from

https://github.com/dtim-upc/DCORE

Acknowledgements

I have so many people to thank. First, and foremost, I am grateful to my advisor Sergi Nadal for his enthusiasm and guidance. This thesis would have not been possible without him. Second, I am due to my co-advisor Stijn Vansummeren. His insightful remarks and suggestions have made some of the contributions of this thesis possible. Thanks to Marco Bucchi, Alejandro Grez, Andrés Quintana, and Cristian Riveros, among many others, for their numerous contributions to the field of complex event recognition that made this work possible. I owe a debt of gratitude to my colleague and friend, Juan Pablo Royo. He is the reason I started this master's degree after three years away from the academia. I cannot express enough gratitude toward my family, old and new, who have supported me in every possible way during these two harsh years. To my parents. They have supported me during the span of this master's degree. They are both remarkable parents and I am very lucky to be their son. To my brother Adrià. He is the best brother someone could ask for. I am left to thank my partner Marta. She has been with me every step on the way. No words can truly express how I feel: I love you and thank you.

Abstract

Complex Event Recognition (CER) has emerged as a prominent technology for detecting situations of interest, in the form of query patterns, over large streams of data in real-time. Thus, having query evaluation mechanisms that minimize latency is a shared desiderata. Nonetheless, the evaluation of CER queries is well known to be computationally expensive. Indeed, such evaluation requires maintaining a set of partial matches which grows super-linearly in the number of processed events. While most prominent solutions for CER run in a centralized setting, this has proved inefficient for Big Data requirements, where it is necessary to scale the system to cope with an increasing arrival rate of events while maintaining a stable throughput. To overcome these issues, we propose a novel distributed CER system that focuses on the efficient evaluation of a large class of complex event queries, including n-ary predicates, time windows, and partition-by event correlation operator. This system uses a state-of-the-art automaton-based distributed algorithm that circumvents the super-linear partial match problem. Moreover, in the presence of heavy workloads, the system can scale-out by increasing the number of processing units with little overhead. We additionally provide a proof of correctness of the algorithm. We experimentally compare our system against the state-of-the-art sequential CER engine that inspired our work and show that our system outperform its predecessor in the presence of queries with complex predicates. Furthermore, we show that, in the presence of Big Data requirements, our system performance is overall better.

Contents

List of Figures **v**

1 Introduction **1**

1.1 Contributions 4

1.2 Outline 5

2 Related Work **6**

2.1 CER systems 6

2.2 Stream partitioning 7

2.3 Distributed CER 9

2.4 Chapter summary 10

3 Preliminaries **11**

3.1 Distributed computing 11

3.2 Complex event logic 12

3.3 Selection strategies 15

3.4 Computational model 16

3.5 Chapter summary 18

4 Distributed CER **19**

4.1 Distributed CER framework 20

4.2 Distributed CER Engine 21

4.2.1 Distribution Strategies 23

4.3 Distributed CORE 24

4.4 Chapter summary 26

5 Distributed evaluation algorithm **27**

5.1 The data structure 28

5.2 Auxiliary data structures 31

5.3 The evaluation algorithm 33

5.4 The Enumeration procedure 36

5.5 Chapter summary 43

6 Experimental evaluation **44**

6.1 DCORE in a nutshell 44

6.2 Experimental setup 45

6.3 Experiments on the evaluation of complex predicates 48

6.4 Experiments on the scalability of the framework 49

6.5 Experiments on DCORE's evaluation algorithm under Big Data requirements 50

6.6 Chapter summary 53

7 Conclusions and future work **54**

Appendix A **55**

A.1 Proof of Theorem 5.3 55

A.2 Algorithms Chapter 4 56

A.2.1 Maximal Complex Event Enumeration 56

Bibliography **60**

List of Figures

1.1 Exemplary stream of events measuring temperature (T) and relative humidity (H) 2

1.2 Query on a wireless sensors network stream, which goal is to detect fires. 2

3.1 CEQL query on a wireless sensors network stream. 13

3.2 The semantics of CEL. 15

3.3 A CEA representing the query from Figure 3.1 and an example of stream. 17

4.1 Distributed CER framework. 20

4.2 Distributed CER Engine architecture. 22

4.3 Distributed CORE architecture. 25

5.1 Visualisation of the four cases of method union(u). The u are union nodes, where the dashed and bold arrows point to the left and right node, respectively. 31

5.2 Visualisation of method merge(ul). The u are union nodes, where the dashed and bold arrows point to the left and right node, respectively. 32

5.3 Illustration of Algorithm 1 on the CEA $\mathcal{A}$ and stream S of Figure 3.3. 33

5.4 Illustration of Algorithm 2 on the tECS $\mathcal{E}$ of Figure 5.3. 37

6.1 Queries considered in the experimental setting 47

6.2 Example input streams 47

6.3 The performance of evaluating queries Q_1, Q_2, and Q_3 over stream S_1, S_2, S_3, respectively. 48

6.4 The coefficient of variation of evaluating queries Q_2, and Q_3 under distributions strategies RR, PoTC, ES, and MCEE. . . 49

6.5 The horizontal scalability of DCERE and DCORE evaluating query Q_2 over increasing number of processing units. 50

6.6 The horizontal scalability of DCORE under increasing number of processing units. 51

6.7 The execution time on each processing unit of DCORE evaluating queries Q_1, Q_2, and Q_3. 52

6.8 Comparison of the execution time of an optimal distributed system based on CORE against DCORE. 53

Chapter 1

Introduction

Complex Event Recognition (CER), also called Complex Event Processing, refers to the activity of identifying, in streams of high-velocity continuously arriving primitive event data, collections of events that collectively satisfy some pattern. These collections of events that satisfy some pattern are referred as *complex events*. Conceptually, CER systems allow expressing patterns that match incoming events not only on the basis of their content, but also on where they occur in the input stream, and how this order relates to other events in the stream, in addition to other constraints between events. CER queries distinguish themselves from streaming queries supported by engines such as Flink [1], or Spark [2] in that CER queries include *regular expressions operators* like *sequencing*, *disjunction* and *iteration* to express spatio-temporal constraints that are not supported by stream processing engines.

In recent years, CER has emerged as a prominent technology. It has been successfully applied in scenarios like maritime monitoring [3], network intrusion detection [4], industrial control systems [5], and real-time analytics [6]. Prominent examples of CER systems from academia and industry include Cayuga [7], CORE [8], Esper [9], SASE [10], and TESLA [11], among others (see surveys [12, 13]). All such systems share the common goal of providing timely reaction to situations of interest in a real-time manner. Thus, having query evaluation mechanisms that minimize latency is a shared desiderata. Nonetheless, the evaluation of CER queries is well-known to be computationally expensive. Indeed, such evaluation requires the maintenance of a set of partial matches which grows *super-linearly* in the number of processed events. We illustrate this problem with an example as follows.

Example 1.1. *Consider a stream produced by wireless sensors placed in a warehouse, whose main objective is to detect fires. We assume each sensor can measure both temperature (in Celsius degrees) and relative humidity (as a percentage). Additionally, each sensor is assigned a id corresponding to the zone of the warehouse where the sensor is located. The* events *produced by the sensors are composed of the id of the sensor and a measurement corresponding to temperature or relative humidity. We write $T(id, val)$ for an event reporting temperature val from sensor id, and $H(id, val)$ for an event reporting humidity val from sensor id. An excerpt of the stream of events, indexed by order of arrival, is depicted in Figure 1.1.*

type	H	T	H	H	T	H	H	T	T	...
id	1	1	2	1	2	2	1	1	1	...
val	50	24	49	24	24	42	23	40	45	
timestamp	0	1	2	3	4	5	6	7	8	...

FIGURE 1.1: Exemplary stream of events measuring temperature (T) and relative humidity (H)

For the sake of illustration, assume that it has been detected that when the temperature of a storage room increases from below 30 celsius degrees to above 40 celsius degrees and the humidity is below 25% there is a high probability of fire. The following query retrieves the id of the zone where the fire might be originated so the notification system can warn the security team.

```
SELECT t2.id FROM warehouse
WHERE (T as t1; H as h1; T as t2)
FILTER t1[val < 30] AND h1[val < 25]
  AND t2[val > 40] AND t1[id] = h1[id]
  AND h1[id] = t2[id]
WITHIN 10 events
```

FIGURE 1.2: Query on a wireless sensors network stream, which goal is to detect fires.

When the query from Figure 1.2 is applied to the input stream from Figure 1.1, the resulting complex events are: $\{1, 3, 7\}$, $\{1, 6, 7\}$, $\{1, 3, 6, 7\}$, $\{1, 3, 8\}$, $\{1, 6, 8\}$, $\{1, 3, 6, 8\}$, $\{1, 3, 7, 8\}$, $\{1, 6, 7, 8\}$, *and* $\{1, 3, 6, 7, 8\}$. *Observe that, within a given time window, the number of* partial matches *that consist of a temperature measurement followed by a humidity measurement followed by a temperature measurement may easily be cubic in the number of events in the window.*

In order to overcome the issues illustrated by Example 1.1, current CER engines either materialize, or lazily compute the set of partial matches to be able to quickly determine when a temperature measurement constitute an answer to our query. In either case, an $\Omega(N^3)$ operation is required to complete the processing of the complex event, where N is the number of previous events seen within the sliding window. This gets worsened under the default *skip-till-any-match* [14] policy, where queries that include the iterator operator may have sets of partial matches that grow *exponentially* in N [8]. Given the computational challenges of CER query evaluation, there has been ongoing research on this field [15, 16], where a wide range of different execution models has been proposed [12, 13]. Although, all of these system still suffer from overhead super-linear in the size of the stream, and thus their scalability is limited to queries over short time windows.

An attempt to overcome the detrimental super-linear complexity of contemporary CER systems is the *COmplex event Recognition Engine (CORE)* engine [8]. Such engine builds on top of a *rigorous* and *efficient* framework for CER that leverages the so called *Complex Event Logic* (CEL) [17, 16]. To do so, it employs a formal language for specifying complex events, called *CEQL*, that contains many features used in the literature including time windows as well as a partition-by event correlation operator [18, 8]. Such language can be compiled into a *formal computational model* called *Complex Event Automata* (CEA). CORE incorporates an efficient algorithm for evaluating CEA over event streams using constant time, under data complexity, per event followed by output-linear delay enumeration of the complex events, which is not affected by the length of the stream, size of the query, or size of the time window [16, 8].

One downside of CORE is that its filtering capabilities are limited to unary predicates. [18] shows that unary CEL and CEA are expressively equivalent, however, incomparable when equipped with n-ary predicates (e.g., equi-joins like `t1[id] = h1[id]`). In particular, when CEL is restricted to binary predicates, it is strictly more expressive than CEA. As a result, CORE cannot embed the processing of n-ary filtering predicate in the automaton computational model, and thus cannot guarantee optimal performance under non-unary predicates. This only get aggravated in the presence of iteration operators (i.e., the *kleene star*), where the set of partial matches may grow exponentially in

the size of the stream, resulting in an exponential cost of enumerating the complex events.

Departing from the discussion and challenges identified above, in this thesis, we embark on the task of giving a new distributed framework for CER that deals with the limitations of many CER system to express and process complex predicates while preserving optimal performance. To that end, we explore how the evaluation of CER queries with n-ary filter predicates can be distributed and parallelized. Considering the fact that such kind of complex filter predicates cannot be embedded into the automaton computational model of CORE, we propose to consider them as a post-process after the enumeration phase. Hence, this thesis is focused on studying and proposing different distribution strategies that optimize such phase. We consider, implement and compare multiple distributed architectures, from the processing of complex events in a centralized fashion distributing the filtering predicates to performing the processing of complex events in a distributed fashion as well. All such features are implemented in a novel distributed architecture for CER, namely DCORE (which stands for *Distributed COmplex event Recognition Engine*).

Note. Throughout the development of this thesis, several new publications on the area had been published (e.g., [16, 8]), which impacted the results of this work.

1.1 Contributions

Our contributions are summarized as follows:

(i) We present a distributed framework for CER. This framework circumvents the filtering limitations of CORE while preserving optimal throughput. Based on this framework, we implemented two different architectures: DCERE and DCORE. DCORE uses the novel distributed evaluation algorithm for CER presented in this work.

(ii) We present a novel distributed evaluation algorithm for CER. The proposed algorithm tackles (1) the super-linear complexity of non-unary predicates, and (2) the exponential complexity of the enumeration phase. Our work includes a proof of correctness of this algorithm.

(iii) We show that our distributed framework is practical. Our experiments show that, in the presence of Big Data requirements, our distributed framework outperforms CORE on processing queries with complex predicates.

1.2 Outline

The document is organised as follows. We discuss related work in Chapter 2. We give an introduction to CEQL and describe how CEQL is compiled into CEA in Chapter 3. We introduce the distributed CER framework on Chapter 4. In Chapter 5 we present the novel distributed evaluation algorithm. We dedicate Chapter 6 to the implementation of the framework and the experiments. We present our conclusions and future work on Chapter 7.

Chapter 2

Related Work

2.1 CER systems

CER systems are usually divided into three categories: automata-based, tree-based, and logic-based, with some systems (e.g. [9, 11]) being hybrids. We situate our work among the three approaches, and refer the reader to recent surveys [12, 13, 15] for an in-depth discussion of these classes of systems.

Automata-based systems. *Automata-based systems* typically propose a CER query language that is inspired by regular expressions which are evaluated by custom automata models. Our work has been inspired by CORE [8], and falls in the class of automata-based systems. Previous proposals (like SASE [10]) do not provide *denotational semantics* for their language. As a result, either iterations cannot be nested [14], or their semantics is confusing [19]. Other proposals (like TESLA [11]) have formal semantics, but they do not include the iteration operator. An exception is Cayuga [7], but their sequencing operator is non-associative, which results in confusing semantics. CORE [8] is the first framework that provides a well-defined formal semantics that is compositional, allowing arbitrary nesting of operators. Moreover, it is the first evaluation algorithm that guarantees, under data complexity, constant time per event and output-linear delay enumeration.

Tree-based systems. *Tree-based systems* [9, 20, 21] typically consider a CER query language that is inspired by a regular expression. Unlike automata-based system, the queries are evaluated by constructing and evaluating a tree

of CER operators. Again, the semantics of queries is not formally defined. Moreover, the evaluation trees do not have formal performance guarantees, even for regular CER queries.

Logic-based systems. *Logic-based systems* [22, 23, 24] typically express CER queries as rules in some form of logic, e.g., *temporal logic* or *event calculus*, and consequently evaluate CER as logical inference. Consequently, logic-based systems have formal semantics. However, iteration is often expressed by means of recursive rules instead of as an individual operator. Such rule can detect that a CER pattern applies repeatedly, but they do not typically capture the participating events as part of the complex event. Then, the semantics of iteration in logic-based system is different from CEQL.

CORE. *COmplex event Recognition Engine (CORE)* [8] is the first CER engine that circumvents the super-linear partial match problem. It is the first system to propose a representation of partial matches with formal, proven, and optimal performance guarantees. It uses the query language CEQL that has well-defined formal semantics [18]. One limitation is that the evaluation algorithm is sequential. Another limitation is that it is restricted to unary CEL.

2.2 Stream partitioning

The basic idea of partitioning a stream of events is to split it so that each processing unit receives only a fraction of it. Thus, the goal is to maximize load balance across processing units. This is the main approach in the dataflow graph paradigm.

Shuffle. Also known as round robin, this is the most basic approach for stream partitioning. It consists of blindly routing events to the processing units in a circular fashion. A perfect load balance is achieved, as each processing unit will receive exactly an even fraction of the stream. This is the best approach for stateless operators, as they are executed to individual events, however it will poorly perform for the stateful case where events must be colocated. In such case, the cost of data repartitioning will be extremely high.

Field. Also known as hash, relies on hash functions defined over attributes of the stream to decide to which processing unit to route each event. The

most naïve approach is to define a key and use it as input to the hash function. This will distribute all events that should be collocated into the same processing unit, and thus stateful operators will have a good performance. However, such settings will greatly fail in the presence of skewness in the used key. To alleviate such constraints, cost-based approaches have been proposed to deal with skewed streams. In order to dynamically adapt to changes in the stream, such methods require to continuously monitor the keys. For instance, [25] introduces the concept of *load imbalance*. It uses two hash functions, an explicit and a consistent hash, with the goal of minimizing an objective function that combines such load imbalance the state migration cost. To monitor the most frequent keys, the *Lossy Counting* algorithm is used. Similarly, [26] presents the *Distribution-aware Key Grouping* algorithm (DKG). DKG has a learning phase where the heavy hitter keys are detected, using the *Space Saving* algorithm, then in the deployment phase it is used to route events.

Partial key grouping. Partial key grouping approaches lie in the middle between the shuffle and field approaches. They build upon the idea of balanced allocations [27], stating that routing tuples to the least "full" (in terms of size) processing unit, out of $d \geq 2$ choices, will highly increase the overall load balancing and thus, performance. [28] presents an approach using two hash functions for the case when $d = 2$, later extended for the case when $d \geq 2$ [29]. In such settings, frequent keys are monitored using the heavy hitter algorithm *Space Saving*. Partial key grouping algorithms avoid sending to the same processing unit skewed keys, and thus avoid the bottleneck generated by hashing approaches. This, however, incurs a cost in data shuffling over the network when events must be colocated to build a state.

Network optimization. Such approaches advocate that the cost of moving data over the network to colocate tuples that build a state (i.e., aggregation cost) should be considered as first class citizen in partitoning algorithms, and thus minimized. In [30], the authors propose to track the aggregation cost by counting the number of distinct keys sent to each processing unit, achieved via the *HyperLogLog* algorithm. Then, multiple hash functions are used combining the load imbalance and aggregation cost. A different approach is the one presented in [31], where the authors propose to find correlations in used in subsequent operators and colocate them in the same processing instance. This is achieved by maintaining statistics on the distribution of keys across operators. Then, for each pair of subsequent stateful operators, a bipartite

graph is built where nodes represent each distinct key and its weight, and edges the correlations. Thus, the problem is reduced to a graph partitioning problem such that pairs of keys that are highly correlated are in the same group (i.e., same processing unit).

2.3 Distributed CER

Distributed CER systems typically propose to increase the throughput by distributing the workload into a cluster of machines. Several distributed CER systems have been previously proposed [9, 32, 19, 33, 34], however, they do not usually have a well-defined computational model with clear performance guarantees [33], and they do usually suffer from communication overhead and require complex heuristics to optimize performance [34]

Query partitioning. These approaches deal with an automata-based computational model for CER. The idea behind query partitioning is to replicate an instance of the automaton to each processing unit. Now, however, each incoming event is a candidate to trigger any automata transition. Thus, each instance of the automata will receive all events but disregard most of them. This approach can be combined with hashing, if one can define a key for the events. This is the solution presented in [35] in the context of the Cayuga system [36]. Combined with hashing, [37] also presents an approach to deal with query partitioning with a custom CER language built on top of IBM's System S [38]. [39] presents a fine-grained approach to query partitioning, where besides providing a `PARTITION BY` operator that performs hashing, they also parallelize different runs of the automaton. To ensure correctness when routing the events to the active runs, queries are restricted with a `MAXLENGTH` parameter and batches of events smaller with a cardinality smaller than such parameter are sent to the active run.

Pipelining. The pipelining approach builds on the idea that every NFA with at least one forward edge can be split into smaller automata running on separate processing units [35]. Then, each processing unit will take care of a partition of the automaton, and there will be special transitions that span across processing units. Similarly as before, since an incoming event can cause state transitions, each processing unit must receive every event.

2.4 Chapter summary

In this chapter, we presented the work related to our research. First, we described the three categories of CER systems and introduced the CORE engine. Then, we discussed several approaches for stream partitioning. Finally, we describe existing approaches to distributed CER and their limitations.

Chapter 3

Preliminaries

In this section, we introduce the formal background that support our study. First, we introduce the concepts behind *distributed computing*. Secondly, we describe CEQL and give a formal description. Thirdly, we briefly discuss *selection strategies*. Lastly, we introduce the computational model CEA.

3.1 Distributed computing

A *distributed system* is a system whose components are located on different networked computers that communicate to each other by message passing in order to achieve a common goal. The main three characteristics of a distributed system are: concurrency of the components, lack of global memory and clock, and tolerance to failure of individual components [40]. Nowadays, the term is used in a much wider sense, even referring to autonomous processes that run on the same physical computer and interact with each other by exchanging messages. In our work, we do not make a distinction on whether the system operates on a cluster of networked computers or in a single multi-core computer.

A *distributed program* is composed of an ordered-set of n asynchronous processes $\mathcal{P} = \{P_1, P_2, \ldots, P_n\}$. For a process P_i with $1 \leq i \leq n$, define its *index*, denoted index(P_i), as index(P_i) $= i \in \mathbb{N}$. The index of a process can be used as a *unique* identifier. The processes do not share a global memory and communicate solely by passing messages. Process execution and message transfer

are asynchronous. Without loss of generality, we assume that each process is running on a different processor. Let C_{ij} denote the channel from process P_i to process P_j and let m_{ij} denote a message sent by P_i to P_j. The message transmission delay is finite and unpredictable [40].

3.2 Complex event logic

Events, complex events, and valuations. Fix a set of *event types* $\mathbf{T}$ (e.g., `H` and `T`), a set of *attribute names* $\mathbf{A}$ (e.g., *id*, and *val*), and a set of *data values* $\mathbf{D}$ (e.g., integer values, string values, 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 $t(a) \in \mathbf{D}$ the value of the attribute $a \in \mathbf{A}$ assigned by t, and $t(type) \in \mathbf{T}$ 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 $C(data)$ to denote D, $C(time)$ to denote the time-interval $[i, j]$, and $C(start)$ and $C(end)$ for i and j, respectively.

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 $\mathbf{X}$. We write $V(time)$, $V(start)$, and $V(end)$ 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: $C_V(time) = V(time)$ and $C_V(data) = \bigcup_{X \in \mathbf{X}} V(X)$. The semantics of CEQL will be defined in terms of valuations, which are subsequently transformed into complex events as explained.

Predicates. A (unary) *predicate* is a possibly infinite set P of data-tuples. A data-tuple t *satisfies* predicate P, denote $t \Vdash P$, 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 $T \Vdash P$, if, and only if, $\underset{t \in T}{\forall} t \vDash P$.

CEQL. *Complex Event Query Language (CEQL)* is a practical CER language based on *Complex Event Logic (CEL)*, which is a formal logic that is built from the common operators in the literature of CER and whose expressiveness and complexity have been diligently studied in [17, 18, 16]. We introduce the most relevant features of CEQL by means of an example.

Example 3.1. *We retake the previous example which aims on the detection of fires in a stream produced by a network of wireless sensors placed in a warehouse. Suppose that we are interested in all n-tuples of* `T` *events where the first has temperature below* 30 *Celsius degrees and the rest has temperature above* 30 *Celsius degrees, partitioned by the zone of the warehouse where the sensor is placed. The query from Figure 3.1 expresses this in CEQL.*

```
SELECT *
FROM warehouse
WHERE (T as t1; T+ as ts)
FILTER t1[val < 30]
  AND ts[val > 30]
PARTITION BY id
WITHIN 5 minutes
```

FIGURE 3.1: CEQL query on a wireless sensors network stream.

The `FROM` *clause indicates the input streams. The* `WHERE` *clause indicates the pattern of events that need to be matched in the stream. The pattern can be any* unary *CEL expression [16]. In our query, the pattern* `T as t1; T+ as ts` *indicates that we want to capture all complex events that consist of an event type* `T` *followed by an arbitrary number of events of type* `T`*. In particular, the operator (;) indicates sequencing and the operator (+) indicates non-empty repeated sequencing (i.e.* kleene plus*). We want to remark that sequencing in CEL is non-contiguous. Consequently, the* `T` *events do not need to be contiguous — there might be other events in between. The* `FILTER` *clause allows to filter events by unary predicates. The clause* `PARTITION BY` *allows to express correlation among events in the form of equi-joins. The* `WITHIN` *clause specifies the time-window. In our query, the time between the first event* `T` *and*

the last event `T` *must be within 5 minutes. Finally, the* `SELECT` *clause allows to project the result.*

Next, we give the formal syntax and semantics of CEQL.

```
SELECT         [selection strategy] <list of variables>
FROM           <list of streams>
WHERE          <CEL formula>
(PARTITION BY <list of attributes>)?
(WHITHIN       <time>)?
```

We remark that CEL includes `FILTER`, and so CEQL does not need a separate `FILTER` clause. The `WHERE` clause expects a pattern written in Complex Event Logic (CEL) [16], whose abstract syntax is represented by the following grammar:

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

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

The semantics of CEQL is as follows. A CEQL query first evaluates its `FROM` clause, then its `PARTITION BY` clause, and subsequently its `WHERE, SELECT, WITHIN` clauses, in that order. The `FROM` clause specifies the list of streams. All these streams are logically merged into a single stream S. The optional `PARTITION BY` clause logically partitions this stream into multiple substreams $S_1, S_2 \ldots$ and executes the `WHERE-SELECT-WITHIN` clauses on each substream. The union of the outputs generated for each substream constitute the final output. Note, the different streams could be evaluated in parallel. CEQL's `WHERE` and `FILTER` clause are derived from the semantics of CEL in Figure 3.2. Specifically, given a stream $S = t_0 t_1 t_2 \ldots$, 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. The `AS` clause is a variable assignment that takes an existing valuation $V \in [\![\varphi]\!](S)$ and extends it by gathering all positions $\bigcup_Y V(Y)$ in variable X, keeping all other variables as in V. The `FILTER` $X[P]$ clause retains only those valuations for which the content of variables X satisfies predicate P. The `OR` clause takes the union of two sets of valuations. The sequencing operator $(\varphi;\varphi)$ uses the time-interval for capturing all pairs of valuations in which the first

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

FIGURE 3.2: The semantics of CEL.

is chronologically followed by the second. The semantics of iteration ($\varphi+$) is defined as the application of sequencing (;) one or more times over the same formula φ. The projection π_L modifies valuations by setting all variables that are not in L to empty. Hence, `WHERE-FILTER` return a set of valuations when evaluated over a stream. The `SELECT` clause, if it does not mention a selection strategy (explained later), corresponds to a projection in CEL. Finally, if ϵ is a time-interval, then the `WITHIN` clause operate on the resulting set of valuations $[\![\varphi \text{ WITHIN } \epsilon]\!](S) = \{V \in [\![S]\!] \mid V(end) - V(start) \leq \epsilon\}$.

Complex event semantics. The complex event semantics of CEL and CEQL is obtained by first evaluating the query under the valuations semantics, and then removing variables. That is, if φ is a CEL formula or CEQL query, its complex event semantics $[\![\varphi]\!](S)$ is defined $[\![\varphi]\!](S) := \{C_V \mid V \in [\![\varphi]\!](S)\}$.

3.3 Selection strategies

Selection strategies (or *selectors*) are unary operators over CEL formulas that restrict the set of results and speed up query processing. We present four

selection strategies [17, 16]: *strict* (STRICT), *next* (NXT), *last* (LAST) and *max* (MAX). STRICT and NXT are motivated by the *strict-contiguity* and *skip-till-next-match* selector strategies proposed by SASE [10], while LAST and MAX are useful selection strategies from a semantic viewpoint [16]. Next, we describe and formally specify selection strategy MAX, as it is relevant in our work, and refer the interested reader to [16] for a definition and discussion of the other selection strategies. For the sake of the discussion, we will define the *support* of a valuation V as the set of all positions appearing in the range of V, i.e., $sup(V) = \bigcup_{X \in \mathbf{X}} V(X)$.

MAX. This selection strategy keeps the maximal complex events in terms of set inclusion, which could be naturally more useful because these complex events are the *most informative*. Formally, given a CEL formula φ we say that $V \in [\![\text{MAX}(\varphi)]\!](S)$ holds iff $V \in [\![\varphi]\!](S)$ and for all $V' \in [\![\varphi]\!](S)$, if $sup(V) \subseteq sup(V')$, then $sup(V) = sup(V')$ (i.e., V is maximal with respect to set containment). For example, given a CEL query φ, if $[\![\varphi]\!](S)$ returns $\{3, 6, 7\}$, $\{3, 4, 7\}$, and $\{3, 4, 6, 7\}$. Then, MAX(φ) will only return $\{3, 4, 6, 7\}$, which is the maximal complex event.

3.4 Computational model

As explained in Section 3.2, evaluating a CEQL query corresponds to evaluating the query's `SELECT-WHERE-WITHIN` clauses on either a single stream, or multiple different substreams. In our work, the `SELECT-WHERE` part of a query is compiled into a *Complex Event Automaton* [17, 16], which is a form of finite automaton that produces complex events. Our evaluation algorithm is then defined in terms of CEA: it takes as input a CEA $\mathcal{A}$, the time-window ϵ specified in the `WITHIN` clause, and a stream S, and uses this to compute $[\![\varphi]\!](S)$. 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, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of final states. We will denote transitions in Δ by $q \xrightarrow{P/m} q'$. A *run* of $\mathcal{A}$ over stream S from positions i to j is a sequence $\rho := q_i \xrightarrow{P_i/m_i} q_{i+1} \xrightarrow{P_{i+1}/m_{i+1}} \dots \xrightarrow{P_j/m_j} q_{j+1}$ such that q_i is the initial state of $\mathcal{A}$ and for every $k \in [i, j]$ it holds that $q_k \xrightarrow{P_k/m_k} q_{k+1} \in \Delta$ and $t_k \vDash P_k$. A run ρ is *accepting* if $q_{j+1} \in F$. An accepting run of $\mathcal{A}$ over S from i to j naturally defines the complex event

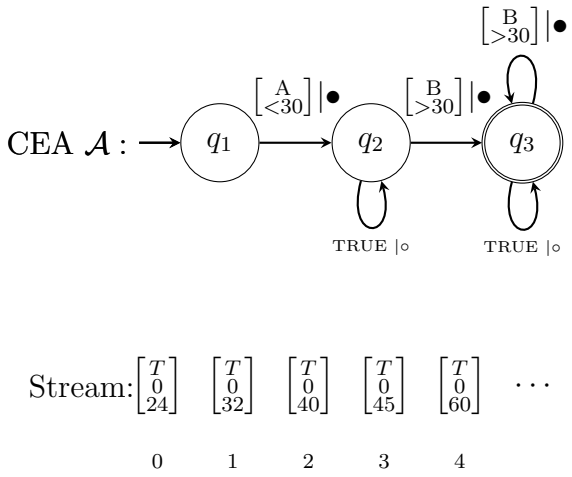


FIGURE 3.3: A CEA representing the query from Figure 3.1 and an example of stream.

$C_\rho := ([i, j], \{k \mid i \leq k \leq j \wedge m_k = \bullet\})$. Finally, we define the semantics of $\mathcal{A}$ over a stream S as $[\![\mathcal{A}]\!](S) := \{C_\rho \mid \rho \text{ is an accepting run of } \mathcal{A} \text{ over } S\}$.

Example 3.2. *In Figure 3.3 we show the compilation of the query from Figure 3.1 into an equivalent CEA $\mathcal{A}$. We depict predicates by listing, in array notation, the event type, and the constraint on the temperature attribute. The initial state is q_1 and there is only one final state: q_4. The figure also includes an example stream S, where the values correspond to the event type, the identifier attribute, and the temperature attribute, in that order.*

The usefulness of CEA comes from the fact that CEL can be translated into CEA [17, 16]. 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.

Theorem 3.1 (Theorem 1 [8])**.** *For every CEL formula φ we can construct a CEA $\mathcal{A}$ of size linear in φ such that for every ϵ:*

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

Our evaluation algorithm will compute the right-hand side of this equation. It requires 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 state q, if $P_1 \cap P_2 \neq \emptyset$ then $m_1 \neq m_2$. To put it another way, an event t may trigger both transitions at the same time only if one transition marks the event, but the other does

not. In [17, 16], it was shown that any CEA can be I/O-determinized, with a possibly exponential blow-up in the size of the automaton.

3.5 Chapter summary

In this chapter we have presented the preliminary work that is required for our work. First, we introduced the concept of distributed computing. Then, we described the syntax and semantics of CEQL and CEL. Afterwards, we presented the selection strategies, and explained the semantics of MAX. Finally, we introduced the computational model CEA, which is used to evaluate CEQL.

Chapter 4

Distributed CER

In this chapter we propose a novel framework for distributed CER based on the following observation: given the efficient evaluation of CEA in a centralized manner [16, 8], it is hard to foresee scenarios where it can benefit from distribution. Indeed, both automata distribution approaches (i.e., query partitioning and pipelining) will incur a big overhead in terms of network communication during the evaluation process to synchronize the compact data structure representing the partial matches, and the enumeration process. However, as previously discussed, CEA is very limited in terms of filtering capabilities, allowing only *unary* predicates. Thus, one might wonder how to include the evaluation of more complex filters over non-unary predicates such as binary predicates (e.g., equi-joins like `h1[id] = t2[id]`), or second-order predicates (e.g., the sequence of `T[val]` must monotonically increase). In [18], it is discussed that CEL and CEA are equivalent in expressive power when CEL is restricted to unary predicates, but incomparable in general. Thus, non-unary CEL, in general, cannot be compiled into an equivalent CEA. However, one could split the pattern matching process and the filtering in CER. In other words, we would maintain the generation of complex events in CEA, but leverage on a distributed framework for complex filtering.

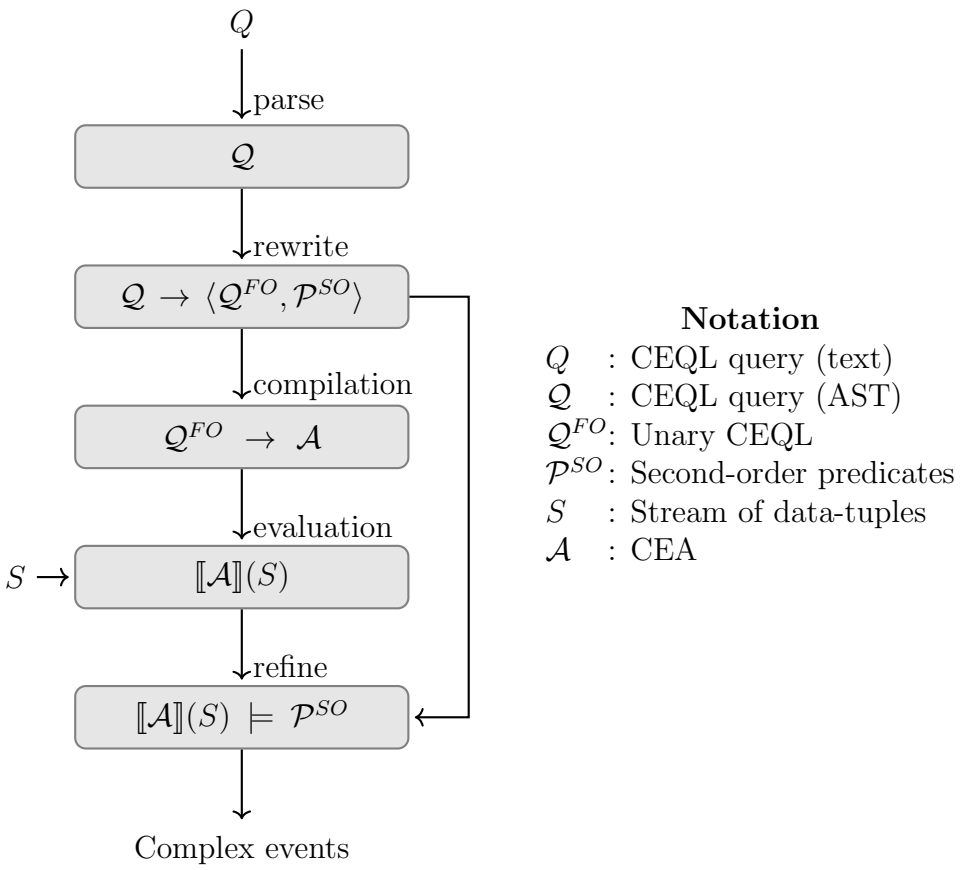


FIGURE 4.1: Distributed CER framework.

4.1 Distributed CER framework

In this section, we describe our proposal of a framework for distributed CER (see Figure 4.1). Our framework uses CEQL as its query language, allowing n-ary CEL predicates inside the `WHERE` clause. It receives a stream of data-tuples S and a query Q as text. The query is parsed using a context free grammar parsing algorithm (e.g., Earley's algorithm [41]). The *rewrite algorithm* is applied to the resulting abstract syntax tree of the query $\mathcal{Q}$. The rewrite algorithm translates the non-unary CEL predicate of a CEQL query into a unary CEL predicate $\mathcal{Q}^{FO}$, and a data structure encoding the corresponding non-unary predicates $\mathcal{P}^{SO}$ to be applied later. The resulting unary CEQL query $\mathcal{Q}^{FO}$ is compiled into a CEA $\mathcal{A}$ by [16, Theorem 6.2]. Then, this CEA is evaluated by the efficient evaluation algorithm presented in [8], which takes, under *data complexity*, constant time per input event to maintain a data structure representing the set of partial complex events. Notice, the result of this evaluation $[\![\mathcal{A}]\!](S)$ is not yet a valid output since we still need to apply the

`FILTER` clause for the non-unary predicates. The *refine algorithm* is applied distributedly to the set of complex events $[\![\mathcal{A}]\!](S)$. The refine algorithm uses the non-unary predicates $\mathcal{P}^{SO}$ to filter the set of complex events (denoted as $[\![\mathcal{A}]\!](S) \models \mathcal{P}^{SO}$ in Figure 4.1). Finally, the set of complex events corresponding to the input query is enumerated.

Note. Due to time constraints, we implemented a limited rewrite and refine algorithm which considers specific kind of queries. Implementing a generic rewrite and refine algorithm is outside of the scope of this thesis and it is left for future work.

It is clear that the refine algorithm has to be applied distributedly in order to achieve optimal scalability and performance. However, there are still two relevant decisions left: where do you compile and process the CEA and how do you distribute the resulting complex events among the processing units. We propose two different architectures that take different approaches to previous questions. The first architecture, called *Distributed CER Engine (DCERE for short)*, compiles and executes the CEA in a centralized manner and distributes the resulting complex events to each processing unit. The second architecture, called *Distributed CORE (DCORE for short)*, broadcast the events, and compiles and executes the CEA on each processing unit. We remark that implementing the latter is more challenging than implementing the former. For the former, we could use any state-of-the-art sequential evaluation algorithm (e.g. [16], or [8]). However, the latter requires an efficient distributed evaluation algorithm for CEA, that to the best of our knowledge, is yet to be outlined.

The rest of this chapter is dedicated to explaining in detail both alternatives.

4.2 Distributed CER Engine

In this section we discuss *Distributed CER Engine (DCERE)*, an implementation of the framework proposed in Section 4.1. This implementation (illustrated in Figure 4.2) is built on top of a distributed actor model. We emphasize that actor models can be used to either encode parallelism by considering each core of a processor as an actor, or distributed programming by considering each processing unit in the network as an actor. Indeed, in the

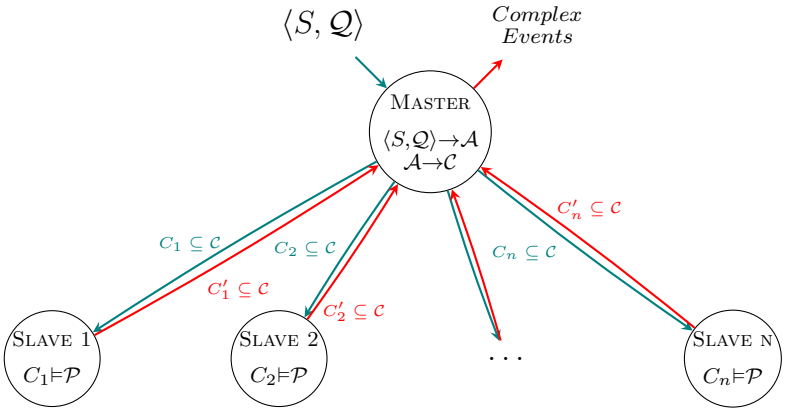


Figure 4.2: Distributed CER Engine architecture.

implementation from Chapter 6, we consider as many actors as the product of the number of processing units in our cluster by the number of cores of each processor.

We will differentiate between two roles of actors: master actors and slave actors. There will be a single master actor in the cluster, and as many slave actors as processing units left. The master actor will act as the leader of the cluster and will orchestrate the rest of the slave actors. The master actor receives a stream of data-tuples and a query as an input. It parses and rewrite the input query to unary CEL. Then, it compiles the query to CEA, and evaluate it to retrieve the corresponding set of complex events. For the compilation and evaluation, we will use CORE [8] which guarantees constant update of the data structure and output-linear delay enumeration. Then, we apply one of the selection strategies (explained below) to distribute the set of complex events among the slave actors. Asynchronously and independently, each processing unit receives its corresponding subset of complex events and it applies the refine algorithm to the complex events. Afterwards, once the refine algorithm is finished, the actor returns the resulting subset of complex events to the master node. Finally, once the master has received the output of all processing units, it executes the configured response (e.g., write the complex events to disk, send an email, activate an alarm).

4.2.1 Distribution Strategies

In this section we describe the distribution strategies used by DCERE to distribute the set of complex events among the set of actors on the distributed system. We have considered distribution strategies ranging from *load-balancing algorithms* to *stream partitioning algorithms*. Additionally, we have proposed our own novel distribution strategy specific to the distribution of complex events, called *Maximal Complex Event Enumeration (MCEE)*. We empirically compare the proposed selection strategies in Chapter 6. Furthermore, the interested reader may find the algorithms in Appendix A.2.

Round Robin. *Round Robin (RR)* is the most basic load-balancing strategy. It consist of blindly routing the complex events to the actors in a circular fashion. Although simple, it works well in practice.

Power of Two Choices. The *Power of Two Choices (PoTC)* [42] is a randomized load balancing algorithm introduced in *queue theory*. Given n actors, for each each complex event, we randomly pick d actors. Finally, from those d choices, we pick the one with lowest workload. Surprisingly, having $d = 2$ choices leads to an exponential improvement in the load-balance, whereas $d > 2$ choices is only constant factor better than $d = 2$.

Exact Search. *Exact Search (ES)* is a load balancing algorithm equivalent to PoTC when $d = n$. This distribution strategy was designed taking into account that computing the static load of an actor is cheap, while computing the dynamic load is not. Therefore, for each complex event, we query the static load in constant time of all the actors of the system, and send the complex event to the actor with lowest workload.

Maximal Complex Event Enumeration. *Maximal Complex Event Enumeration (MCEE)* is a load balancing algorithm specific to CER. The idea behind MCEE is to take advantage of the selection strategy MAX to retrieve only the *maximal complex events* that are the most *informative*, distribute them, and retrieve all complex events included in the maximal complex events. One of the challenges of this approach is the choice of an algorithm to distribute maximal complex events. In the absence of disjunctions or iterations in a query, queries produce few maximal complex events. Hence, enumerating and filtering *whole* maximal complex events in single nodes will incur in skewness in the load balance, which degrades performance. Another challenge

of this approach is finding the way to avoid duplicated outputs. Example 4.1 illustrates this problem.

Example 4.1. *We continue our example of detecting fires in a warehouse to illustrate that a naïve implementation based on* MCEE *generates duplicates. Given a query $\mathcal{Q}$ with pattern* `(T+; H+)` *and stream $S = \{T_1, H_1, T_2, H_2\}$. Query $\mathcal{Q}$ evaluated under selection strategy* MAX *over stream S produces the following complex events: $\{T_1, H_1, H_2\}$, and $\{T_1, T_2, H_2\}$. The first maximal complex event includes complex events: $\{T_1, H_1\}$, $\{T_1, H_2\}$, and $\{T_1, H_1, H_2\}$. The second maximal complex event includes: $\{T_1, H_2\}$, $\{T_2, H_2\}$, and $\{T_1, T_2, H_2\}$. Notice, complex event $\{T_1, H_2\}$ has been outputted twice.*

A priori, this distribution strategy seems to increase the performance of the recognition process: (1) the automata enumerates a smaller set of complex events, (2) less data has to be sent through the network incurring in a smaller communication overhead, and (3) the possibly exponential cost of the enumeration is distributed among the $n - 1$ slaves nodes. However, under in-depth analysis, (1) and (3) do not hold:

> **(1)**. Theorem 7.2 [16] shows that the compilation of the selection strategies has an exponential blow-up in the size of the automaton. In other words, the number of states and transitions increases exponentially resulting in a larger evaluation time for CEA.
>
> **(3)**. To guarantee duplicate-free results the algorithm needs to enumerate all complex events, *including the duplicated ones*, and them. This incurs, per complex event, in a linear factor overhead in the size of the complex event that undermines the gains from the distribution.

For completeness, we implemented this selection and empirically verified our hypothesis that this strategy does not perform well in practice.

4.3 Distributed CORE

In this section we discuss *Distributed CORE (DCORE)*, an implementation of the framework proposed in Section 4.1. This implementation (illustrated

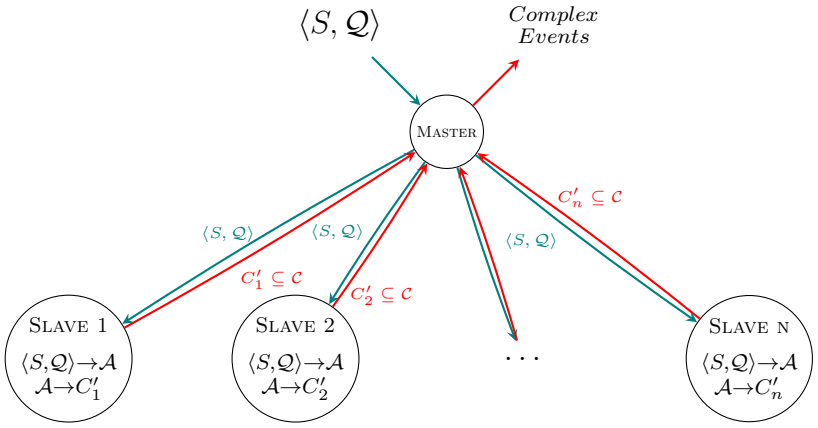


Figure 4.3: Distributed CORE architecture.

Figure 4.3) is also built on top of a distributed actor model, but the master actor plays a secondary role. It is only used to retrieve the complex events from the rest of the slave actors in order to execute the configured response. Indeed, if the response can be executed on the slaves actors (e.g., writing to an output stream), then the master actor can be removed from the system. We assume that the system can duplicate the input data. Otherwise, assume that the master node receives as an input a stream and a query, and *broadcasts* them to the rest of the actors.

The following steps are all execute asynchronously and independently on each node of the distributed system, and there is no communication between the nodes throughout this process. Each node in the system: receives an input stream and a query, parses and rewrites the input query to unary CEL, compiles the query to CEA, and finally evaluates the CEA, applies the refine algorithm and enumerates the resulting complex events.

In this implementation, the refine algorithm is executed at the same time as the enumeration process (corresponding to $\mathcal{A} \to C'_1$ in Figure 4.3), not requiring an additional traversal of the complex events. In other words, the output of the evaluation algorithm is already the output corresponding to the complex events captured by our query.

We emphasize that our system needs to evaluate, filter and enumerate disjoint subsets of complex events; otherwise, this implementation would be equivalent to a sequential version, but slower. For this reason, we require an efficient distributed evaluation algorithm for CEA. But, as far as we are concerned,

such an algorithm does not exist. For this reason, we devote Chapter 5 to the design of such an algorithm.

4.4 Chapter summary

In this chapter we presented a framework for distributed CER. Then, we discussed how a distributed framework for automaton-based CER would look like, and described our proposal for such a framework. Later, we introduced DCERE, an implementation of our framework and described several distribution strategies. Finally, we introduced DCORE, another implementation of our framework, and motivated the search for a distributed evaluation algorithm for CEA.

Chapter 5

Distributed evaluation algorithm

In this chapter, we propose an *efficient* and *distributed* evaluation algorithm for CEA, heavily inspired by [8], that computes the set $[\![\mathcal{A}]\!]^\epsilon(S) := \{C \mid C \in [\![\mathcal{A}]\!](S) \wedge C(end) - C(start) \leq \epsilon\}$ where $\mathcal{A}$ is a CEA, ϵ a time window, and S a stream. Actually, it will compute this set *incrementally*: at every position j in the stream, it outputs the set $[\![\mathcal{A}]\!]^\epsilon_j(S) := \{C \mid C \in [\![\mathcal{A}]\!]^\epsilon(S) \wedge C(end) = j\}$.

The evaluation algorithm is executed on a *distributed system* composed by independent processes $\mathcal{P}$. The processes do not share a global memory and communicate solely by passing messages. Each asynchronous process incrementally updates a data structure that *compactly* represents partials outputs. Whenever a new tuple arrives, it takes *constant time* (in data complexity [43]) to update the compact data structure. Moreover, the distributed system may enumerate *cooperatively* without message passing, at each position j, the complex events of $[\![\mathcal{A}]\!]^\epsilon_j(S)$, one by one, and without duplicates. During the enumeration, each process enumerates at most $\frac{|[\![\mathcal{A}]\!]^\epsilon_j(S)|}{|\mathcal{P}|}$ complex events with *output-linear delay* after printing the first complex event. This means that the time required to print the following complex events is linear in the size of the complex event being printed. We remark that our algorithm is asymptotically optimal: any evaluation algorithm needs to inspect every input event and enumerate the query's answers.

It might seem suboptimal to have all processes evaluating the CEA and updating the data structure. Surprisingly, we show that it is not the case. We

describe an alternative implementation, where the CEA and the compact data structure is only evaluated in one of the processes, and show that our implementation is asymptotically equivalent. The algorithm is as follows: we assign a process (e.g. P_1) to ingesting events and updating the data structure representing the partials outputs. Once a complex event is found, we distribute the data structure among the rest of the processes and enumerate the complex events on each. If implemented in a naïve way, the update algorithm may take linear time (instead of constant time), because distributing the tECS takes time proportional to its size which is linear in the size of the stream. However, if the distribution of the tECS is done incrementally on each input tuple, then each process will need to keep a copy that has to be incrementally updated and it would take, at least, constant time to update this copy. Furthermore, each process will need to keep all the auxiliary data structures used in the evaluation of the CEA. We emphasise that this alternative algorithm is asymptotically equivalent to our algorithm. In fact, our algorithm performs better in practise because it requires no communication among the processes.

The rest of the chapter is structured as follows. In Section 5.1 we define the data structure and its operations. In Section 5.2 we define auxiliary data structures used by the evaluation algorithm. In Section 5.3 and 5.4 we describe the evaluation algorithm and discuss its complexity. We conclude with a summary of the main points discussed in this chapter.

5.1 The data structure

The data structure is called *timed Enumerable Compact Set* (tECS). 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 `left(u)` and the right child `right(u)`. Every non-union node n is labelled 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 `pos(n)` for the label of non-union node n and `next(o)` for the unique child of output node o. For a node n, define its *descending-paths*, denoted `paths(n)`, recursively as follows: if n is a bottom node, then `paths(n)` = 1; if n is an output node, then `paths(n) = paths(next(n))`; if

n is a union node, `paths(n) = paths(left(n)) + paths(right(n))`. Every node n carries paths(n) as an extra label; thus the descending-paths can be retrieved in constant time. To simplify presentation in what follows, we write nodes of any kind as n, bottom, output and union nodes as b, o, u, respectively, and we denote the sets of bottom, output and union nodes by N_B, N_O and N_U, respectively.

A tECS represents sets of open complex events. An *open complex event* is a pair (i, D) where $i \in \mathbb{N}$ is the starting position and D is a finite subset of $\{i, i+1, \ldots\}$ positions. An open complex event is almost a complex event, but 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 complex events.

Let $\bar{p} = n_1, n_2, \ldots, n_k$ be a *full-path* in $\mathcal{E}$ such that n_k is a bottom node. Then $\bar{p}$ represents the open complex event $[\![\bar{p}]\!]_{\mathcal{E}} = (i, D)$, where $i = pos(n_k)$ is the label of bottom node n_k and D are the labels of the other non-union nodes in $\bar{p}$.

Given a node n, $[\![\mathrm{n}]\!]_{\mathcal{E}}$ is the set of open complex events represented by n and consists of all open complex events $[\![\bar{p}]\!]_{\mathcal{E}}$ with $\bar{p}$ a full-path in $\mathcal{E}$ starting at n.

To enumerate, at every position j, the set of complex events $[\![\mathcal{A}]\!]^{\epsilon}_{j}(S)$, we will need to enumerate, for some nodes in $\mathcal{E}$, the set $[\![\mathrm{n}]\!]^{\epsilon}_{\mathcal{E}}(j) := \{([i, j], D) \mid (i, D) \in [\![\mathrm{n}]\!]_{\mathcal{E}} \wedge j - i \leq \epsilon\}$ i.e. all open complex events represented by n that, when closed with j, are within a time window of size ϵ.

In order to enumerate the set of complex events $[\![\mathrm{n}]\!]^{\epsilon}_{\mathcal{E}}(j)$, one by one, without duplicates, and with output-linear delay, it will be necessary to impose three restrictions on the structure of the tECS $\mathcal{E}$:

Time-ordered. For a node n define its *maximum-start*, denoted max(n), as $\mathrm{max(n)} = \mathrm{max}(i \mid (i, D) \in [\![\mathrm{n}]\!]_{\mathcal{E}})$. Then, a tECS is *time-ordered* if (1) every node n includes max(n) as an attribute that can be access in constant time, and (2) for every union node u it holds that $\mathrm{max(left(u))} \geq \mathrm{max(right(u))}$. Needless path traversals are avoided on a time-ordered tECS: we always check first that $j - \mathrm{max(n)} \leq \epsilon$, and when we traverse a union node u, we always visit left(u) before right(u) and we only visit right(u) if $j - \mathrm{max(right(u))} \leq \epsilon$.

K-bounded. For a node n define its *output-depth*, denoted odepth(n), recursively as: if n is a non-union node, then odepth(n) = 0; otherwise, odepth(n) = odepth(left(n)) + 1. Then, a $\mathcal{E}$ is *k-bounded* if odepth(n) $\leq k$ for every node n. The k-boundedness restriction is necessary to preserve output-linear delay; otherwise the corresponding full-path $\bar{p}$ of a node n in $\mathcal{E}$ may contain an unbounded number of union nodes to visit before reaching the bottom node violating the output-linear delay.

Duplicate-free. A tECS $\mathcal{E}$ 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 holds that $[\![\bar{p}]\!]_{\mathcal{E}} \neq [\![\bar{q}]\!]_{\mathcal{E}}$.

Remember that the purpose of constructing tECS $\mathcal{E}$ is to be able to enumerate the set $[\![\mathcal{A}]\!]^{\epsilon}_{j}(S)$ at every position j. If $\mathcal{E}$ is time-ordered, k-bounded for $k = 3$, and duplicate-free, then Theorem 5.1 ensures that we enumerate set $[\![\mathcal{A}]\!]^{\epsilon}_{j}(S)$.

Theorem 5.1. *Let $\mathcal{E}$ be a time-ordered tECS, n a duplicate-free node of $\mathcal{E}$, ϵ a time-window, $\mathcal{P}$ the set of all processes. Let C_p be the output of Algorithm 2 on process $p \in \mathcal{P}$. Then, $\bigcup_{P \in \mathcal{P}} C_P = [\![n]\!]^{\epsilon}_{\mathcal{E}}(j)$.*

In Section 5.4, we describe Algorithm 2, and show its correctness.

Operations on tECS. We define three operations on $\mathcal{E}$. In order to ensure that newly created nodes are 3-bounded, we require that the argument nodes of these operations are safe. A node is *safe* if it is a non-union node or if both odepth(u) = 1 and odepth(right(u)) ≤ 2. All three operations on tECS return safe nodes.

b $\leftarrow$ `new-botom`(i). The first method adds a new bottom node b labelled by $i \in \mathbb{N}$ to $\mathcal{E}$.

$o \leftarrow$ `extend`(n, j). The second method adds a new output node o to $\mathcal{E}$ with pos(o) = j and next(o) = n.

u $\leftarrow$ `union`($\mathrm{n}_1, \mathrm{n}_2$). The third method returns a union node u such that $[\![u]\!]_{\mathcal{E}} = [\![\mathrm{n}_1]\!]_{\mathcal{E}} \cup [\![\mathrm{n}_2]\!]_{\mathcal{E}}$. Both n_1 and n_2 are required to be safe and $\max(\mathrm{n}_1) = \max(\mathrm{n}_2)$. If n_1 is a non-union then a new union node u is created which is connected to n_1 and n_2 as shown in Figure 5.1a. If n_2 is a non-union node, then u is created as shown in Figure 5.1b. When n_1 and n_2 are union nodes: if $\max(\mathrm{right}(\mathrm{n}_1)) \geq \max(\mathrm{right}(\mathrm{n}_2))$, three new union nodes, u, u_1, u_2 are added

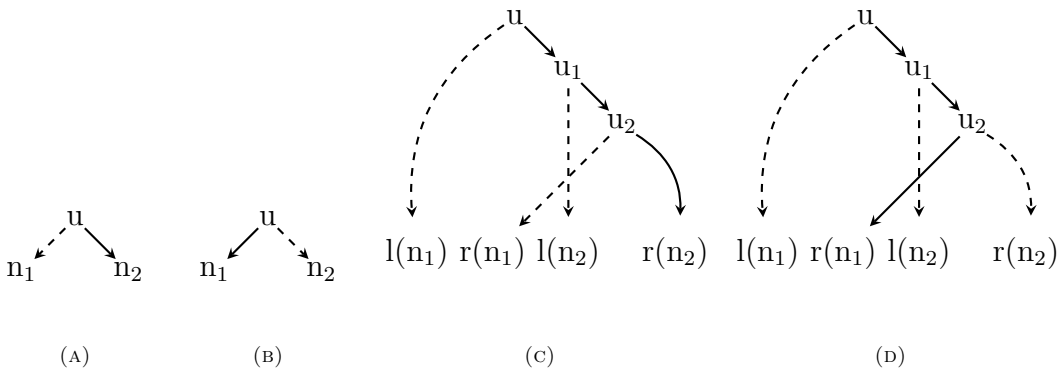


FIGURE 5.1: Visualisation of the four cases of method union(u). The u are union nodes, where the dashed and bold arrows point to the left and right node, respectively.

and connected as show in Figure 5.1c; otherwise, as in Figure 5.1d. Notice, the output-depth of the created union nodes is at most 3.

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

5.2 Auxiliary data structures

In this section, we introduce two auxiliary data structures that are going to be used to incrementally maintain $\mathcal{E}$ during the evaluation of the algorithm.

Union-lists and its operations. A *union-list* is a non-empty sequence of safe nodes, denoted $\text{ul} = \text{n}_0, \text{n}_1, \ldots, \text{n}_k$ such that n_0 is a non-union node, $\max(\text{n}_0) \geq \max(\text{n}_i)$, and $\max(\text{n}_i) > \max(\text{n}_{i+1})$ for every $0 < i \leq k$. We define three operations on union-lists, all of them taking safe nodes as arguments.

ul ← `new-ulist(n)`. The first method creates a new union-list with a single non-union node n.

`insert(ul, n)`. The second method mutates, in situ, the union-list $\text{ul} = \text{n}_0, \ldots, \text{n}_k$ by inserting a safe node n such that $\max(\text{n}) \leq \max(\text{n}_0)$. If there is a $i > 0$ such that $\max(\text{n}_i) = \max(\text{n})$, then it replaces n_i in ul by the result of $\text{union}(\text{n}_i, \text{n})$. Otherwise, we consider two cases: if $\max(\text{n}) = \max(\text{n}_0)$, then n

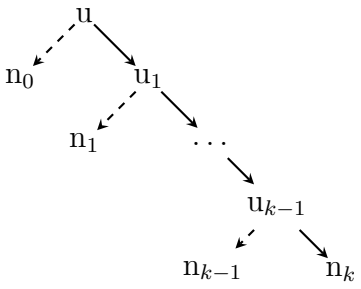


FIGURE 5.2: Visualisation of method merge(ul). The u are union nodes, where the dashed and bold arrows point to the left and right node, respectively.

is inserted after n_0; otherwise, n is inserted between n_i and n_{i+1} with $i > 0$ such that $\max(n_i) > \max(n) > \max(n_{i+1})$.

$u \leftarrow$ `merge(ul)`. The third method takes a union-list ul and returns a node u such that $[\![u]\!]_{\mathcal{E}} = [\![n_0]\!]_{\mathcal{E}} \cup \ldots \cup [\![n_k]\!]_{\mathcal{E}}$. If $k = 0$, then $u = n_0$, or else, we add k union nodes to $\mathcal{E}$, and connect them as shown in Figure 5.2. Since n_0 is a non-union node, $\text{odepth}(u) \leq 1$. And, because all n_i are safe, $\text{odepth}(u_i) \leq 2$. As a result, it is easy to see that u is safe. Furthermore, all of the new union nodes are time-ordered and 3-bounded.

We remark that all operations on union-lists take time linear in the length of ul.

Hash table and its operations. A *hash table* is an abstract data structure that maps keys to values by the use of a *hash function*. During the evaluation of the algorithm, we will use a hash table to map CEA states to nodes of union-lists. Let T be such a hash table, we write $T[q]$ for the union-list associated to state q and $T[q] \leftarrow ul$ for inserting or updating the union-list associated with state q. We define two methods on hash tables.

`keys(`T`)`. The first method returns the set of states q that have a union-list associated with. We write $q \in \text{keys}(T)$ for checking if q is present in T.

`ordered-keys(`T`)`. The second method returns keys(T) as a list sorted in the order in which keys have been inserted into T. If a key is insert multiple times, then it is the time of the first that is used for sorting.

We assume that hash table lookups and insertions take constant time, and iterating over has constant delay.

5.3 The evaluation algorithm

In this section, we present the evaluation algorithm. It receives as input a I/O deterministic CEA $\mathcal{A} = (Q, \Delta, q_0, F)$, a stream S, and time-window ϵ and may enumerate, on process p, a subset of $[\![\mathcal{A}]\!]^{\epsilon}_{j}(S)$ at every position j. We remark that when the evaluation Algorithm 1 is executed simultaneously on each process $p \in \mathcal{P}$, it may enumerate the set $[\![\mathcal{A}]\!]^{\epsilon}_{j}(S)$ at every position j. We next describe the algorithm and analyse its complexity.

Algorithm 1 incrementally maintains (1) a tECS $\mathcal{E}$ that represents the set of open complex events and (2) a set of *active states* of $\mathcal{A}$. A state $q \in Q$ is *active* at a stream position j if there is some run of $\mathcal{A}$ which is in state q at position j. The algorithm uses a hash table T that links active states of Q to union-list nodes to incrementally maintain the $\mathcal{E}$.

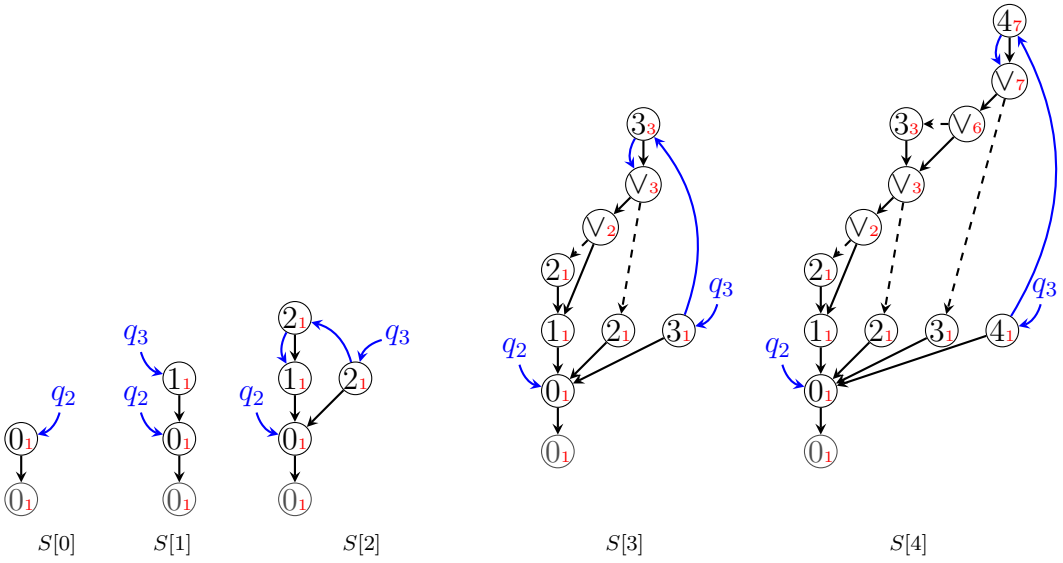


FIGURE 5.3: Illustration of Algorithm 1 on the CEA $\mathcal{A}$ and stream S of Figure 3.3.

During the description, the reader may find it helpful to refer to Figure 5.3, which illustrates how Algorithm 1 evaluates the CEA $\mathcal{A}$ of Figure 3.3 over the stream S of Figure 3.3. Each subfigure corresponds to the state after processing $S[j]$. The tECS is denoted in black, the attribute *paths* of each node is coloured red, and 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.

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

```
1  procedure Evaluation(A, S, ε, p)
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, p)
13
14 procedure ExecTrans(p, ul, t, j)
15     n ← merge(ul)
16     if q ← Δ(p, t, •) then
17         n' ← extend(n, j)
18         ul' ← new-ulist(n')
19         Add(q, n', ul')
20     if q ← Δ(p, t, ∘) then
21         Add(q, n, ul)
22
23 procedure Add(q, n, ul)
24     if q ∈ keys(T′) then
25         T′[q] ← insert(T′[q], n)
26     else
27         T′[q] ← ul
28
29 procedure Output(j, ε, p)
30     for p ∈ keys(T) do
31         if p ∈ F then
32             n ← merge(T[p])
33             Enumerate(n, ε, j, p)
```

Algorithm 1 consist of four procedures: Evaluation, ExecTrans, Add, and Output. The main procedure Evaluation starts by initialising the current stream position j and the hash table T (lines 2 and 3). We assume that yield(S) returns the next tuple in the stream S. We execute the `while` (lines 4-12): for every tuple t, we update j and initialise the hash table T'. The hash tables T and T' will hold the actives states at position $j-1$ and j, respectively. Lines 7-8 take into account that a new run may start at any position in the stream. For this, the algorithm creates a new union-list starting at position j and executes all transitions of initial state q_0 by calling ExecTrans. The `for` of lines 9-10 consider runs active at position $j-1$.

For this case, we iterate over all active states at position $j-1$, and execute all transitions by calling ExecTrans. Then, we replace T by T' to prepare for the next iteration and call Output in case we need to enumerate any complex event closed at position j.

Procedure ExecTrans 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 active runs that reached p. At line 15, it merges ul into a single node n. At line 16, we check if there is a marking transition $p \xrightarrow{P/\bullet} q$ in Δ such that $t \vDash P$. Recall that because CEA $\mathcal{A}$ is I/O deterministic there is at most one such transition. If there is such a transition, at lines 17-19, we extend all open complex events represented by n with the new position j and add them to $T[q]$. At line 20, we repeat the same but for non-marking transition $p \xrightarrow{P/\circ} q$. Note, this time we do not extend open complex events represented by node n because the transition is non-marking.

The procedure Add adds open complex events represented by node n to $T'[q]$. If $q \in \text{keys}(T')$ we have already reached q on iteration j (line 24), then we insert n at union list $T'[q]$ (line 25). Otherwise, we initialise the entry q of T' with the union-list representation of n (line 27)

The procedure Output enumerates a subset of all complex events in $[\![\mathcal{A}]\!]^{\epsilon}_{j}(S)$ at process p. The method is called at line 12, when T contains all active states at position j. We iterate over all active states $p \in \text{keys}(T)$ (line 30) and check if p is a final state (line 31). If the state is final, we merge the union list $T[p]$ into a node n in $\mathcal{E}$ and call Enumerate$(\mathcal{E}, \text{n}, \epsilon, j, p)$ where Enumerate is the algorithm of Theorem 5.1 described in Section 5.4.

This concludes the presentation of Algorithm 1. Now, we analyse its complexity. When a new tuple arrives, Algorithm 1 updates T, T', and $\mathcal{E}$ by means of methods in Section 5.1 and 5.2 which either take constant time or linear time with respect to the size of the union-list. For every position j, the length of every union list is bounded by the number of active states (see Appendix B.3 [8]). Then, because we iterate over all transitions in the worst case (line 9), and executing a transition takes time proportional to the length of the union-list, which is at most the number of states, we may conclude that the time for processing a new tuple is $\mathcal{O}(|Q| \cdot |\Delta|)$. This is constant under data complexity.

Finally, we analyse its correctness. Because Algorithm 1 builds $\mathcal{E}$ only through the methods of Section 5.1 and 5.2, we guarantee that 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 correctness of Algorithm 1.

Theorem 5.2. *After the j-th iteration of* Evaluation*, the union of the* Output *procedure on each process $p \in \mathcal{P}$ enumerates the set $[\![\mathcal{A}]\!]^{\epsilon}_{j}(S)$ with output-linear delay after enumerating the first complex event.*

The proof of Theorem 5.2 is technical and the reader may find it in [8].

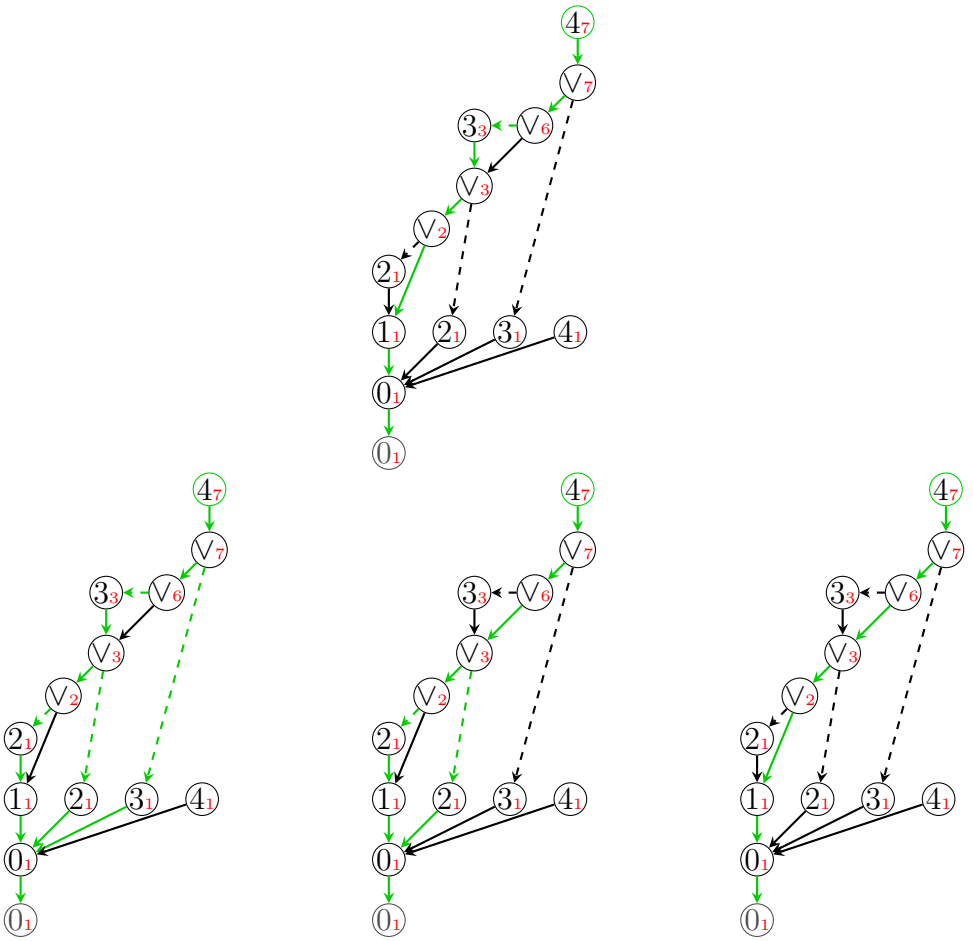
5.4 The Enumeration procedure

We provide Algorithm 2 and show that: (1) it enumerates a subset of complex events $S \subseteq [\![\mathrm{n}]\!]^{\epsilon}_{\mathcal{E}}(j)$ where $|S| = \mathcal{O}(\frac{[\![\mathrm{n}]\!]^{\epsilon}_{\mathcal{E}}(j))}{|\mathcal{P}|})$, and (2) it does so with output-linear delay after reaching the first complex event. Furthermore, we prove Theorem 5.1 to show that (3) the union of the enumeration of Algorithm 2 on each process $P \in \mathcal{P}$ corresponds to $[\![\mathrm{n}]\!]^{\epsilon}_{\mathcal{E}}(j)$.

We next describe how Algorithm 2 works. During the description, the reader may find it helpful to refer to Figure 5.4, which illustrates how Algorithm 2 enumerates the tECS $\mathcal{E}$ of Figure 5.3. In particular, how complex events $[\![4]\!]^{\epsilon}_{\mathcal{E}}(j)$ are enumerated by distributing the workload on each process. Assuming $|\mathcal{P}| = 3$, each subfigure depicts the enumeration of the paths assigned to each process. The tECS is denoted in black, the attribute *paths* of each node in red, and the traversed paths in green. Notice, each process traverses different paths but all paths are traversed at the end.

Algorithm 2 uses a stack *st* with common stack operations: `new-stack()` to create an empty stack, `push(st, e)` to add an element `e` at the top of the stack, and `pop(st)` to remove and return the element on the top of the stack. When the stack is empty, we will interpret $e \leftarrow pop(st)$ as `false`. We assume that stack operations can be performed in constant time.

Recall that $\mathcal{E}$ encodes the DAG $G_{\mathcal{E}} = (N, E)$ where N are the vertices, and E the edges that go from any union node u to left(u) and right(u), and from any output node o to next(o). For every node $n' \in N$, let $paths_{\geq\tau}(n')$ be all



(A) Process 0 (B) Process 1 (C) Process 2

FIGURE 5.4: Illustration of Algorithm 2 on the tECS $\mathcal{E}$ of Figure 5.3.

paths of $G_{\mathcal{E}}$ that start at n' and end at some bottom node b with $pos(b) \geq \tau$, and $paths_{\geq\tau,\sigma,\delta}(n')$ be a subset of $paths_{\geq\tau}(n')$ of size at most δ starting after path π_σ, where π_σ is the σ-th path from $\mathcal{E}$ starting at node n' such that $0 \leq \sigma \leq |\text{paths}(n')|$. It is clear that there exists an isomorphism between $[\![\text{n}]\!]^{\epsilon}_{\mathcal{E}}(j)$ and $paths_{\geq j-\epsilon}(n)$ i.e. for every complex event within a time window of size ϵ there exists exactly one path that reaches a bottom node b with $pos(b) \geq j - \epsilon$, and vice versa. Formally,

Theorem 5.3 ($[\![\text{n}]\!]^{\epsilon}_{\mathcal{E}}(j) \longleftrightarrow paths_{\geq j-\epsilon}(n)$)**.** *For every complex event within a time window of size ϵ there exists exactly one path that reaches a bottom node b with $pos(b) \geq j - \epsilon$, and vice versa.*

We defer the proof of Theorem 5.3 to Appendix A.1 to avoid disrupting the flow of the discussion.

Algorithm 2: Enumeration of $paths_{\geq\tau,\sigma,\delta}$(n).

```
1  procedure ENUMERATE(E, n, ϵ, j, p)
2      δ ← ⌈paths(n) / |P|⌉
3      σ ← index(p) · δ
4      st ← new-stack()
5      τ ← j − ϵ
6      if max(n) ≥ τ then
7          push(st,⟨n, ∅, σ, δ⟩)
8      while (n′, P, σ′, δ′) ← pop(st) do
9          while true do
10             if n′ ∈ N_B then
11                 output([pos(n′), j], P)
12                 break
13             else if n′ ∈ N_O then
14                 P ← P ∪ pos(n′)
15                 n′ ← next(n′)
16             else if n′ ∈ N_U then
17                 if max(right(n′)) ≥ τ then
18                     if paths(left(n′)) > σ′ then
19                         δ″ ← δ′ − max(0, paths(left(n′)) − σ′)
20                     else
21                         δ″ ← δ′
22                     σ″ ← max(0, σ′ − paths(left(n′)))
23                     if paths(right(n′)) > σ″ ∧ δ″ > 0 then
24                         push(st, ⟨right(n′), P, σ″, δ″⟩)
25                 if paths(left(n′)) > σ′ then
26                     n′ ← left(n′)
27                 else
28                     break
```

Algorithm 2 receives as an input a tECS, a node n, a time-window ϵ, a position j, and a process p and traverses a fraction of $G_{\mathcal{E}}$ in a DFS-way left-to-right order. First, computes the parameters σ, δ corresponding to the starting and ending path to enumerate, respectively. The value of these parameters can be computed statically i.e. without message interchanging. Each iteration of the `while` of line 8 traverses a new path starting from the point it branches from the previous path (except for the first iteration). For this, the stack *st* is used to store the node and partial complex event of that branching point. Then, the `while` of line 9 traverses through the nodes of the next path, following the left direction whenever a union node is reached and adding the right node to the stack whenever need. The `if`s of line 23 and line 25 make sure that enumeration starts on path π_σ so only $paths_{\geq j-\epsilon,\sigma,\delta}$ are traversed. Moreover,

by checking for every node n' its value $max(n')$ before adding it to the stack, it makes sure of only going through paths in $paths_{\geq j-\epsilon}$.

A simpler recursive algorithm could have been used, however, the constant-delay output might not be guaranteed because the number of backtracking steps after branching might be as long as the longest path of $G_{\mathcal{E}}$. To guarantee constant steps after branching and assure constant-delay output, Algorithm 2 uses a stack which allows to jump immediately to the next branch. We assume that storing P in the stack takes constant time. We materialize this assumption by modelling P as a linked list of positions, where the list is ordered by the last element added. To update P with position i, we only need to create a node i that points to the previous last element of P. Then, storing P on the stack is just storing the pointer of the last element added.

This concludes the presentation of Algorithm 2. In the reminding of this section, we prove properties (1), (2) and (3).

We start by proving that Algorithm 2 enumerates $paths_{\geq\tau,\sigma,\delta}$ with output-linear delay after reaching the starting path π_σ, provided that $\mathcal{E}$ is k-bounded and time-ordered and n is a duplicate-free node.

Lemma 5.4. *Fix k, $\mathcal{P}$ and $p \in \mathcal{P}$. Let $\mathcal{E}$ be a k-bounded and time-ordered tECS, n a node of $\mathcal{E}$, and ϵ a time-window. Then, Algorithm 2 enumerates $paths_{\geq\tau,\sigma,\delta}$ with output-linear delay after reaching the starting path π_σ.*

Proof. Fix $\mathcal{E}$, τ, σ and δ. We first show that Algorithm 2 traverses all paths $paths_{\geq\tau,\sigma,\delta}(n)$. Notice, the order in which the paths are traversed is completely determined by the order of the union nodes: for each union node u, the paths to its left are traversed first, and then the ones to its right. Formally, for every node n' define the leftmost path from n' as $\pi_{\swarrow}(n') := n_0 \to n_1 \to \ldots \to n_l$ such that $n_0 = n'$ and, for every $i \leq l$:

- if $n_i \in N_B$, then $i = l$,
- if $n_i \in N_O$, then $n_{i+1} = next(n_i)$, and
- if $n_i \in N_U$, then $n_{i+1} = left(n_i)$.

For the first path, though, the order is different because the algorithm needs to skip all path before π_σ. Formally, for every node n' define the leftmost path

from n' after π_σ as $\pi_{\swarrow>\sigma}(n') := n_0 \to n_1 \to \ldots \to n_l$ such that $n_0 = n'$ and, for every $i \leq l$:

- if $n_i \in N_B$, then $i = l$,
- if $n_i \in N_O$, then $n_{i+1} = next(n_i)$, and
- if $n_i \in N_U$ and $paths(n_i) > \sigma$, then $n_{i+1} = left(n_i)$, otherwise, $n_{i+1} = right(n_i)$.

Consider a path $\pi := n_0 \to n_1 \to \ldots \to n_l$, and let $j \leq l$ be the last position such that n_j is a union node, $n_{j+1} = left(n_j)$, $max(right(n_j)) \geq \tau$, and $paths(n_j) > \sigma$. Then, let π_j^u be the path π up to position j i.e. that stops at such union node.

Let $P = \{\pi_1, \pi_2, \ldots, \pi_\delta\}$ be the set of paths enumerated by Algorithm 2 in that order. Then, by analysing Algorithm 2, one can see that $\pi_1 = \pi_{\swarrow>\sigma}(n)$ and, for every $i \leq \delta$, $\pi_i = \pi_{i-1}^u \cdot \pi_{\swarrow}(right(u))$. To put it another way, after reaching the starting path π_σ, it performs a greedy DFS from left to right: the first path to enumerate is $\pi_1 = \pi_{\swarrow>\sigma}(n)$, then each π_i is the path in $paths_{\geq\tau,\sigma,\delta}(n)$ that branches from π_{i-1} to the right at the deepest level u and from there follows the leftmost path. Moreover, to jump from π_{i-1} to π_i, the node popped by the stack is exactly u, that is, the last node of π_{i-1}^u.

To show that the enumeration is done with output-linear delay after enumerating the first path, we study how long it takes between enumerating the complex events of π_{i-1} and π_i. Consider that π_{i-1} was just traversed and its complex event was output by line 11. Then, the `break` of line 12 is executed, breaking the `while` of line 9. Afterwards, either the stack is empty and the algorithm ends, or a pair (n', P) is popped from the stack, where n' corresponds to the last node of π_{i-1}^u. From that point, it is straightforward to see that the number of iterations of the while of line 9 (each taking constant time) is equal to the number of nodes l in $\pi_{\swarrow}(n')$, so those nodes are traversed and the complex event of the path π_i is output. But, because $\mathcal{E}$ is k-bounded, then $l \leq k \cdot |C|$, where C is the complex event of π_i. Finally, the time taken is bounded by the size of the output, and the enumeration is performed with output-linear delay after reaching the first path. □

Then, we present Lemma 5.5 that follows from Theorem 5.3 which will be necessary in the following proof of Theorem 5.6.

A note on notation conventions: we denote *injective functions* as $x \mapsto f(x)$.

Lemma 5.5 ($paths_{\geq j-\epsilon,\sigma,\delta}(n) \mapsto [\![\mathrm{n}]\!]^{\epsilon}_{\mathcal{E}}(j)$)**.** *For every path in $paths_{\geq j-\epsilon,\sigma,\delta}(n)$ there exist exactly one complex event in $[\![n]\!]^{\epsilon}_{\mathcal{E}}(j)$ within a time window of size ϵ that starts at event $pos(b)$ and ends at event j.*

Proof. Fix j, ϵ, σ, and δ. Let n be a node in $\mathcal{E}$. Recall that $paths_{\geq j-\epsilon,\sigma,\delta}(n) \subseteq paths_{\geq j-\epsilon}(n)$. Let $S := paths_{\geq j-\epsilon,\sigma,\delta}(n)$ be the subset of $paths_{\geq j-\epsilon}(n)$ corresponding to $paths_{\geq j-\epsilon,\sigma,\delta}(n)$. Then, by Theorem 5.3, $S \subseteq [\![\mathrm{n}]\!]^{\epsilon}_{\mathcal{E}}(j)$. And so, $paths_{\geq j-\epsilon,\sigma,\delta}(n) \mapsto [\![\mathrm{n}]\!]^{\epsilon}_{\mathcal{E}}(j)$. □

Now, we can finally prove that Algorithm 2 enumerates a subset of $[\![\mathrm{n}]\!]^{\epsilon}_{\mathcal{E}}(j)$ of size $\mathcal{O}(\frac{[\![\mathrm{n}]\!]^{\epsilon}_{\mathcal{E}}(j))}{|\mathcal{P}|})$ with output-linear delay after the first complex event, provided that $\mathcal{E}$ is k-bounded and time-ordered and n is a duplicate-free node.

Theorem 5.6. *Fix k, $\mathcal{P}$ and $p \in \mathcal{P}$. Let $\mathcal{E}$ be a k-bounded and time-ordered tECS, n a node of $\mathcal{E}$, ϵ a time-window. Then, Algorithm 2 enumerates a subset of $[\![n]\!]^{\epsilon}_{\mathcal{E}}(j)$ of size $\mathcal{O}(\frac{[\![n]\!]^{\epsilon}_{\mathcal{E}}(j))}{|\mathcal{P}|})$ with output-linear delay after enumerating the first complex event in $[\![n]\!]^{\epsilon}_{\mathcal{E}}(j)$.*

Proof. Fix τ. Let $\delta = \frac{|paths_{\geq \tau}(n)|}{|\mathcal{P}|}$ and $\sigma = index(p) \cdot \delta$ be constants as in Algorithm 2. By Lemma 5.4, Algorithm 2 enumerates $paths_{\geq \tau,\sigma,\delta}$ with output-linear delay after π_σ. We need to prove that: (1) $paths_{\geq \tau,\sigma,\delta}(n)$ corresponds to a subset S of $[\![\mathrm{n}]\!]^{\epsilon}_{\mathcal{E}}(j)$, (2) $|S| = \mathcal{O}(\frac{[\![\mathrm{n}]\!]^{\epsilon}_{\mathcal{E}}(j))}{|\mathcal{P}|})$, and (3) it enumerates with output-linear delay after enumerating the first complex event in $[\![\mathrm{n}]\!]^{\epsilon}_{\mathcal{E}}(j)$.

- (1) By Lemma 5.5, immediately follows that $paths_{\geq \tau,\sigma,\delta}(n)$ corresponds to a subset $S \subseteq [\![\mathrm{n}]\!]^{\epsilon}_{\mathcal{E}}(j)$.
- (2) By Lemma 5.4, Algorithm 2 enumerates the set $paths_{\geq \tau,\sigma,\delta}(n)$ of size at most δ, then S is of size at most $\frac{|paths_{\geq \tau}(n)|}{|\mathcal{P}|}$. And, by Theorem 5.3, S is at most of size $\frac{|[\![\mathrm{n}]\!]^{\epsilon}_{\mathcal{E}}(j))|}{|\mathcal{P}|}$.
- (3) By Lemma 5.4, Algorithm 2 enumerates with output-linear delay after $\pi_\sigma \in paths_{\geq \tau,\sigma,\delta}(n)$. And, by Lemma 5.5, for every path $\pi \in paths_{\geq \tau,\sigma,\delta}(n)$ there exists one complex event in $[\![\mathrm{n}]\!]^{\epsilon}_{\mathcal{E}}(j)$.

□

To conclude, we start with Lemma 5.7 and finally prove Theorem 5.1 from Section 5.1 corresponding to property (3).

Lemma 5.7. *Let $\mathcal{E}$ be a time-ordered tECS, n a duplicate-free node of $\mathcal{E}$, ϵ a time-window, $\mathcal{P}$ the set of all processes. Let $P_p := paths_{\geq\tau,\sigma,\delta}(n)$ be the output of Algorithm 2 on process $p \in \mathcal{P}$. Then, $\bigcup_{p\in\mathcal{P}} P_p = paths_{\geq\tau}(n)$.*

Proof. Fix τ and $\mathcal{P}$.

Let n be a node in $\mathcal{E}$. Recall that $paths_{\geq\tau}(n) = \{\pi_0, \pi_1, \ldots, \pi_m\}$ where $m = |paths_{\geq\tau}(n)|$ and $paths_{\geq\tau,\sigma,\delta}(n) = \{\pi_\sigma, \pi_{\sigma+1}, \ldots, \pi_{\sigma+\delta}\}$ is a subset of $paths_{\geq\tau}(n)$.

For every process $p \in \mathcal{P}$, Lemma 5.4 states that Algorithm 2 enumerates $paths_{\geq\tau,\sigma,\delta}$ where σ and δ are variables that depend on p and P, respectively.

Let $\delta = \lceil \frac{m}{|\mathcal{P}|} \rceil$ and $\sigma = index(p) \cdot \delta$ be constants as in Algorithm 2. Let $P_0 = paths_{\geq\tau,0,\delta}(n)$ be the output of process 0, $P_1 = paths_{\geq\tau,\delta,2\cdot\delta}(n)$ be the output of process 1, ..., $P_{|\mathcal{P}|-1} = paths_{\geq\tau,(|\mathcal{P}|-1)\cdot\delta,|\mathcal{P}|\cdot\delta}(n)$ be the output of process $|\mathcal{P}|-1$, i.e. process 0 enumerates $\{\pi_0, \ldots \pi_{\delta-1}\}$, process 1 enumerates $\{\pi_\delta, \ldots \pi_{2\cdot\delta-1}\}$, ..., process $(|\mathcal{P}| - 1)$ enumerates $\{\pi_{(|\mathcal{P}|-1)\cdot\delta}, \ldots, \pi_{\mathcal{P}\lceil\frac{m}{\mathcal{P}}\rceil}\}$. Then, the union of all the outputs corresponds to the set of paths $\{\pi_0, \pi_1, \ldots, \pi_{(|\mathcal{P}|-1)\frac{m}{\mathcal{P}}}, \pi_m\}$ that corresponds to the definition of $paths_{\geq\tau}(n)$. And so, $\bigcup_{p\in\mathcal{P}} P_p = paths_{\geq\tau}(n)$. □

Theorem (5.1)**.** *Let $\mathcal{E}$ be a time-ordered tECS, n a duplicate-free node of $\mathcal{E}$, ϵ a time-window, $\mathcal{P}$ the set of all processes. Let C_p be the output of Algorithm 2 on process $p \in \mathcal{P}$. Then, $\bigcup_{p\in\mathcal{P}} C_p = [\![n]\!]^\epsilon_\mathcal{E}(j)$.*

Proof. Fix τ and $\mathcal{P}$. For every node n in $\mathcal{E}$, Lemma 5.7 states that the union of the output of each process corresponds to $paths_{\geq\tau}(n)$. Then, we apply Theorem 5.3, which states that there is an isomorphism between $paths_{\geq\tau}(n)$ and $[\![\mathrm{n}]\!]^\epsilon_\mathcal{E}(j)$, to the output of Lemma 5.7. And so, the union of the output of each process in $\mathcal{P}$ corresponds to $[\![\mathrm{n}]\!]^\epsilon_\mathcal{E}(j)$ i.e. $\bigcup_{p\in\mathcal{P}} C_p = [\![\mathrm{n}]\!]^\epsilon_\mathcal{E}(j)$. □

This concludes the proofs of properties (1), (2), and (3).

5.5 Chapter summary

In this chapter we have presented an efficient distributed algorithm for evaluating CEA $\mathcal{A}$. First, we introduced the compact data structure tECS and the union-lists, its restrictions and operations. Secondly, we explained thoroughly the evaluation algorithm and analysed its complexity. Finally, we described and discussed the ENUMERATION procedure in detail.

Chapter 6

Experimental evaluation

In this chapter, we propose a set of experiments to study the performance and scalability of both our distributed framework for CER and our distributed evaluation algorithm. The designed experiments focus on the evaluation of complex predicates. This chapter is organized as follows. First, we describe our implementation. Then, we describe the design and set up of our experiments, including how we generated the synthetic data and the characteristics of the system. Then, for each experiment, we describe the experiment and discuss the results. Finally, we summarise the main conclusions of the chapter.

6.1 DCORE in a nutshell

In this section, we review some implementation details of DCORE. We implemented DCORE to run on the JVM [44]. Its code is open-source and available at `https://github.com/dtim-upc/DCORE` under the GNU GPLv3 license. DCORE implementation depends on a fork of CORE. This fork implements our novel distributed evaluation algorithm.

CORE. CORE is implemented in Java 11 and includes the following built-in optimizations. Complex predicates are encoded in an efficient array representation, which are only evaluated once during the execution of the algorithm. Regarding I/O-determinization, this is run *on the fly* and many of the steps are cached and only computed once. CORE also employs advanced memory management: nodes in the tECS data structure are weakly referenced, while

the strong references are stored in a list that is pruned once in a while taking into account nodes that are outside of the time window, allowing the garbage collector to reclaim that memory without the need to modify the tECS data structure.

DCORE. DCORE is implemented in Scala 2.12 [45]. Scala [45] is a statically typed programming language that fuses object-oriented and functional programming, which can freely interoperate with Java. DCORE is built on top of Akka [46] (Akka Cluster in particular). Akka is a set of open-source libraries for designing scalable, resilient system that span processor cores and networks. Akka's use of the actor model provides a level of abstraction that makes it easier to write correct concurrent, parallel and distributed systems. DCORE depend on several libraries that are downloaded and compiled using Sbt [47]. Sbt is a typesafe and parallel build tool for Scala and Java projects. Sbt guarantees reproducibility of the compilation of the project. As previoulsy mentioned, DCORE depends on our own fork of CORE that implements the novel distributed evaluation algorithm. DCORE is built as a white box command-line interface program and has many available parameters. The landing page of the project has a detailed explanation on how to compile and run the project. Respect to the correctness of the implementation, we designed several automated tests to validate that the implementation is close to the specification.

6.2 Experimental setup

We compare our framework's implementations, DCERE and DCORE, against the leading CER system: CORE [8]. The comparison with alternative competitors such as SASE [10], Esper [9], FlinkCEP [32] or TESLA/T-REX [11] is left out of this work considering that CORE's throughput is three orders of magnitudes higher than them. CORE, DCERE, and CORE are semantically comparable since they all use the same query language. Unfortunately, CORE cannot evaluate non-unary predicates, thus we extended the system with an ad-hoc rewrite and refine algorithms, similar to how DCERE is implemented, but with no distributed part.

Setting. We run our experiments on a server equipped with a 6-core (2 threads per core) i7-8700 processor running at stock frequency, 32GB of RAM,

Arch Linux operating system, 5.15.7-arch1-1 kernel version, OpenJDK Runtime 11.0.12, and the OpenJDK 64-Bit Server Virtual Machine build 11.0.12. The Virtual Machine is restarted with 1024MB of freshly allocated memory before each experiment.

Note. All the experiments are run on a single server instead of a cluster of servers. We speculate that the results of our experiments in a cluster would be similar, with small differences due to the overhead of network communication. Validating this hypothesis is left for future work.

We compare systems with respect to their performance. All reported numbers are averages taken over three repetitions of each experiment. We measure the performance of each system, expressed as the execution time of the experiment in milliseconds, as follows. To avoid measuring the data loading time of each system, we first load the input stream completely in main memory. We then start the timer, run the experiment, and stop the timer as soon as the last output is enumerated. Recognized complex events are written to /dev/null device to guarantee that the events are enumerated (avoiding *dead call optimization*), but at the same time not measuring writing costs. The *null device* is a virtual device present in all Linux systems that results in zero cost on writing operations. For consistency reasons, we have verified that all systems produced the same set of complex events.

ACM SIGMOD 2022 Availability & Reproducibility. Our work complies with ACM SIGMOD 2022 Availability & Reproducibility criteria [48] : (1) our prototype is provided as a white box, (2) the process to generate the input data is available, (3) the experiments can be reproduced in order to generate the experimental data, and (4) the script/spreadsheet to transform the raw data into the plots is provided.

As far as we are concerned, there does not exist a standard benchmark for complex event recognition. For this reason, we experiment with a set of queries over synthetic data. We considered sequence queries and iteration queries for our experiments, which have been used for benchmarking in CER before (for example [7] and [49]). Specifically, we have considered the three queries depicted in Figure 6.1 where the pattern P is different on each query. We remark that the considered experimental queries contain a *binary predicate* (i.e., `A[id] = B[id]`). Indeed, it would be interesting to evaluate other types

of complex predicates, but this would require an actual implementation of the rewrite and refine algorithm, which is outside of the scope of this work.

```
SELECT *
FROM S
WHERE P
FILTER A[id] = B[id]
WITHIN 100 events
```

Q_1 :	$P = A; B; C$
Q_2 :	$P = A; B+; C$
Q_3 :	$P = A+; B+; C$

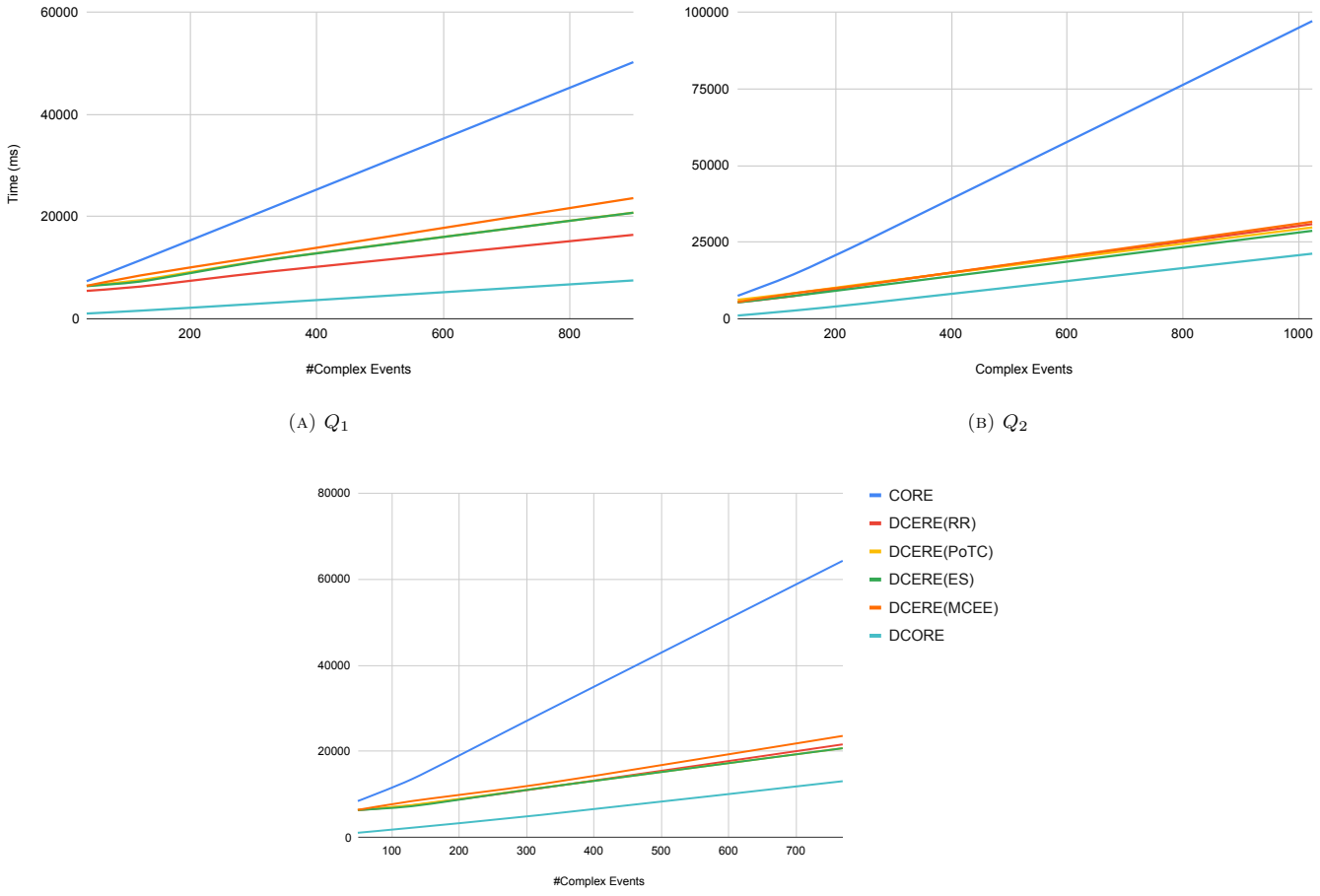
FIGURE 6.1: Queries considered in the experimental setting

We generated three different input streams, one per query, that guarantee that the number of complex events matched is the same in each. As depicted in Figure 6.2, in S_1, we generate as many events of type A as events of type B, and only a single closing event C, which forces the partial open events to close. In S_2, we generated a single event of type A, followed by many events of type B, and a closing event C. In S_3, we generated a stream of pairs A, B and a single closing event C. The *id* attribute has been uniformly at random assigned with values $\{1, 2, 3\}$. The length of each stream is different for a particular size of set of complex events (e.g., to generate 100 complex events, S_1 requires a stream containing 10 events of type A and 10 events of type B, while S_2 only requires a stream containing 7 events of type B).

S_1 :	A,A...B,B...C
S_2 :	A,B,B,B...C
S_3 :	A,B,A,B...C

FIGURE 6.2: Example input streams

The following sections are devoted to the experimental results. For each experiment, we will state the aspect under evaluation, describe the experiment, and discuss the results.



(A) Q_1

(B) Q_2

(C) Q_3

FIGURE 6.3: The performance of evaluating queries Q_1, Q_2, and Q_3 over stream S_1, S_2, S_3, respectively.

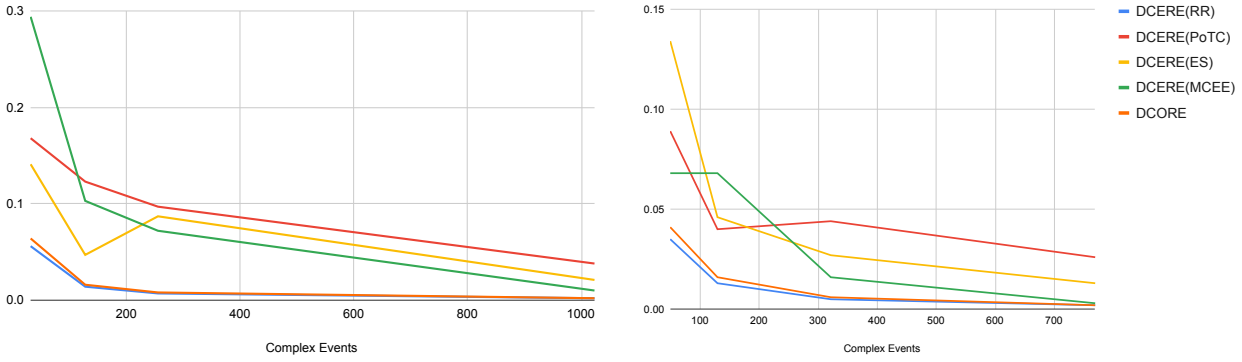
6.3 Experiments on the evaluation of complex predicates

In this section, we designed an experiment to measure the performance characteristics of CORE, DCERE, and DCORE under queries with complex predicates. This experiment also evaluates the performance of DCERE under different distributions strategies. In Figure 6.3, we display the results of evaluating queries Q_2, and Q_3 for an increasing number of complex events. The recognition time of all system increases as the number of complex event grows. Nevertheless, the degrading behaviour of CORE is more evident in this plot. Surprisingly, all distribution strategies have similar performance. The results of this experiment validate our hypothesis that distribution strategy Maximal

Complex Event Enumeration does not perform well in practise, as discussed in Section 4.2.1. We conclude that DCORE performs better than both CORE and DCERE. Furthermore, in the presence of complex and large streams, our novel distributed evaluation algorithm outperforms its sequential predecessor.

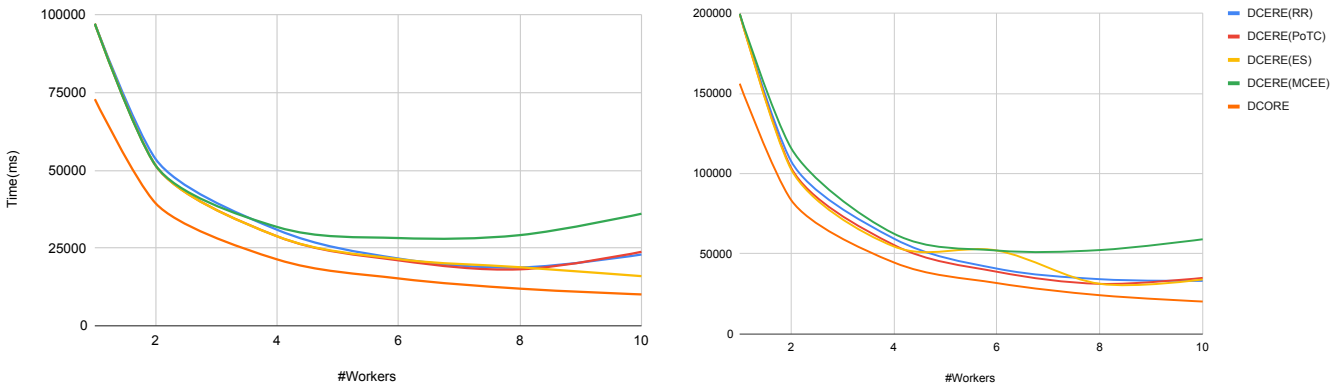
6.4 Experiments on the scalability of the framework

In this section, we study the scalability of DCERE and DCORE. For this reason, we designed two different experiments. The first experiment studies the quality on the load-balance of the different distribution strategies. For this experiment, we compute the *coefficient of variation* of the load of each processing unit during the evaluation of the queries Q_2, and Q_3. The coefficient of variation (CV) is a standarized measure of *dispersion* of a frequency distribution and it is defined as the ratio of the *standard deviation* to the *mean*. The lower the CV the less variability among the loads of the processing units; hence, a better scalability of the system on uniform processing units. In Figure 6.4, we show the coefficient of variation of the different distribution strategies (i.e. RR, PoTC, ES, and MCEE) and DCORE. The results show that all systems have a low coefficient of variation (< 0.3), with almost optimal CV on DCORE.



(A) Q_2 (B) Q_3

FIGURE 6.4: The coefficient of variation of evaluating queries Q_2, and Q_3 under distributions strategies RR, PoTC, ES, and MCEE.



(A) 1024 complex events

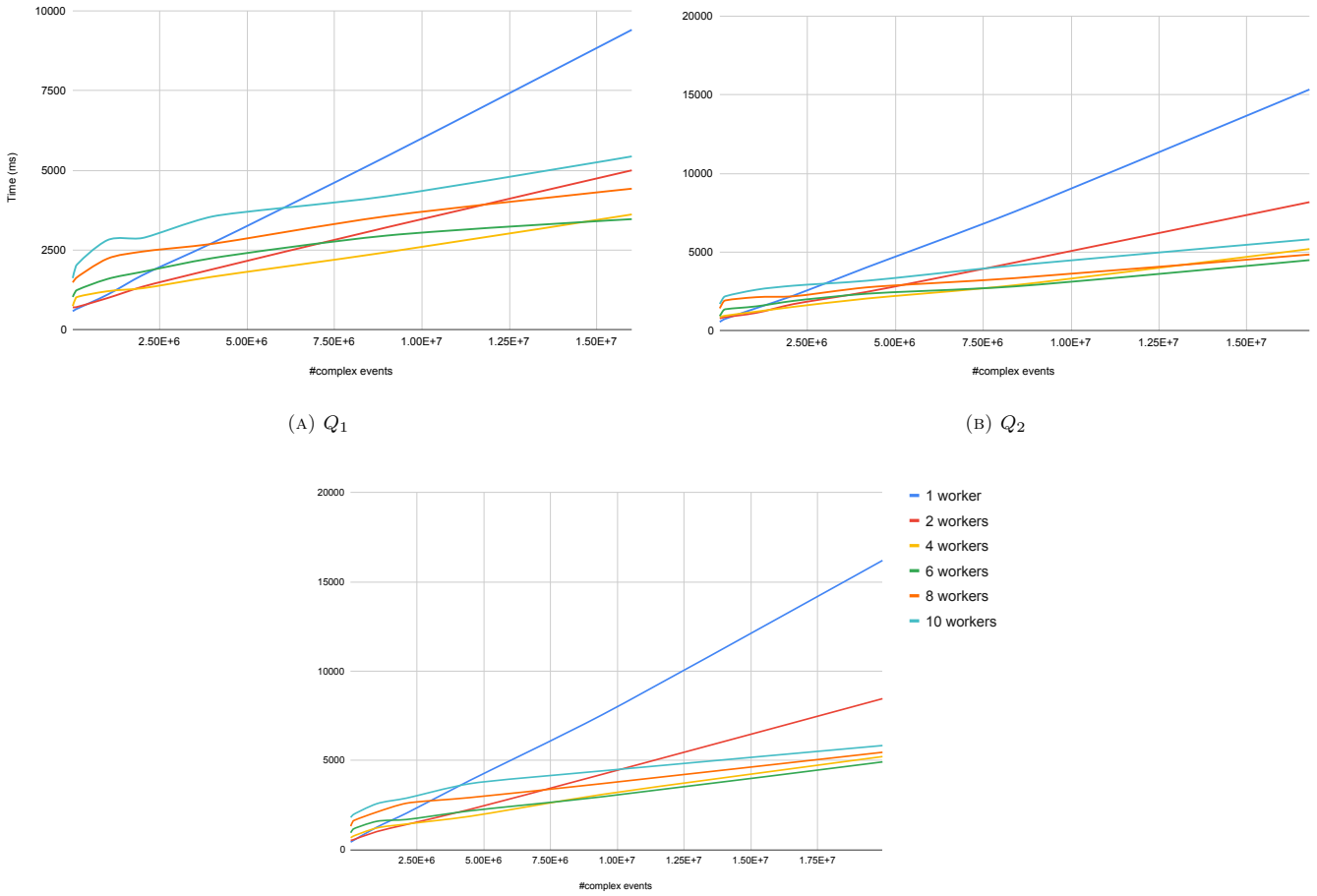
(B) 2048 complex events

FIGURE 6.5: The horizontal scalability of DCERE and DCORE evaluating query Q_2 over increasing number of processing units.

The second experiment studies the horizontal scalability of the systems. In Figure 6.5, we show the execution time of the processing of query Q_2 on an increasing number of processing units, from 1 to 10. This experiment is run on two different streams. At the left, we executed the experiment with an stream that contained 1024 complex events. At the right, the number of complex events is doubled, 2048. This will allow us to study the impact of an increase of complexity in the scalability of the system. The results show that DCORE scales *almost linearly* in the number of processing units. However, DCERE reaches a threshold of *negative results* around 8 processing units, where increasing the number of processing units only degrades the performance of the system. In other words, DCORE scaling is better compared to DCERE.

6.5 Experiments on DCORE's evaluation algorithm under Big Data requirements

In this last section, we study the performance of the distributed evaluation algorithm of DCORE under heavy loads. For this reason, we designed three experiments. In the first experiment, we run DCORE over queries Q_1, Q_2, and Q_3, with variable number of processing units $1, 2, 4, 6, 8, 10$. For each configuration, we run the experiment on an increasing number of complex events. Notice, the number of complex events in this experiment is *orders of magnitudes (OOM)* larger than in previous ones. In Figure 6.6, we show the



(A) Q_1

(B) Q_2

(C) Q_3

FIGURE 6.6: The horizontal scalability of DCORE under increasing number of processing units.

execution time of the processing of queries Q_1, Q_2, and Q_3 under different number of processing units. The results show that under small loads, the sequential version of the algorithm (corresponding to DCORE with a single processing unit) performs better than the distributed version. Under heavy loads (i.e., more than 1 million complex events) the distribution pays off. The optimal number of processing units depend on the scale of the problem. For example, under a load of 10^6 complex events the best configuration is 4 workers. We see that depending on the query, the performance of the system changes. We conclude that under small loads distributing the system only degrades performance, while under heavy loads the distribution speeds up the process of recognition of complex events.

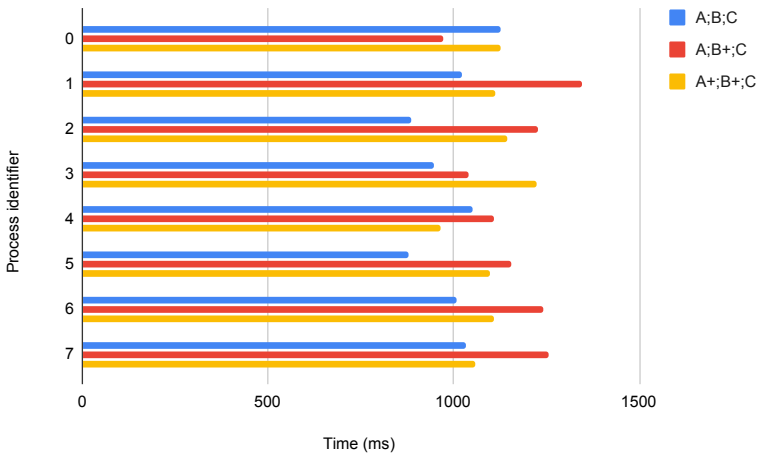
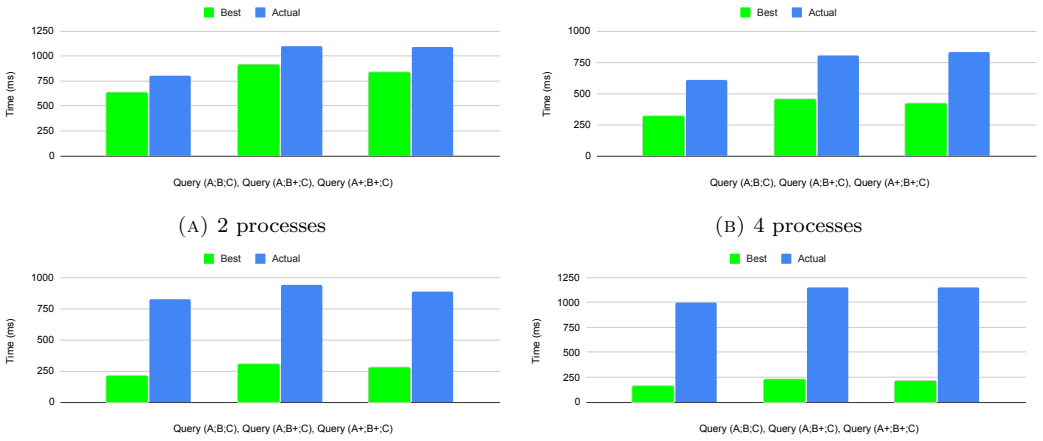


FIGURE 6.7: The execution time on each processing unit of DCORE evaluating queries Q_1, Q_2, and Q_3.

In the second experiment, we study the execution time of each processing unit during the evaluation of queries Q_1, Q_2, and Q_3. The goal of this experiment is to analyse the amount of work of each processing unit. A balanced distribution of work loads would result in overall better performance of the system. In Figure 6.7, we show the results of the second experiment on 8 processing units over a fixed length of stream. The results show no evidence of significant differences between the execution time on each processing unit. We conclude that this is a relevant factor on the performance of DCORE.

In this last experiment, we compare the performance of DCORE against an hypothetical optimal distributed system based on CORE. The goal of this experiment is to analyse how close is DCORE from an optimal distributed system with same performance characteristics on the evaluation algorithm (e.g., CORE). For this experiment, we evaluated queries Q_1, Q_2, and Q_3 over a stream of fixed length that produced $2 \cdot 10^6$ complex events. The experiment is repeated for different number of processes: $2, 4, 6, 8$. In Figure 6.8, we show the results of this experiment. The blue bars corresponds to the execution time of the experiment for DCORE, while the green bars corresponds to the execution time of CORE divided by the number of processing units of each iteration, which hypothetically corresponds to the best performance DCORE could ever achieve.



(A) 2 processes

(B) 4 processes

(C) 6 processes

(D) 8 processes

FIGURE 6.8: Comparison of the execution time of an optimal distributed system based on CORE against DCORE.

The results show that DCORE is close to the optimal value when distributed over 2 processing units. However, the gap grows as the number of processing units increases. We conjecture that the cost of enumerating the first complex event in the data structure diminish the performance gains from distributions. In other words, in order to compensate the cost of enumerating the first path on the tECS, which is linear in the size of the complex event, each processing unit requires a reasonable amount of complex events.

6.6 Chapter summary

In this chapter, we presented a set of experiments to study the performance and scalability of both our distributed framework for CER and our distributed evaluation algorithm. First, we describe the implementation of DCORE. Then, we described the general settings of the experiments such as hardware, input streams and queries, among others. Finally, we presented and discussed the experiments, and their corresponding results.

Chapter 7

Conclusions and future work

We presented a novel distributed CER framework that focuses on the efficient evaluation of a large class of complex event queries, including n-ary predicates. We proposed two implementations based on such framework: DCERE and DCORE. In particular, DCORE uses a novel evaluation algorithm that tackles the super-linear complexity of non-unary predicates and the exponential complexity of the enumeration. Furthermore, our experiments results show that our framework is practical and outperforms its competitors on queries with complex predicates over large streams of data.

We plan to extend our research in a few directions. We will extend our framework with a generic rewrite and refine algorithms. We are also working on an extension of the distributed evaluation algorithm that takes into account time windows during the distribution phase. Finally, we will explorer the correlation capabilities offered register and data automata, and the aggregation capabilities of cost register automata.

Appendix A

A.1 Proof of Theorem 5.3

Theorem A.1 ($[\![\mathrm{n}]\!]^{\epsilon}_{\mathcal{E}}(j) \longleftrightarrow paths_{\geq j-\epsilon}(\mathrm{n})$)**.** *For every complex event within a time window of size ϵ there exists exactly one path that reaches a bottom node b with $pos(b) \geq j - \epsilon$, and vice versa.*

Proof. Fix j, ϵ, and $\mathcal{E}$. Let n be a node in $\mathcal{E}$. The proof follows by the definition of $[\![\mathrm{n}]\!]^{\epsilon}_{\mathcal{E}}(j)$, $[\![\mathrm{n}]\!]_{\mathcal{E}}$, $[\![\bar{p}]\!]_{\mathcal{E}}$, and $paths_{\geq\tau}(\mathrm{n})$. Recall that

- $[\![\mathrm{n}]\!]^{\epsilon}_{\mathcal{E}}(j) = \{([i,j], D) \mid (i, D) \in [\![\mathrm{n}]\!]_{\mathcal{E}} \wedge j - i \leq \epsilon\}$ encodes all open complex events represented by n in $\mathcal{E}$ that, when closed with j, are within a time window of size ϵ,
- $[\![\mathrm{n}]\!]_{\mathcal{E}} = \bigcup_{\bar{p}; start(\bar{p})=n} [\![\bar{p}]\!]_{\mathcal{E}}$ encodes all open complex events $[\![\bar{p}]\!]_{\mathcal{E}}$ with $\bar{p}$ a full-path in $\mathcal{E}$ starting at n, and
- $[\![\bar{p}]\!]_{\mathcal{E}} = (i, D)$ where $\bar{p} = n_1, n_2, \ldots, n_k$ be a *full-path* in $\mathcal{E}$ such that n_k is a bottom node, $i = pos(n_k)$ is the label of the bottom node n_k, and D is the set of labels of the other non-union nodes in $\bar{p}$.

First, we prove $[\![\mathrm{n}]\!]^{\epsilon}_{\mathcal{E}}(j) \longmapsto paths_{\geq j-\epsilon}(\mathrm{n})$. Given a complex event $([i,j], D) \in [\![\mathrm{n}]\!]^{\epsilon}_{\mathcal{E}}(j)$, there is an open complex event $(i, D) \in [\![\mathrm{n}]\!]_{\mathcal{E}}$ that is represented as the full-path $\bar{p} = n_1, n_2, \ldots, n_k$ in $\mathcal{E}$ such that n_k is a bottom node and $i = pos(n_k)$ is the label of the bottom node n_k. Notice, $n_1 = \mathrm{n}$ is the starting node, $j = pos(n_1)$ is the label of the starting node n_1, and $j - i \leq \epsilon$. By definition, $\bar{p} \in paths_{\geq\tau}(\mathrm{n})$.

Secondly, we prove that $paths_{\geq j-\epsilon}(\mathrm{n}) \longmapsto [\![\mathrm{n}]\!]^{\epsilon}_{\mathcal{E}}(j)$. The proof follows by expanding the definition of $paths_{\geq \tau}(\mathrm{n})$ and following the steps of the previous proof in reverse order.

Finally, by $[\![\mathrm{n}]\!]^{\epsilon}_{\mathcal{E}}(j) \longmapsto paths_{\geq j-\epsilon}(\mathrm{n})$ and $paths_{\geq j-\epsilon}(\mathrm{n}) \longmapsto [\![\mathrm{n}]\!]^{\epsilon}_{\mathcal{E}}(j)$, $[\![\mathrm{n}]\!]^{\epsilon}_{\mathcal{E}}(j) \longleftrightarrow paths_{\geq j-\epsilon}(\mathrm{n})$ immediately holds.

□

A.2 Algorithms Chapter 4

A.2.1 Maximal Complex Event Enumeration

Algorithm 3: Distributed enumeration of a set of maximal complex events $\mathcal{C}_V^{\max}$ over a set of processing units $\mathcal{P}$.

```
1  procedure Mcee(C_V^max, P)
2      K ← ∅
3      for C_V^max ∈ C_V^max do
4          K ← K ∪ Configurations(C_V^max).Map(λK → ⟨K, C_V^max⟩)
5      D ←GroupBy(K, λ⟨K, _⟩ → K)
6      Distribute(P, D)
7
8  procedure Enumerate(D)
9      for ⟨K, C_V^max⟩ ∈ D do
10         T ← new-node()
11         for C_V^max ∈ C_V^max do
12             G ← Partition(C_V^max)
13             Loop(T, G, ∅, ⊥, K)
14
15 procedure Loop(n, G, C_V, new, K)
16     switch G do
17         case ∅ do
18             if new then
19                 return C_V
20         case G ∪ G do
21             k ← K(type(G))
22             N ← (G choose k)
23             for i ∈ N do
24                 if ∃n' ∈ children(n). event(n') = i then
25                     Loop(n', G, C_V ∪ i, new)
26                 else
27                     n' ←new-node(i)
28                     add(children(n), n')
29                     Loop(n', G, C_V ∪ i, ⊤, K)
```

Algorithm 3 consist of two procedures: MCEE and ENUMERATE. The main procedure is MCEE, which is executed in the master actor, while ENUMERATE is executed on each slave actor. It receives as an input a set of *maximal* complex events $\mathcal{C}_V^{\max} := \{C_V^1, \ldots, C_V^n\}$ and a set of processing unit $\mathcal{P} := \{P_1, \ldots, P_n\}$, and outputs all *complex events* $C'_V \subseteq \mathcal{C}_V^{\max}$ distributedly. The `for` (lines 3-4) compute the set of *configurations* corresponding to each maximal complex event in $\mathcal{C}_V^{\max}$ (see Algorithm 4) and pairs each configuration with its maximal complex event. A *configuration* is a binary relation $\mathbf{T} \times \mathbb{N}$ from data-tuples $t \in \mathbf{T}$ to natural numbers. For example, given complex event $C_V := \{T, T, H, H, H\}$, then $\texttt{Configurations}(C_V) = \{\{T, 2\}, \{H, 3\}\}$.

Algorithm 4: Configuration of a complex event C_V.

```
1  procedure CONFIGURATIONS(C_V)
       Input: A complex event C_V = {i, ..., j} with C_V ⊆ 2^ℕ.
       Output: A set 𝒦 of configurations K := T × ℕ where K is the
               mapping from the event type t ∈ T to the size of the group
               of consecutive events of type t in the complex C_V.
2      𝒱 ← ∅
3      i ∪ C'_V ← head(C_V)
4      A ← {i}
5      type(A) ← type(i)
6      for j ∈ C'_V do
7          if type(j) = type(A) then
8              A ← A ∪ j
9              if last(C_V) = j then
10                 𝒱 ← 𝒱 ∪ enum(1, |A|)
11         else
12             𝒱 ← 𝒱 ∪ enum(1, |A|)
13             A ← {j}
14             type(A) ← type(j)
15     𝒲 ← ⨉_{V∈𝒱} V
16     T ← ordered-types(C_V)
17     𝒦 ← ∅
18     foreach W ∈ 𝒲 do
19         K ← ∅
20         for i ← 1 to |W| do
21             K ← K ∪ (T[i], W[i])
22         𝒞 ← 𝒞 ∪ C
23     return 𝒞
```

GROUPBY of line 5 groups the set of tuples $\mathcal{K} := \{\langle C_V, K\rangle, \ldots\}$ by their configuration resulting in the set of tuples $D := \{\langle K, \mathcal{C}_V\}\rangle, \ldots\}$, where $\mathcal{C}_V := \{C_V^1, \ldots, C_V^m\}$. Then, the set D is distributed among the $|\mathcal{P}|$ processing units

using a generic load-balancing algorithm DISTRIBUTE. The choice of implementation does not affect the correctness of the algorithm.

The procedure ENUMERATE receives the set of tuples $D := \{\langle K, \mathcal{C}_V\}\rangle, \ldots\}$ and enumerates all complex events included in each $\mathcal{C}_V$ filtered by K. K is a configuration and for each event type T, it returns the exact number of complex event of that type that must be present in the resulting complex events. This is what allows us to control the load-balancing of the process, by distributing the configurations assigned to each process. For each tuple $\langle K, \mathcal{C}_V\rangle$, lines 10-13 are executed. First, a new *tree* root T is created on line 10. This n-ary tree will be used through the algorithm to detect which complex events have been outputted before to avoid duplicates. Then, for each maximal complex event $C_V \in \mathcal{C}_V$, the procedure PARTITION is executed and the result is given to procedure LOOP.

The procedure PARTITION partitions the complex event C_V in sets of consecutive positions of events of the same type. For example,

$$\text{PARTITION}(\{T, T, H, H, H\}) = \{\{T, T\}, \{H, H, H\}\}$$

LOOP receives as input a node n, a grouped complex event $\mathcal{G}$, a partial complex event C_V, a boolean *new*, and a configuration K. On each iteration, LOOP extends C_V with the next group in $\mathcal{G}$ and the configuration K associated to that group. The `switch` from line 16 is split in two cases. Case 1 (lines 17-19) corresponds to the base case when $\mathcal{G}$ is empty. If the complex event C_V has not been outputted before (i.e., $new = \top$), then we output the complex event C_V and stop. Case 2 (lines 20-29) corresponds to the inductive step when $\mathcal{G}$ has at least a group of events of the same type in the corresponding complex event. Notice, that $\mathcal{G}$ on each iteration is smaller, ergo the algorithm terminates. For each group of events of the same type G, $k \in \mathbb{N}$ is retrieved from the configuration K, which corresponds to the size assigned to that processing unit for the group G. Different processing units will have different sizes assigned to each group, resulting in disjoint complex events enumerated by each unit. The k-combination set $N := \binom{G}{k}$ is computed, where N contains permutations of complex event positions. Then, for each k-combinations of events, lines 23-29 are executed. In both cases, C_V is extended with positions $i \subseteq 2^{\mathbb{N}}$. If there exists a children node n' in n that contains events i, then we recursively call LOOP with extended complex even $C_V \cup i$, but we do not update *new* since an event such as $C_V \cup i$ has already been outputted before. Otherwise, we create a new node n' with events i, extend our current node n

with n', and, as before, we call LOOP, but this time with argument $new = \top$ indicating that complex event C_V has not been enumerated before, which will eventually reach the base case and enumerate this new complex event.

Bibliography

[1] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink™: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38(4):28–38, 2015. URL `http://sites.computer.org/debull/A15dec/p28.pdf`.

[2] Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, and Matei Zaharia. Structured streaming: A declarative API for real-time applications in apache spark. In Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 601–613. ACM, 2018. doi: 10.1145/3183713.3190664. URL `https://doi.org/10.1145/3183713.3190664`.

[3] Manolis Pitsikalis, Alexander Artikis, Richard Dreo, Cyril Ray, Elena Camossi, and Anne-Laure Jousselme. Composite event recognition for maritime monitoring, 2019.

[4] B. Mukherjee, L.T. Heberlein, and K.N. Levitt. Network intrusion detection. *IEEE Network*, 8(3):26–41, 1994. doi: 10.1109/65.283931.

[5] Tibor Vámos. Automation production systems and computer integrated manufacturing: Mikell p. groover. *Autom.*, 24(4):587, 1988. doi: 10.1016/0005-1098(88)90106-9. URL `https://doi.org/10.1016/0005-1098(88)90106-9`.

[6] B. S. Sahay and Jayanthi Ranjan. Real time business intelligence in supply chain analytics. *Inf. Manag. Comput. Secur.*, 16(1):28–48, 2008. doi: 10.1108/09685220810862733. URL `https://doi.org/10.1108/09685220810862733`.

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

[8] Marco Bucchi, Alejandro Grez, Andrés Quintana, Cristian Riveros, and Stijn Vansummeren. CORE: a complex event recognition engine. *CoRR*, abs/2111.04635, 2021. URL `https://arxiv.org/abs/2111.04635`.

[9] Complex event processing, streaming analytics, streaming sql, Jan 2021. URL `https://www.espertech.com/`.

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

[11] Gianpaolo Cugola and Alessandro Margara. TESLA: a formally defined event specification language. In Jean Bacon, Peter R. Pietzuch, Joe Sventek, and Ugur Çetintemel, editors, *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems, DEBS 2010, Cambridge, United Kingdom, July 12-15, 2010*, pages 50–61. ACM, 2010. doi: 10.1145/1827418.1827427. URL `https://doi.org/10.1145/1827418.1827427`.

[12] Nikos Giatrakos, Alexander Artikis, Antonios Deligiannakis, and Minos Garofalakis. Complex event recognition in the big data era: A survey. *Proc. VLDB Endow.*, 10(12):1996–1999, aug 2017. ISSN 2150-8097. doi: 10.14778/3137765.3137829. URL `https://doi.org/10.14778/3137765.3137829`.

[13] Elias Alevizos, Anastasios Skarlatidis, Alexander Artikis, and Georgios Paliouras. Probabilistic complex event recognition: A survey. *ACM Comput. Surv.*, 50(5), sep 2017. ISSN 0360-0300. doi: 10.1145/3117809. URL `https://doi.org/10.1145/3117809`.

[14] Jagrati Agrawal, Yanlei Diao, Daniel Gyllstrom, and Neil Immerman. Efficient pattern matching over event streams. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, page 147–160, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605581026. doi: 10.1145/1376616.1376634. URL `https://doi.org/10.1145/1376616.1376634`.

[15] Gianpaolo Cugola and Alessandro Margara. Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys*, 2012.

[16] Alejandro Grez, Cristian Riveros, Martín Ugarte, and Stijn Vansummeren. A formal framework for complex event recognition. *ACM Trans. Database Syst.*, 46(4):16:1–16:49, 2021. doi: 10.1145/3485463. URL `https://doi.org/10.1145/3485463`.

[17] Alejandro Grez, Cristian Riveros, and Martín Ugarte. A Formal Framework for Complex Event Processing. In Pablo Barcelo and Marco Calautti, editors, *22nd International Conference on Database Theory (ICDT 2019)*, volume 127 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 5:1–5:18, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-95977-101-6. doi: 10.4230/LIPIcs.ICDT.2019.5. URL `http://drops.dagstuhl.de/opus/volltexte/2019/10307`.

[18] Alejandro Grez, Cristian Riveros, Martín Ugarte, and Stijn Vansummeren. On the Expressiveness of Languages for Complex Event Recognition. In Carsten Lutz and Jean Christoph Jung, editors, *23rd International Conference on Database Theory (ICDT 2020)*, volume 155 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 15:1–15:17, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. ISBN 978-3-95977-139-9. doi: 10.4230/LIPIcs.ICDT.2020.15. URL `https://drops.dagstuhl.de/opus/volltexte/2020/11939`.

[19] Nicholas Poul Schultz-Møller, Matteo Migliavacca, and Peter Pietzuch. Distributed complex event processing with query rewriting. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, DEBS '09, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605586656. doi: 10.1145/1619258.1619264. URL `https://doi.org/10.1145/1619258.1619264`.

[20] Mo Liu, Elke Rundensteiner, Kara Greenfield, Chetan Gupta, Song Wang, Ismail Ari, and Abhay Mehta. 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, SIGMOD '11, page 889–900, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450306614. doi: 10.1145/1989323.1989416. URL https://doi.org/10.1145/1989323.1989416.

[21] Yuan Mei and Samuel Madden. Zstream: A cost-based query processor for adaptively detecting composite events. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, page 193–206, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605585512. doi: 10.1145/1559845.1559867. URL https://doi.org/10.1145/1559845.1559867.

[22] Darko Anicic, Paul Fodor, Sebastian Rudolph, Roland Stühmer, Nenad Stojanovic, and Rudi Studer. A rule-based language for complex event processing and reasoning. In Pascal Hitzler and Thomas Lukasiewicz, editors, *Web Reasoning and Rule Systems*, pages 42–57, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-15918-3.

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

[24] Alexander Artikis, Anastasios Skarlatidis, François Portet, and Georgios Paliouras. Logic-based event recognition. *The Knowledge Engineering Review*, 27(4):469–506, 2012. doi: 10.1017/S0269888912000264.

[25] Bugra Gedik. Partitioning functions for stateful data parallelism in stream processing. *VLDB J.*, 23(4):517–539, 2014.

[26] Nicolo Rivetti, Leonardo Querzoni, Emmanuelle Anceaume, Yann Busnel, and Bruno Sericola. Efficient key grouping for near-optimal load balancing in stream processing systems. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems, DEBS '15, Oslo, Norway, June 29 - July 3, 2015*, pages 80–91, 2015.

[27] Yossi Azar, Andrei Z. Broder, Anna R. Karlin, and Eli Upfal. Balanced allocations. *SIAM J. Comput.*, 29(1):180–200, 1999.

[28] Muhammad Anis Uddin Nasir, Gianmarco De Francisci Morales, David García-Soriano, Nicolas Kourtellis, and Marco Serafini. The power of both choices: Practical load balancing for distributed stream processing

engines. In *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*, pages 137–148, 2015.

[29] Muhammad Anis Uddin Nasir, Gianmarco De Francisci Morales, Nicolas Kourtellis, and Marco Serafini. When two choices are not enough: Balancing at scale in distributed stream processing. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, pages 589–600. IEEE, 2016.

[30] Nikos R. Katsipoulakis, Alexandros Labrinidis, and Panos K. Chrysanthis. A holistic view of stream partitioning costs. *PVLDB*, 10(11):1286–1297, 2017.

[31] Matthieu Caneill, Ahmed El-Rheddane, Vincent Leroy, and Noël De Palma. Locality-aware routing in stateful streaming applications. In *Proceedings of the 17th International Middleware Conference, Trento, Italy, December 12 - 16, 2016*, page 4, 2016.

[32] Flinkcep - complex event processing for flink, Jan 2022. URL `https://nightlies.apache.org/flink/flink-docs-release-1.13/docs/libs/cep/`.

[33] Martin Hirzel. Partition and compose: Parallel complex event processing. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, DEBS '12, page 191–200, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450313155. doi: 10.1145/2335484.2335506. URL `https://doi.org/10.1145/2335484.2335506`.

[34] Ruben Mayer, Muhammad Adnan Tariq, and Kurt Rothermel. Minimizing communication overhead in window-based parallel complex event processing. In *Proceedings of the 11th ACM International Conference on Distributed and Event-Based Systems*, DEBS '17, page 54–65, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450350655. doi: 10.1145/3093742.3093914. URL `https://doi.org/10.1145/3093742.3093914`.

[35] Lars Brenna, Johannes Gehrke, Mingsheng Hong, and Dag Johansen. Distributed event stream processing with non-deterministic finite automata. In *Proceedings of the Third ACM International Conference on Distributed*

Event-Based Systems, DEBS 2009, Nashville, Tennessee, USA, July 6-9, 2009, 2009.

[36] Lars Brenna, Alan J. Demers, Johannes Gehrke, Mingsheng Hong, Joel Ossher, Biswanath Panda, Mirek Riedewald, Mohit Thatte, and Walker M. White. Cayuga: a high-performance event processing engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*, pages 1100–1102, 2007.

[37] Martin Hirzel. Partition and compose: parallel complex event processing. In *Proceedings of the Sixth ACM International Conference on Distributed Event-Based Systems, DEBS 2012, Berlin, Germany, July 16-20, 2012*, pages 191–200, 2012.

[38] Martin Hirzel, Scott Schneider, and Bugra Gedik. SPL: an extensible language for distributed stream processing. *ACM Trans. Program. Lang. Syst.*, 39(1):5:1–5:39, 2017.

[39] Cagri Balkesen, Nihal Dindar, Matthias Wetter, and Nesime Tatbul. RIP: run-based intra-query parallelism for scalable complex event processing. In *The 7th ACM International Conference on Distributed Event-Based Systems, DEBS '13, Arlington, TX, USA - June 29 - July 03, 2013*, pages 3–14, 2013.

[40] Ajay D. Kshemkalyani and Mukesh Singhal. *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge University Press, March 2011. ISBN 9780521189842. URL `https://www.cs.uic.edu/~ajayk/DCS-Book`.

[41] Jay Earley. An efficient context-free parsing algorithm. *Commun. ACM*, 13(2):94–102, feb 1970. ISSN 0001-0782. doi: 10.1145/362007.362035. URL `https://doi.org/10.1145/362007.362035`.

[42] M. Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10): 1094–1104, 2001. doi: 10.1109/71.963420.

[43] Moshe Vardi. The complexity of relational query languages (extended abstract). pages 137–146, 01 1982. doi: 10.1145/800070.802186.

[44] Bill Venners. The java virtual machine. *Java and the Java virtual machine: definition, verification, validation*, 1998.

[45] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the scala programming language. 2004.

[46] Derek Wyatt. *Akka concurrency*. Artima Incorporation, 2013.

[47] Sbt: The interactive build tool, Jan 2022. URL `https://www.scala-sbt.org/index.html`.

[48] Acm sigmod 2022 availability & reproducibility, Jan 2022. URL `https://reproducibility.sigmod.org/`.

[49] Medhabi Ray, Chuan Lei, and Elke A. Rundensteiner. Scalable pattern sharing on event streams*. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 495–510, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450335317. doi: 10.1145/2882903.2882947. URL `https://doi.org/10.1145/2882903.2882947`.