Towards Streaming Evaluation of Queries with Correlation in Complex Event Processing

Alejandro Grez
Pontificia Universidad Católica de Chile, Santiago, Chile
Millennium Institute for Foundational Research on Data, Santiago, Chile
ajgrez@uc.cl

Cristian Riveros
Pontificia Universidad Católica de Chile, Santiago, Chile
Millennium Institute for Foundational Research on Data, Santiago, Chile
cristian.riveros@uc.cl

Abstract

Complex event processing (CEP) has gained a lot of attention for evaluating complex patterns over high-throughput data streams. Recently, new algorithms for the evaluation of CEP patterns have emerged with strong guarantees of efficiency, i.e. constant update-time per tuple and constant-delay enumeration. Unfortunately, these techniques are restricted for patterns with local filters, limiting the possibility of using joins for correlating the data of events that are far apart.

In this paper, we embark on the search for efficient evaluation algorithms of CEP patterns with joins. We start by formalizing the so-called partition-by operator, a standard operator in data stream management systems to correlate contiguous events on streams. Although this operator is a restricted version of a join query, we show that partition-by (without iteration) is equally expressive as hierarchical queries, the biggest class of full conjunctive queries that can be evaluated with constant update-time and constant-delay enumeration over streams. To evaluate queries with partition-by we introduce an automata model, called chain complex event automata (chain-CEA), an extension of complex event automata that can compare data values by using equalities and disequalities. We show that this model admits determinization and is expressive enough to capture queries with partition-by. More importantly, we provide an algorithm with constant update time and constant delay enumeration for evaluating any query definable by chain-CEA, showing that all CEP queries with partition-by can be evaluated with these strong guarantees of efficiency.

2012 ACM Subject Classification Information systems → Data streams; Theory of computation → Database query processing and optimization (theory); Theory of computation → Formal languages and automata theory; Theory of computation → Automata extensions

Keywords and phrases Complex event processing, Query languages, Correlation, Constant delay enumeration.

Digital Object Identifier 10.4230/LIPIcs.ICDT.2020.14

Funding A. Grez and C. Riveros were partially funded by the Millennium Institute for Foundational Research on Data.

1 Introduction

Streaming query evaluation is the most crucial problem in complex event processing (CEP). Given a CEP query Q, the streaming evaluation of Q over a stream consists in continuously reading events and outputting all complex events (i.e. sets of events) as soon as the last event that fires Q arrives. This streaming evaluation can be divided in two parts: (1) the process that continuously reads events and updates the state of the system whenever a new event arrives and (2) the process that outputs (i.e. enumerates) all complex events that satisfy the query. Both processes are required to run separately in such a way that the update process calls the enumeration process whenever a new output is found [17].

Given the high-throughput data streams in areas like Network Intrusion Detection [27], Industrial Control Systems [19] or Real-Time Analytics [28], the time and space used by these two processes must be severely restricted. As proposed in [7, 17, 22], an efficient streaming evaluation process should satisfy at least the following two ideals: the update process must take constant time per new event and the enumeration process must take constant delay between two consecutive outputs. Intuitively, this is the best that a CEP system can aim for efficiently processing high-throughput data streams in practice. In [17] a streaming evaluation algorithm with constant update time per event and constant delay enumeration was shown for a meaningful core of CEP query languages when only local filters are allowed. Unfortunately, not all relevant queries in CEP can be evaluated with these strong guarantees, which fosters the search of query operators that allow efficient evaluation.

One of the key features in CEP is correlation [12]: to associate different events that might occur arbitrarily far in the input stream. Verifying that two users have the same id, or verifying an increasing sequence of temperature events, are some examples of how correlation is used in CEP. The most basic operator for adding correlation in CEP are equalities, namely, joining two events which have the same data value. Unfortunately, the evaluation of join queries is a difficult task even in a static setting [1], stressing the difficulties of finding efficient evaluation algorithms of CEP queries with equality predicates. One special operator usually included in CEP systems [5, 31, 15] for correlating events is partition-by [5] (also referred as *segmentation-oriented context* in [16] or just *context* in [15]). As the name suggests, this operator breaks up the events of a stream into partitions where all events of the same partition have the same data value. Despite being a useful operator in CEP, there is a lack of research in evaluating partition-by queries with solid efficiency guarantees, and usually this operator is severely restricted in CEP systems [31].

In this paper, we embark on the search for efficient evaluation of CEP queries with correlation when equality and disequality predicates are used. We first formalize the partition-by operator by extending Complex Event Logic (CEL) [17, 18] with a simple and compositional semantics. To motivate the expressive power of partition-by, we show that CEL with partition-by (but without iteration) is equally expressive as hierarchical queries [7, 22], the biggest subclass of conjunctive queries (CQ) that can be evaluated with constant update time and constant delay enumeration [7].

With a well-defined operator for doing correlation, we study the evaluation of partition-by through a machine model that we called chain Complex Event Automata (chain-CEA), an extension of complex event automata with equality and disequality predicates [17]. Although automata models over data words usually do not have good closure properties [29], we show that the chain-CEA model admits determinization and is expressive enough to capture all CEL queries with partition-by. The most important result of the paper is a streaming evaluation algorithm for the full class of chain-CEA, with constant update time and constant delay enumeration. In particular, this shows that all queries with partition-by can be evaluated efficiently in a streaming fashion.

Related work. Streaming query evaluation has been studied in the context of data stream management systems (DSMS) [5] and complex event processing (CEP) [31, 12, 21]. The notion of constant update time per tuple/event and constant delay enumeration has not been considered until recently [25, 20, 7] and, furthermore, in CEP systems these strong guarantees of efficiency have not been adopted yet [17]. Therefore, the algorithmic approach in CEP systems for evaluating queries with correlation is incomparable to our approach.

New techniques in dynamic query evaluation [8, 2, 9] have recently attracted a lot of attention [7, 22, 23]. In [7, 22], the streaming evaluation of CQ is considered but this does not include queries with order. In [23], inequalities over atoms are considered, but only for

the case of CQ. Our setting also includes disjunction and iteration (but not conjunction), which makes our work orthogonal to the work in [7, 22, 23].

Register automata [24] have been extensively studied in the context of automata theory and XML [29]. Nevertheless, this model has not been studied in the context of CEP and efficient query evaluation. Recently, in [4] a similar extension of complex event automata with registers was proposed. However, this work does not study the determinization and evaluation of this model with constant update time and constant delay enumeration.

2 Preliminaries

In this section, we recall the formal definitions for streams and complex events [17], and give a simplified version of Complex Event Logic (CEL) [18, 17], originally called SO-CEL in [18]. We will later use CEL as a base language to model the partition-by operator.

Streams and complex events. Let $\mathbf{A}$ be a set of *attribute names* and $\mathbf{D}$ be an infinite set of values. A database schema $\mathcal{R}$ is a finite set of relation names, where each relation name $R \in \mathcal{R}$ is associated to a tuple of attributes denoted by $\text{att}(R)$. If R is a relation name, then an R-tuple is a function $t : \text{att}(R) \to \mathbf{D}$. Given $a \in \text{att}(R)$, we write $t.a$ to denote the value $t(a)$, and $\text{att}(t)$ to denote $\text{dom}(t)$. We say that the type of an R-tuple t is R, and denote this by $\text{type}(t) = R$. For any relation name R, $\text{tuples}(R)$ denotes the set of all possible R-tuples. Similarly, for any database schema $\mathcal{R}$, $\text{tuples}(\mathcal{R}) = \bigcup_{R \in \mathcal{R}} \text{tuples}(R)$. Given a schema $\mathcal{R}$, an *$\mathcal{R}$-stream* S is an infinite sequence $S = t_1 t_2 \dots$ where $t_i \in \text{tuples}(\mathcal{R})$. When $\mathcal{R}$ is clear from the context, we refer to S simply as a stream. Given a stream $S = t_1 t_2 \dots$ and a position $i \in \mathbb{N}$, the i-th element of S is denoted by $S[i] = t_i$.

A complex event C is defined as a non-empty and finite set of natural numbers. Intuitively, given a stream $S = t_1 t_2 \dots$ a complex event $C = \{i_1, \dots, i_n\}$ determines the set of tuples $\{t_{i_1}, \dots, t_{i_n}\}$ and, thus, C represents the set of relevant events. We denote by $\min(C)$ and $\max(C)$ the minimum and maximum element of C, respectively. Given two complex events C_1 and C_2, we write $C_1 \cdot C_2$ for their *concatenation*, which is defined as $C_1 \cdot C_2 := C_1 \cup C_2$ whenever $\max(C_1) < \min(C_2)$ and empty otherwise. Given a complex event C we define $S[C] = \{S[i] \mid i \in C\}$, namely, the set of tuples in S positioned at the indices specified by C.

Complex event logic (CEL). Let $\mathbf{X}$ be a finite set of monadic second-order (SO) variables. An SO predicate of arity n is an n-ary relation P over sets of tuples, $P \subseteq (2^{\text{tuples}(\mathcal{R})})^n$. We write $\text{arity}(P) = n$. Let $\mathbf{P}$ be a set of SO predicates. An atom over $\mathbf{P}$ is an expression of the form $P(X_1, \dots, X_n)$ where $P \in \mathbf{P}$ is a predicate of arity n, and $X_1, \dots, X_n \in \mathbf{X}$ (we also write $P(\bar{X})$ for $P(X_1, \dots, X_n)$). A CEL formula is defined by the following syntax:

$$\varphi \;:=\; R \;\mid\; \varphi \;\texttt{IN}\; X \;\mid\; \varphi \;\texttt{FILTER}\; P(\bar{X}) \;\mid\; \varphi \;\texttt{OR}\; \varphi \;\mid\; \varphi\,;\varphi \;\mid\; \varphi+$$

where R ranges over relation names, X over variables in $\mathbf{X}$ and $P(\bar{X})$ over atoms in $\mathbf{P}$. We say φ is an atomic formula if $\varphi = R$.

A valuation is a function $\mu : \mathbf{X} \to 2^{\mathbb{N}}$ such that $\mu(X)$ is a complex event for every $X \in \mathbf{X}$. We define the support of μ by $\text{supp}(\mu) = \bigcup_{X \in \mathbf{X}} \mu(X)$, and the union between μ_1 and μ_2 as $(\mu_1 \cup \mu_2)(X) = \mu_1(X) \cup \mu_2(X)$ for every $X \in \mathbf{X}$. Given a formula φ and a stream S, we say that a complex event C belongs to the evaluation of φ over S under the valuation μ (denoted by $C \in [\![\varphi]\!](S, \mu)$) if one of the following conditions holds:

- $\varphi = R$, $C = \{i\}$, $\text{type}(S[i]) = R$ and $\mu(X) = \emptyset$ for every X.
- $\varphi = \rho$ IN X, $\mu(X) = C$, and there exists a valuation μ' such that $C \in [\![\rho]\!](S, \mu')$ and $\mu(Y) = \mu'(Y)$ for all $Y \neq X$.

type	T	R	R	R	T	R	T	R	...
id	123	155	165	223	252	352	355	411	...
user-id	11	48	48	48	13	13	33	79	...
tweet-id		123	343	123		252		123	...
post/reply	#vote	#ihate	#ihate	#ihate	#vote	#ihate	#ihate	#stop	...
index	1	2	3	4	5	6	7	8	...

Figure 1 A stream S of events from Twitter. T are tweets with an id, a user-id and a post message, and R are responses with an id, a user-id, a tweet-id, and a reply message. The last line is the index of each event in the stream, respectively.

- $\varphi = \rho$ `FILTER` $P(X_1, \ldots, X_n)$, $C \in [\![\rho]\!](S, \mu)$ and $(S[\mu(X_1)], \ldots, S[\mu(X_n)]) \in P$.
- $\varphi = \rho_1$ `OR` ρ_2 and ($C \in [\![\rho_1]\!](S, \mu)$ or $C \in [\![\rho_2]\!](S, \mu)$).
- $\varphi = \rho_1 \,;\, \rho_2$ and there exist complex events C_1 and C_2, and valuations μ_1 and μ_2 such that $C = C_1 \cdot C_2$, $\mu = \mu_1 \cup \mu_2$, $C_1 \in [\![\rho_1]\!](S, \mu_1)$ and $C_2 \in [\![\rho_2]\!](S, \mu_2)$.
- $\varphi = \rho+$, and $C \in [\![\rho]\!](S, \mu)$ or $C \in [\![\rho \,;\, \rho+]\!](S, \mu)$.

We say that C belongs to the evaluation of a CEL formula φ over S at position $n \in \mathbb{N}$, denoted by $C \in [\![\varphi]\!]_n(S)$, if $C \in [\![\varphi]\!](S, \mu)$ for some valuation μ, and $\max(C) = n$.

▶ **Example 1.** As a running example, suppose that we consider the stream from Twitter. For the sake of simplification, suppose that the stream is composed just by tweets (T) and replies (R). A tweet is composed by three attributes: an `id`, a `user-id` and a `post` message. Instead, a reply is composed by four attributes: an `id`, a `user-id`, a `tweet-id` of the replied message, and a `reply` message. Figure 1 shows an example of a stream with this schema.

As an example of a CEL formula, suppose that a journalist wants to detect all pairs of events composed by a tweet followed by a response containing '#voteforjohn' and '#ihatejohn', respectively, representing "hot" debates in Twitter about the election of a candidate called John. This query can easily be defined with the following CEL-formula:

$$\varphi_1 \;:=\; (T \texttt{ IN } X ; R \texttt{ IN } Y) \texttt{ FILTER } (X.\texttt{post} = \text{'\#vote'} \texttt{ AND } Y.\texttt{reply} = \text{'\#ihate'})$$

Here we make use of three operators: sequencing (;) to say we want to find complex events consisting of a T-tuple followed by an R-tuple; variable names (`IN`) to assign variables X and Y to T and R, respectively; and `FILTER` to define the conditions that the events must satisfy. We use conjunction (i.e., `AND`) as a syntactic sugar, which is short for applying a `FILTER` operator for each predicate of the conjunction. Predicates $X.\texttt{post} =$ '#vote' and $Y.\texttt{reply} =$ '#ihate' are basically restricting the T and R tuples t and r so that $t.\texttt{post}$ contains '#voteforjohn' and $r.\texttt{reply}$ contains '#ihatejohn'. Given a stream $S = t_1 t_2 \ldots$ and a valuation μ, one can easily check that $[\![\varphi_1]\!](S, \mu)$ contains complex events of the form $\{k_1, k_2\}$ with $k_1 < k_2$ such that $t_{k_1}.\texttt{post}$ contains '#voteforjohn' and $t_{k_2}.\texttt{reply}$ contains '#ihatejohn'. For example, $\{1, 2\}$, $\{1, 3\}$ and $\{5, 6\}$ in Figure 1 are some outputs of φ_1 over S. Note that the replies are not necessarily replying to the tweet they are paired with, contrary to what one would like. We address this issue in the next section.

▶ **Example 2.** Suppose now that we want to find all sequences of debates that start with a tweet with '#voteforjohn', are followed by one or more responses with '#ihatejohn', and end with a response containing '#stophating'. This query can easily be defined in CEL using (+):

$$\varphi_2 \;=\; \big(T \texttt{ IN } X \;;\; \big(R+\big) \texttt{ IN } Y \;;\; R \texttt{ IN } Z\big) \texttt{ FILTER } \big(X.\texttt{post} = \text{'\#vote'} \texttt{ AND } Y.\texttt{reply} = \text{'\#ihate'} \texttt{ AND } Z.\texttt{reply} = \text{'\#stop'}\big)$$

In φ_2 we use the $(+)$ operator to extract an unbounded sequence of replies, which are then assigned to Y so that the predicate $Y.\texttt{reply} =$ '#ihate' filters only the sequences where all tuples contain '#ihatejohn' (i.e. all tuples t in the complex event represented by Y must satisfy $t.\texttt{reply} =$ '#ihate'). The other predicates are used to ensure that the T-tuple contains '#voteforjohn' and the last R-tuple contains '#stophating'. Finally, one can check that φ_2 defines the desired property. For example, if we evaluate φ_2 over S in Figure 1, then $\{1, 2, 4, 8\}$ and $\{1, 3, 4, 6, 8\}$ will be some outputs in $[\![\varphi_2]\!]_8(S)$.

A relevant feature in CEP is to skip arbitrary events when a formula is evaluated [12]. For example, for φ_1 it would make no sense looking for two contiguous events T and R. For this reason, the sequencing operator allows to skip an arbitrary number of events between two relevant events. The iteration operator has a similar semantics, which results in that for every sequence captured by it, the powerset of events is also captured. To remedy this problem CEL also includes the so-called selection strategies [17, 18], namely, operators for filtering the set of output to a meaningful subset. In this paper, our results do not include the evaluation over selection strategies. We leave this for future work.

CEL fragments and unary predicates. Given a set O of operators (e.g. `OR`, $+$), we define CEL[O] to be the set of CEL formulas constructed from atomic formulas, `IN`, and operators in O. For example, φ_1 is in CEL[;, `FILTER`] and φ_2 is in CEL[;, `FILTER`, $+$]. Furthermore, we define CEL+O as the set of all CEL formulas extended with O.

Although CEL does not restrict the set of predicates that can be used by `FILTER`, not necessarily all predicates can be evaluated efficiently (or are even computable). For this reason, in [18] the analysis of CEL was restricted to SO-extensions of first-order unary predicates. Formally, let $\mathbf{U}$ be the set of all unary predicates over tuples, i.e. $\mathbf{U} = \{P \subseteq \text{tuples}(\mathcal{R})\}$. Given $P \in \mathbf{U}$ we define the SO-extension $P^{\text{SO}} \subseteq 2^{\text{tuples}(\mathcal{R})}$ of P such that $A \in P^{\text{SO}}$ if, and only if, $t \in P$ for all $t \in A$. We denote by $\mathbf{U}^{\text{SO}}$ the set of all SO-extensions of predicates in $\mathbf{U}$. When writing predicates with SO variables we are referring to the SO extension of the first order predicate. For example, $P := x.\texttt{post} =$ '#vote' is a predicate in $\mathbf{U}$ such that $t \in P$ iff t has the attribute `post` and $t.\texttt{post}$ contains #voteforjohn. Then $P^{\text{SO}} := (X.\texttt{post} =$ '#vote') is the SO-extension of P that defines all complex events whose tuples satisfy P.

In [17, 18], it was shown that all CEL formulas restricted to predicates in $\mathbf{U}^{\text{SO}}$ can be evaluated efficiently. For this reason, from now on we assume that for any fragment or extension of CEL, all `FILTER` are restricted to predicates in $\mathbf{U}^{\text{SO}}$.

Streaming evaluation with constant-delay enumeration. As it is standard in the literature [7, 22], we consider evaluation algorithms on Random Access Machines (RAM) with addition and uniform cost measure [3]. Furthermore, we assume the existence of a key-value index (e.g. hash index) that allows insertions and deletions in $O(1)$ time and the index uses space linear in the number of insertions. In other words, we assume to have perfect hashing of linear size [10]. Although this is not realistic for practical computers, it can be simulated with a $O(\log(n))$-factor in the evaluation process with n the number of insertions in the index. Our complexity analysis is always in data complexity, namely, we assume that the CEL query φ and the schema $\mathcal{R}$ of the stream are fixed. Finally, we restrict the set $\mathbf{U}$ to unary predicates with constant time evaluation, namely, for every predicate P in $\mathbf{U}$ and every tuple t, we assume that checking whether $t \in P$ takes constant-time.

For efficient evaluation in CEP, we adapt the notion of constant-delay used in [6, 11] for streaming evaluation. Our evaluation process is a streaming algorithm divided in two parts: (1) consuming new events and updating the internal memory of the system and (2)

generating complex events from the internal memory of the system. A streaming evaluation algorithm with *constant update time* and *constant-delay enumeration* is an algorithm that reads a stream $S = t_1 t_2 \dots$ sequentially and evaluates a formula φ over S such that (1) the time spent between reading t_i and t_{i+1} is bounded by $\mathcal{O}(|t_i|)$, and (2) it maintains a data structure D in memory, such that after reading t_n, the set $[\![\varphi]\!]_n(S)$ can be enumerated from D with constant-delay. The enumeration requires the existence of a routine ENUMERATE that enumerates $[\![\varphi]\!]_n(S) = \{C_1, C_2, \dots, C_m\}$ one by one without repetitions. We call delay(C_i) the time it takes between enumerating C_i and C_{i+1}, and we say ENUMERATE runs with constant-delay if there exists a constant k depending only on φ such that delay$(C_i) = k \cdot |C_i|$ for all i. We remark that (1) is a natural restriction for a streaming algorithm, while (2) is the minimum requirement if an arbitrarily large set of arbitrarily large outputs must be produced [30]. Given that our analysis is in data complexity (i.e. φ and $\mathcal{R}$ are fixed), then the update time $\mathcal{O}(|t|)$ has a hidden factor that depends on $|\varphi|$ and $|\mathcal{R}|$.

3 Partition-by: syntax and semantics

Our main motivation in this paper is to study queries with correlation in CEP. One of the main operators for joining multiple events is partition-by [15, 31] (also referred as *segmentation-oriented context* in [16] or just *context* in [15]). Intuitively, events in a stream are usually correlated by an attribute that has the same value, e.g. an id. Then this attribute is "partitioning" the stream in multiple streams, where all events of the same stream contain the same value. In this section, we formally define the `PART-BY` operator in CEL, and motivate its usefulness by showing that it is expressive enough to define hierarchical queries.

Given two formulas φ_1 and φ_2, we denote by $\varphi_1 \subseteq \varphi_2$ when φ_1 is a subformula of φ_2. Consider a formula φ and variables $X_1, \dots, X_k$ of φ. We say that $X_1, \dots, X_k$ form a *variable cover* of φ if, for every atomic subformula ρ of φ, i.e. $\rho \subseteq \varphi$ and $\rho = R$ for some R, there is some $i \leq k$ and formula $\psi = \psi'$ `IN` X_i such that $\rho \subseteq \psi \subseteq \varphi$, namely, all the events captured by atomic subformulas will be captured by some of the variables $X_1, \dots, X_k$ in φ. For example, in Example 2 variables X, Y and Z form a variable cover of φ_2.

We extend the syntax of CEL with the operator `PART-BY` as follows. A formula φ is in CEL+`PART-BY` if it satisfies the syntax of CEL, plus the following rule:

$$\varphi \;:=\; \varphi \texttt{ PART-BY } [X_1.a_1, \dots, X_k.a_k]$$

where $X_1, \dots, X_k \in \mathbf{X}$ form a variable cover of φ and $a_1, \dots, a_k \in \mathbf{A}$ are attributes. The semantics of the `PART-BY` operator is defined as follows. Consider a complex event C, a stream $S = t_1 t_2 \dots$ and a valuation μ. Then, $C \in [\![\varphi \texttt{ PART-BY } [X_1.a_1, \dots, X_k.a_k]]\!](S, \mu)$ if $C \in [\![\varphi]\!](S, \mu)$ and for all $i, j \in \mathbb{N}$, $l \in \mu(X_i)$ and $m \in \mu(X_j)$, it holds that $S[l].a_i = S[m].a_j$. Thus, all events must contain the same data value in their attributes. For the case we only want to partition using a single attribute a that is common among all events (e.g. an id), we add the syntactic sugar φ `PART-BY` $[a]$, which is defined as φ `PART-BY` $[a] := (\varphi$ `IN` $X)$ `PART-BY` $[X.a]$, where X is a fresh variable that does not appear in φ. Clearly, X is a variable cover of φ.

▶ **Example 3.** In Example 1 we wanted to extract all pairs of tweets and replies that contain #voteforjohn and #ihatejohn, respectively. Although φ_1 extract these complex events, it fails to relate a reply with the tweet is replying to. For this, we can use the partition-by operator as follows:

$$\varphi_1^* \;:=\; \big((T \texttt{ IN } X; R \texttt{ IN } Y) \texttt{ FILTER } (X.\texttt{post} = \text{`\#vote'} \texttt{ AND } Y.\texttt{reply} = \#\text{ihate'})\big) \texttt{ PART-BY } (X.\texttt{id}, Y.\texttt{tweet-id})$$

Clearly, X, Y form a variable cover of φ_1. Furthermore, PART-BY restricts the output to pairs t and r with $t.\texttt{id} = r.\texttt{tweet-id}$. In Figure 1 now only $\{1,2\}$, $\{1,4\}$ and $\{5,6\}$ are in $[\![\varphi_1^*]\!](S)$.

▶ **Example 4.** Now, we want to restrict formula φ_2 in Example 2 in order to correlate tweets and replies in a meaningful way. Suppose that we want to restrict φ_2 such that all replies are replying to T and all #ihatejohn replies are from the same user. Then we can extend φ_2 with PART-BY to impose these restrictions (we omit the filters for the sake of readability):

$$\varphi_2^* \;=\; \Big[\big(T \texttt{ IN } X \;;\; \big(R + \big) \texttt{ PART-BY } (\texttt{user-id}) \texttt{ IN } Y \;;\; R \texttt{ IN } Z\big) \texttt{ FILTER } (\cdots)\Big] \\ \texttt{PART-BY } (X.\texttt{id}, Y.\texttt{tweet-id}, Z.\texttt{tweet-id})$$

This formula shows the advantage of using nesting of PART-BY. The internal PART-BY over attribute user-id restricts all #ihatejohn replies to have the same identifier, namely, they come from the same user. Then the external PART-BY forces all replies to have the same tweet-id as the first tweet and, therefore, they are replies of the same tweet. In Figure 1, $\{1,3,4,6,8\}$ is no longer an output but $\{1,2,4,8\}$ still is.

Partition-by and hierarchical queries. Partition-by models a join operator that usually appears in CEP systems [5, 15, 31]. Although this operator can be considered rather restrictive, interestingly, it is related to the class of hierarchical queries [13, 26], the biggest class of conjunctive queries without projection that can be evaluated in a streaming fashion [7, 22]. To formally define hierarchical queries we first introduce some notation. Given a database schema $\mathcal{R}$, we assume an arbitrary total order $<$ over the attribute names $\mathbf{A}$. For $R \in \mathcal{R}$ with $\text{att}(R) = \{a_1, \ldots, a_k\}$ and $a_1 < \ldots < a_k$, we write $R(x_1, \ldots, x_k)$ for variables $x_1, \ldots, x_k$ to denote that x_i is assigned to attribute a_i. We call $R(x_1, \ldots, x_k)$ an atom. A (full) conjunctive query Q is an expression $R_1(\bar{x}_1) \wedge \ldots \wedge R_k(\bar{x}_k)$ where each $R_i(\bar{x}_i)$ is an atom (i.e. we restrict our discussion to CQ without projection). Given a conjunctive query Q with k atoms and a stream $S = t_1 t_2 \ldots$ we say that a complex event C satisfies Q if $|C| \leq k$ and $\{t_i \mid i \in C\} \models Q$. We define $[\![Q]\!]_n(S)$ as all complex events C that satisfy Q and $\max(C) = n$.

From now on, we restrict our analysis to hierarchical conjuctive queries. Specifically, for a variable x define the set $\text{atom}(x)$ of all atoms in Q where x is mentioned. Then Q is hierarchical [13, 26] if for every x and y it holds that either $\text{atom}(x) \subseteq \text{atom}(y)$, $\text{atom}(x) \supseteq \text{atom}(y)$, or $\text{atom}(x) \cap \text{atom}(y) = \emptyset$. For example, the query $R(x) \wedge S(x,y)$ is hierarchical and $R(x) \wedge S(x,y) \wedge T(y)$ is not.

Unfortunately, CEL+PART-BY is not enough to capture the expressiveness of hierarchical queries. The reason is that partition-by combined with sequencing forces all events with correlated values to be "adjacent". On the other hand, hierarchical queries do not impose any order over tuples. For this reason, we consider the ALL-operator, a standard CEP operator studied in [18]. Formally, given formulas φ_1 and φ_2 we define the formula $\varphi_1 \texttt{ ALL } \varphi_2$ such that for a stream S and valuation μ it holds that $C \in [\![\varphi_1 \texttt{ ALL } \varphi_2]\!](S, \mu)$ if there exist complex events C_1, C_2, and valuations μ_1, μ_2 such that $C_1 \in [\![\rho_1]\!](S, \mu_1)$, $C_2 \in [\![\rho_2]\!](S, \mu_2)$, $C = C_1 \cup C_2$ and $\mu = \mu_1 \cup \mu_2$. In other words, ALL makes the pair union of complex events coming from evaluating φ_1 and φ_2, separately. Interestingly, CEL[ALL, PART-BY] captures exactly the expressiveness of hierarchical queries.

▶ **Proposition 5.** *For every hierarchical query Q, there is a formula φ in* CEL[ALL, PART-BY] *such that $[\![Q]\!]_n(S) = [\![\varphi]\!]_n(S)$ for every stream S and position n, and vice versa.*

The previous proposition shows the motivation of partition-by from the perspective of hierarchical CQ. Although CEL+PART-BY is not enough to capture the expressibility of hierarchical CQ, it shows that partition-by is related with a subclass of CQ that can be evaluated efficiently in a streaming fashion.

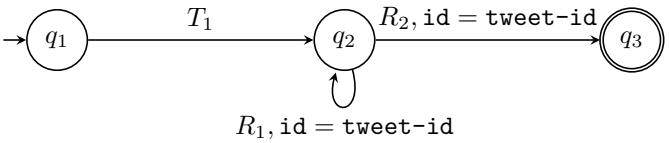

Figure 2 An example of chain-CEA with unary predicates $T_1 := \text{type}(T) \wedge \text{post} = \text{`\#vote'}$, $R_1 := \text{type}(R) \wedge \text{reply} = \text{`\#ihate'}$, and $R_2 := \text{type}(R) \wedge \text{reply} = \text{`\#stop'}$.

4 Chain complex event automata

Similarly to [17], we base our evaluation approach on an automata model to represent CEL+`PART-BY`. We present an automata model, called chain Complex Event Automata (chain-CEA), and show that each formula in CEL+`PART-BY` can be represented by this model.

In order to express the `PART-BY` operator, the automata model needs to be able to handle equality predicates. Given attributes $a, b \in \mathbf{A}$ define the equality and disequality predicates as $P_{a=b} = \{(t_1, t_2) \mid a \in \text{att}(t_1) \wedge b \in \text{att}(t_2) \wedge t_1.a = t_2.b\}$ and $P_{a\neq b} = \text{tuples}(R) \setminus P_{a=b}$. A *conjunctive binary predicate*, or binary predicate for short, is a predicate B that is a conjunction of equality and disequality predicates, i.e., $B = \bigcap_{i=1}^{n}(P_{a_i \sim b_i})$, where $a_i, b_i \in \mathbf{A}$ and $\sim_i \in \{=, \neq\}$. For simplicity, we usually drop the predicate notation and denote B simply as $\bigwedge_{i=1}^{n}(a_i \sim b_i)$. For example, $(a = b \wedge c \neq d)$ represents the predicate $B = P_{a=b} \cap P_{c\neq d}$, and thus $(t_1, t_2) \in B$ if $t_1.a = t_2.b$ and, if $c \in \text{att}(t_1)$ and $d \in \text{att}(t_2)$, then $t_1.c \neq t_2.d$. To separate equalities and disequalities from B, we will usually denote $B = B_= \wedge B_\neq$ where $B_=$ and $B_\neq$ are binary predicates composed only by equalities and disequalities, respectively. We denote by $\mathbf{B}$ the set of all binary predicates.

A *chain complex event automaton* (chain-CEA) is a tuple $\mathcal{A} = (Q, \Delta, I, F)$ where Q is a finite set of states, the transition relation Δ is a set of tuples (p, P, B, q), where $p, q \in Q$, $P \in \mathbf{U}$ and $B \in \mathbf{B}$, and $I, F \subseteq Q$ are the initial and final set of states, respectively. A configuration of $\mathcal{A}$ is defined by a state and a position in the stream, i.e. a pair $(q, i) \in Q \times \mathbb{N}$. An initial configuration is a pair (q, i) where $q \in I$ and $i = 0$. A run ρ of $\mathcal{A}$ over a stream $S = t_1 t_2 \ldots$ is a sequence of configurations: $(q_0, i_0) \xrightarrow{P_1/B_1} (q_1, i_1) \xrightarrow{P_2/B_2} \ldots \xrightarrow{P_n/B_n} (q_n, i_n)$ such that (q_0, i_0) is an initial configuration and, for every $j \leq n$: $i_{j-1} < i_j$, $(q_{j-1}, P_j, B_j, q_j) \in \Delta$, $t_{i_j} \in P_j$ and $(t_{i_{j-1}}, t_{i_j}) \in B_j$, where we consider t_0 being the empty tuple with no attributes. Further, the run ρ above induces the complex event $C_\rho = \{i_j \mid j > 0\}$. We say that ρ is an accepting run if $q_n \in F$. We define the set of complex events of $\mathcal{A}$ over S ending at position n as $[\![\mathcal{A}]\!]_n(S) = \{C_\rho \mid \rho \text{ is an accepting run of } \mathcal{A} \text{ and } \max\{C\} = n\}$.

It is worth noting that, even though only conjunctions and negations of equality predicates are allowed, in practice every logical combination (i.e. $\wedge$, $\vee$ and $\neg$) can be managed by simulating disjunction using multiple transitions. However, we need this restricted definition to later simplify the evaluation algorithm in Section 5.

▶ **Example 6.** Recall our complex events in Example 2 of a tweet with #voteforjohn, followed by one or more responses with #ihatejohn, and ending with a response saying #stophating. Suppose now that instead of correlating all responses with the first tweet, we want to extract a chain of responses, namely, for each contiguous responses r_1 and r_2 it holds that $r_1.\texttt{id} = r_2.\texttt{tweet-id}$ (i.e. r_2 is a reply of r_1). In Figure 2 we show a chain-CEA defining this query. If the automaton is in the initial state q_1 and receives a tweet t event containing #voteforjohn, it moves to q_2 and stores t. Then for each response r containing #ihatejohn whose `tweet-id` is equivalent to the id of the stored event, it forgets that event and stores r. Finally, when it receives an R-event containing #stophating which is responding the stored event, it reaches a final state.

The previous example shows a meaningful CEP query definable by a chain-CEA. This type of queries are very useful in practice (see for example query (7) in [12]). The next result shows that chain-CEA are expressive enough to cover the class of CEL+PART-BY formulas.

▶ **Proposition 7.** *For any formula φ in* CEL+PART-BY*, there exists a chain-CEA $\mathcal{A}$ such that $[\![\varphi]\!]_n(S) = [\![\mathcal{A}]\!]_n(S)$ for every S and n.*

On the other hand, one can show that the chain-CEA from Example 6 cannot be defined by any CEL+PART-BY formula. This, together with Proposition 7, shows that that CEL+PART-BY is strictly included in the queries defined by chain-CEA.

Like in [17] for CEA, here the determinization of chain-CEA is a crucial property for having efficient streaming evaluation and necessary property for removing duplicate runs that produce the same output. We start by defining our notion of deterministic chain-CEA. Similarly to [17], a deterministic chain-CEA must be "deterministic" with respect to the input and output, namely, given a stream S and a complex event C, there exists at most one run over S that produces C. Formally, we say that a chain-CEA $\mathcal{A} = (Q, \Delta, I, F)$ is I/O deterministic (or just deterministic) if $|I| = 1$ and, for every pair of transitions $(p, P_1, B_1, q_1) \neq (p, P_2, B_2, q_2)$, it holds that $(P_1 \cap B_1[t]) \cap (P_2 \cap B_2[t]) = \emptyset$ for every tuple t, where $B_i[t]$ is the set of all t' such that $(t, t') \in B_i$. In other words, the conditions (P_1, B_1) and (P_2, B_2) must be disjoint. One can easily check that the chain-CEA from Example 6 is deterministic.

▶ **Theorem 8.** *Chain-CEA admit determinization, namely, for any chain-CEA $\mathcal{A}$ there exists a deterministic chain-CEA $\mathcal{A}'$ such that $[\![\mathcal{A}]\!]_n(S) = [\![\mathcal{A}']\!]_n(S)$ for every S and n.*

A natural question that arises from the definition of chain-CEA is whether disequalities are strictly necessary in an automata model for CEP. For example, one can easily see that disequalities are not necessary for defining CEL+PART-BY formulas, since the partition-by operator only requires to check that the same value is used through a contiguous subsequence of the output. In the next result, we show that disequalities are indeed necessary if we want to find an automata model that admits determinization. More precisely, let chain-CEA$^=$ be the class of chain-CEA where all transitions are restricted to equalities.

▶ **Proposition 9.** *There exists a chain-CEA$^=$ $\mathcal{A}$ such that there exists no I/O deterministic chain-CEA$^=$ equivalent to $\mathcal{A}$.*

We are ready to state the main result of the paper.

▶ **Theorem 10.** *For every chain-CEA, there exists a streaming evaluation algorithm with constant update time and constant delay enumeration.*

By combining Proposition 7 and Theorem 10, we get that for any formula in CEL+PART-BY there exists a streaming evaluation algorithm with constant update time per tuple and constant delay enumeration. It is important to stress that chain-CEA is more general than CEL+PART-BY, in particular, the chain-CEA in Figure 2 cannot be defined by a CEL+PART-BY formula, but it can still be evaluated efficiently. We leave open whether there exists a set of predicates $\mathcal{P}$ (like PART-BY) such that CEL+$\mathcal{P}$ characterizes what is definable by chain-CEA.

5 A streaming evaluation algorithm for chain-CEA

In this section we show how to evaluate a chain-CEA over a stream with constant update time and constant-delay enumeration. We explain first the main data structures used by the algorithm to later show how to evaluate a chain-CEA.

The run DAG. In our algorithms, we compactly represent sets of runs by using a directed acyclic graph (DAG) annotated with configurations. Formaly, let $\mathcal{A} = (Q, \Delta, q_0, F)$ be a deterministic chain-CEA. A run DAG G of $\mathcal{A}$ (or just run DAG) is a tuple $G = (V, E, \bot, \kappa)$ consisting of a finite set of vertices V, a set of edges $E \subseteq V \times V$, a special vertex $\bot \in V$, and a function κ that maps every $v \in V$ to a configuration $\kappa(v) \in Q \times \mathbb{N}$ of $\mathcal{A}$. It is required that the graph (V, E) is acyclic, $\kappa(\bot) = (q_0, 0)$, and for every $v \in V$ there is a directed path from v to $\bot$. Furthermore, it is also required that for every $(u, v) \in E$ with $\kappa(u) = (q_1, i_1)$ and $\kappa(v) = (q_2, i_2)$, it holds that $i_1 > i_2$.

Intuitively, a vertex v labeled by $\kappa(v) = (q, i)$ is encoding the last configuration of a run over a stream S. Moreover, by the last two conditions every path starting in v and ending in $\bot$ is representing a run where configurations are listed in decreasing order. We make this intuition more precise as follows. Let $\pi = v_n, \ldots, v_1, \bot$ be a path from $v = v_n$ to $\bot$ in G and $\kappa(v_j) = (q_j, i_j)$ for $j \leq n$. Then $\kappa(\bot), \kappa(v_1), \ldots, \kappa(v_n)$ represents a run of $\mathcal{A}$ and $\mathrm{CE}(\pi) = \{i_1, \ldots, i_n\}$ the complex event defined by π. We denote by $\mathrm{CE}(v)$ the set of all complex events defined by paths from v to $\bot$ in G, and $\mathrm{CE}(U) = \bigcup_{v \in U} \mathrm{CE}(v)$ for $U \subseteq V$.

Note that there could be two paths starting from v in G that define the same complex event in $\mathrm{CE}(v)$. We say that a run DAG G is *safe* if $\mathrm{CE}(v_1) \cap \mathrm{CE}(v_2) = \emptyset$ for every $v_1, v_2 \in V$. Indeed, the safety property allows to enumerate all complex events in G without repetitions.

▶ **Lemma 11.** *Let $G = (V, E, \bot, \kappa)$ be a safe run DAG such that there is a procedure that, given any vertex $v \in V$, enumerates its neighborhood $\{u \mid (v, u) \in E\}$ with constant-delay. Then there exists a procedure that, given $U \subseteq V$, it enumerates $\mathrm{CE}(U)$ with constant delay.*

Therefore, by the previous lemma we can use a safe run DAG to encode the outputs of our evaluation algorithm for chain-CEA and enumerate these outputs with constant delay.

An index for binary predicates. In our evaluation algorithm we will need a special index over vertices of a run DAG to efficiently evaluate the binary predicates of a chain-CEA. Given a new event t and a state p, we want to quickly retrieve all configurations (p, i) that have reached p and such that $(t_i, t) \in B$ for some $e = (p, P, B, q) \in \Delta$. The run DAG will encode configurations (p, i), but we will need an index to store t_i and quickly "check" $(t_i, t) \in B$.

To define this index, we first need to introduce some notation. Let $B = \bigwedge_{i=1}^{n} (a_i \sim_i b_i)$ be a binary predicate with $\sim_i \in \{=, \neq\}$. Without loss of generality, we assume that all conditions $a_i \sim_i b_i$ in B are different. Let $\{(a_i, b_i)\}_i$ be a set of fresh attribute names not used in the schema $\mathcal{R}$. Given a tuple t, we define the left projection and right projection of t with respect to B as the tuples $\overleftarrow{\pi}_B(t)$ and $\vec{\pi}_B(t)$, respectively, with attributes in $\{(a_i, b_i)\}_i$ such that $\overleftarrow{\pi}_B(t).(a_i, b_i) = t.a_i$ whenever $a_i \in \mathrm{att}(t)$ and $\vec{\pi}_B(t).(a_i, b_i) = t.b_i$ whenever $b_i \in \mathrm{att}(t)$. Otherwise, if $a_i \notin \mathrm{att}(t)$ or $b_i \notin \mathrm{att}(t)$, then $\overleftarrow{\pi}_B(t).(a_i, b_i)$ and $\vec{\pi}_B(t).(a_i, b_i)$ are not defined, respectively. The left and right projections extract the relevant information of a tuple t to define $B[t]$. To see this, we say that t_1 and t_2 are *totally different*, denoted by $t_1 \not\equiv t_2$, if and only if $t_1.a \neq t_2.a$ for every $a \in \mathrm{att}(t_1) \cap \mathrm{att}(t_2)$, that is, they are different point-wise.

▶ **Lemma 12.** *Let $B = B_= \wedge B_\neq$ be a binary predicate. Then $(t, t') \in B$ if, and only if, $\overleftarrow{\pi}_{B_=}(t) = \vec{\pi}_{B_=}(t')$ and $\overleftarrow{\pi}_{B_\neq}(t) \not\equiv \vec{\pi}_{B_\neq}(t')$.*

With the previous notation, we are ready to define our index of a transition, called the *equality-disequality index* or ED-index for short. Let $G = (V, E, \bot, \kappa)$ be a run DAG and let $e = (p, P, B_= \wedge B_\neq, q)$ be a transition. We define the ED-index Index_e as a set of triples $(v, t_=, t_\neq)$ where $v \in V$ and $t_=$, $t_\neq$ are left projections with respect to $B_=$ and $B_\neq$, respectively. Intuitively, Index_e will keep all configurations that are at state p and are

Algorithm 1 Evaluation of a det. chain-CEA $\mathcal{A} = (Q, \Delta, q_0, F)$ and a stream $S = t_1 t_2 \ldots$.

```
1:  procedure EVALUATION(A, S)
2:      INIT()
3:      for i := 1 to ∞ do
4:          FIRETRANSITIONS(i)
5:          UPDATEINDICES(i)
6:          ENUMERATE(∪_{q∈F} U_q^i)
7:  procedure INIT( )
8:      G ← NewMappingGraph(q_0)
9:      U_{q_0}^0 ← {⊥}
10:     for all e_0 = (q_0, P, ∅, q) ∈ Δ do
11:         Index_{e_0}^0 ← {(⊥, t_∅, t_∅)}
12: procedure FIRETRANSITIONS(i)
13:     for all e = (p, P, B_= ∧ B_≠, q) ∈ Δ do
14:         (t_=, t_≠) ← (π⃗_{B_=}(t_i), π⃗_{B_≠}(t_i))
15:         if t_i ∈ P ∧ Index_e^{i-1}[t_=, t_≠] ≠ ∅ then
16:             v ← AddNewVertex(G, q, i)
17:             Connect(G, v, Index_e^{i-1}[t_=, t_≠])
18:             U_q^i ← U_q^i ∪ {v}
19: procedure UPDATEINDICES(i)
20:     for all e = (p, P, B_= ∧ B_≠, q) ∈ Δ do
21:         Index_e^i ← Index_e^{i-1}
22:         (t_=, t_≠) ← (←π_{B_=}(t_i), ←π_{B_≠}(t_i))
23:         for all v ∈ U_p^i do
24:             Index_e^i ← Index_e^i ∪ {(v, t_=, t_≠)}
```

"waiting" to trigger e. More specifically, given a stream $S = t_1 t_2 \ldots$ if $(v, t_=, t_\neq) \in \text{Index}_e$ then $\kappa(v) = (p, i)$ and $t_= = \overleftarrow{\pi}_{B_=}(t_i)$ and $t_\neq = \overleftarrow{\pi}_{B_\neq}(t_i)$. Thanks to Lemma 12, whenever we want to check if $(t_i, t) \in B_= \wedge B_\neq$ for a new tuple t, we only need to obtain the tuple $(v, t_=, t_\neq)$ from Index_e and check whether $t_= = \vec{\pi}_{B_=}(t)$ and $t_\neq \not\equiv \vec{\pi}_{B_\neq}(t)$. This motivates the following main query of an ED-index: given a pair of tuples $t'_=$ and $t'_\neq$:

$$\text{Index}_e[t'_=, t'_\neq] = \{v \in V \mid (v, t_=, t_\neq) \in \text{Index}_e \wedge t_= = t'_= \wedge t_\neq \not\equiv t'_\neq\} \tag{1}$$

That is, $\text{Index}_e[t'_=, t'_\neq]$ returns all vertices v representing configurations $\kappa(v) = (p, i)$ such that there is a tuple t' with $t'_= = \vec{\pi}_{B_=}(t')$ and $t'_\neq = \vec{\pi}_{B_\neq}(t')$ and $(t_i, t') \in B_= \wedge B_\neq$. We will use the ED-index to store configurations and to quickly return them when e is fired.

The streaming evaluation algorithm. In Algorithm 1 we show how to evaluate a deterministic chain-CEA over a stream. The main procedure is EVALUATION that receives as input a deterministic chain-CEA $\mathcal{A} = (Q, \Delta, q_0, F)$ and a stream $S = t_1 t_2 \ldots$. This procedure is composed of four subprocedures: INIT for initializing the main data structures, FIRETRANSITIONS(i) for firing the transitions in Δ given a new tuple t_i, UPDATEINDICES(i) for updating each Index_e given the previous tuple t_i, and, finally, ENUMERATE for enumerating all complex events ending at position i. For the sake of presentation, instead of having a yield function that provides each next tuple in the stream, we explicitly index each new phase by i (i.e. associated to tuple t_i) and iterate from 1 to "infinity" (the main for-loop at line 3). Then, given the next tuple t_i, in each i-phase we fire the transitions and update the indices with t_i, and enumerate all complex events at position i. In the sequel, we will first explain the data structures used by the algorithm to later describe each subprocedure.

Algorithm 1 maintains three structures that are used by all subprocedures: the run DAG $G = (V, E, \perp, \kappa)$, the ED-indices Index_e for each $e \in \Delta$, and set of vertices $U_q \subseteq V$ for each $q \in Q$. As it was explained before, G will encode runs of $\mathcal{A}$ and Index_e will allow us to quickly evaluate the binary predicate at e. Moreover, for each $q \in Q$ the set U_q will keep the new vertices v (i.e. configurations) at q. These sets will be useful for updating the indices and enumerating all new results. For the sake of presentation, we assume that G, Index_e, and U_q are defined globally and accessible by all subprocedures.

In each i-phase, the algorithm will update G to represent all runs of $\mathcal{A}$ over S until position i. To that end, it will use the following methods on run DAGs. The first method, `NewMappingGraph`(q_0), creates a new event DAG G containing only the vertex $\perp$ with

$\kappa(\bot) = (q_0, 0)$ and empty sets of vertices V and edges E. The second method, `AddNewVertex`, receives an event DAG G and a configuration (q, i), and creates a fresh vertex v with $\kappa(v) = (q, i)$, and adds it to V. Finally, the method returns the vertex v. The last method, `Connect`, receives as input a run DAG G, a vertex v on G, and a nonempty set of vertices $U \subseteq V$, and connects v with each vertex in U, namely, (v, u) is added to E for every $u \in U$. Although `AddNewVertex` and `Connect` could temporary break the properties of G (e.g. acyclicity), we will use it one after the other and it will be clear that the properties of G are always preserved.

For the structures Index_e and U_q, the reader might have noticed that in Algorithm 1 we use a superscript Index_e^i and U_q^i. This i is denoting the "version" of Index_e and U_q at phase i. We assume that each new i-version is always initialized as empty (i.e. $U_q^i = \emptyset$ and $\text{Index}_e^i = \emptyset$). It is important to note that for U_q^i we use the index i just to simplify the presentation (i.e. we could have reuse a set U_q in each phase). However, for Index_e^i the superscript is crucial to denote the version of Index_e when, for example, a vertex v is connected with the set $\text{Index}_e^i[t_=, t_\neq]$ (see line 17). As it will be discussed later (see Section 6), Index_e is a (partially) persistent data structure [14] and the superscript is denoting the i-version of the structure.

We are ready to describe each subprocedure in Algorithm 1. The algorithm starts with Init that is in charge of initializing G, Index_e^0, and U_q^0 before phase 1. For this, a new event DAG G is created and the vertex with the initial configuration $\bot$ is assigned to $U_{q_0}^0$ (recall that $U_q^i = \emptyset$ for $i \geq 0$ by assumption). Intuitively, this represents that the initial configuration is ready to start. For initializing Index_e, we assume without loss of generality that all outgoing transitions from q_0 use trivial predicates, namely, $B = \emptyset$ for every $e_0 = (q_0, P, B, q) \in \Delta$. Then $(\bot, t_\emptyset, t_\emptyset)$ is the only triple that must contain $\text{Index}_{e_0}^0$ with $t_\emptyset$ the empty tuple.

For each new phase i, we call FireTransitions(i) that check for each transition $e = (p, P, B_= \wedge B_\neq, q)$ whether it can be fired or not given the new tuple t_i (line 13). For this, we extract from t_i its right-projections $t_=$ and $t_\neq$ with respect to $B_=$ and $B_\neq$, respectively. Then we check if t_i satisfies P and whether there exists a previous configuration (p, j) such that (t_j, t_i) satisfies $B_= \wedge B_\neq$. We do this through Index_e, $t_=$, and $t_\neq$ by checking if $\text{Index}_e^{i-1}[t_=, t_\neq] \neq \emptyset$. If this is the case, all pairs of configurations (p, j) and (q, i) with $(p, j) \in \text{Index}_e^{i-1}[t_=, t_\neq]$ satisfy e and we must extend G with a new configuration (q, i) that represents all these new runs. For this, we create a new node v in G for configuration (q, i) and connect v with each vertex in $\text{Index}_e^{i-1}[t_=, t_\neq]$ (lines 16-17). Finally, the new vertex v is added to the set U_q^i of new vertices in state q at phase i.

The next step in phase i is to update Index_e^{i-1} to its new version Index_e^i given t_i. For this, we use the set U_p^i to update each transition $e = (p, P, B_= \wedge B_\neq, q)$. More specifically, in UpdateIndices(i) we iterate over each transition $e = (p, P, B_= \wedge B_\neq, q)$ and make Index_e^i equal to its previous version. Then, we extract from t_i its left-projections $t_=$ and $t_\neq$ with respect to $B_=$ and $B_\neq$, respectively, and add $(v, t_=, t_\neq)$ to Index_e^i for each $v \in U_p^i$. Recall that U_p^i contains all the new vertices added during FireTransitions(i) and, in particular, $\kappa(v) = (p, i)$ for each $v \in U_p^i$. After UpdateIndices(i) is done, the ED-index Index_e^i contains all the relevant information for checking $B_= \wedge B_\neq$ in the next phases.

Up to this point, it is straightforward to prove the following invariant after each phase i, which leads to the correctness proof of Algorithm 1.

▶ **Lemma 13.** *Consider $\{U_q^i\}_{q \in Q}$ and G after the end of the i-phase. Then, for every run $(q_0, 0), (q_1, i_1) \ldots, (q_n, i_n)$ of $\mathcal{A}$ over S with $i_n = i$, there exist $v \in U_{q_n}^i$ and a path $v_n, \ldots, v_0$ in G with $v_n = v$ and $v_0 = \bot$ such that $\kappa(v_j) = (q_j, i_j)$ for every $j \leq n$. Conversely, for every $v \in U_q^i$ and every path $v_n, \ldots, v_0$ in G with $v_n = v$ and $v_0 = \bot$, it holds that $\kappa(v_0), \ldots, \kappa(v_n)$ is a run of $\mathcal{A}$ over S. Moreover, if $\mathcal{A}$ is deterministic, then G is safe.*

The final step at phase i is to enumerate all complex events of accepting runs. For this, we call the subprocedure ENUMERATE over the set of vertices $\cup_{q \in F} U_q^i$. By Lemma 13, we know that G correctly encodes all runs of $\mathcal{A}$ until the i-th tuple of S and, moreover, G is safe (i.e each complex event is represented by exactly one path in G). Therefore, we can easily enumerate all complex events $[\![\mathcal{A}]\!]_i(S)$ one-by-one and without repetitions, by enumerating all paths in G starting at vertices in $\cup_{q \in F} U_q^i$ and ending at $\bot$.

It is only left to show that Algorithm 1 satisfies constant update time and constant-delay enumeration. To do this, we have to dig deeper into the implementation of Index_e, which is the goal of the last section.

6 A persistent index structure for equalities and disequalities

Fix a transition $e = (p, P, B_= \wedge B_{\neq}, q)$. Let $(v_0, t_0, r_0), (v_1, t_1, r_1), \ldots$ be a sequence of triples such that v_i is a vertex and t_i, r_i are tuples for all $i \in \mathbb{N}$. Furthermore, define $\text{Index}_e^0 = \emptyset$ and $\text{Index}_e^i = \text{Index}_e^{i-1} \cup \{(v_i, t_i, r_i)\}$. Call (v_i, t_i, r_i) an insertion and i the version of Index_e.

To have constant update time and constant-delay enumeration, Index_e must satisfy the following properties, for every pair of tuples t, r and point in time $i \in \mathbb{N}$:

1. every new insertion in Index_e takes constant time, and
2. for all $j \leq i$, $\text{Index}_e^j[t, r]$ can be can be enumerated with constant-delay.

The last condition implies that Index_e is a persistent data structure [14], namely, it preserves the previous version (i.e. Index_e^j) of itself whenever it is modified.

We claim that, if Index_e satisfies the above three properties, then Algorithm 1 runs with constant update time and constant-delay enumeration. First, given that $\mathcal{A}$ is fixed, then it is clear that every step of Algorithm 1 can be done in constant time, except lines 15, 17, and 24. Checking whether $\text{Index}_e[t, r] \neq \emptyset$ (line 15) or doing an insertion in Index_e (line 24) can be done in constant time by properties (2) and (1), respectively. Furthermore, one can execute $\texttt{Connect}(G, v, \text{Index}_e^i[t, r])$ (line 17) in constant time if, instead of coding the graph G with adjacency lists, we represent the neighborhood of each vertex v by storing t, r, and i in v and, because of (2), we can later call $\text{Index}_e^i[t, r]$ whenever needed. Finally, from Lemma 11 we know that, if the neighborhood of each vertex from a safe run DAG can be enumerated with constant delay, then $\text{CE}(U)$ can also be enumerated with constant delay. Given that $\text{Index}_e^i[t, r]$ allows to enumerate the neighborhood of each vertex, then the enumeration with constant-delay follows.

In the sequel, we show how to implement Index_e in order to satisfy properties (1) and (2).

Case without disequalities. If e does not have disequalities (i.e. $B_{\neq}$ is trivial), then for every $(v, t, r) \in \text{Index}_e$, we can drop r and keep only (v, t). To satisfy (1) and (2) we use a key-value index $\texttt{DS}$ where keys are tuples t and each value $\texttt{DS}[t]$ is a list of pairs $(u_0, i_0), \ldots, (u_n, i_n)$ where each u_k is a vertex and i_k is a "timestamp", namely, the phase when u_k was inserted. Then, for every new insertion (u_i, t_i) in phase i, we go to $\texttt{DS}[t_i]$ and insert (u_i, i) at the end of the list. Finally, for every query of the form $\text{Index}_e^j[t]$ we can go into $\texttt{DS}[t]$, jump into the pair (u_k, i_k) with $i_k = j$ and enumerate $(u_k, i_k), \ldots, (u_0, i_0)$ with constant-delay. Recall that by our RAM model of computation, we can find the list $\texttt{DS}[t]$ and find the pair (u_k, i_k) inside $\texttt{DS}[t]$ in constant time (in the latter case, we need another key-value index for $\texttt{DS}[t]$ that, given j, finds $(\texttt{DS}[t])[j] = (u_k, j)$). Furthermore, by keeping $\texttt{DS}[t]$ as a linked list, one can easily enumerate $(u_k, i_k), \ldots, (u_0, i_0)$ with constant-delay.

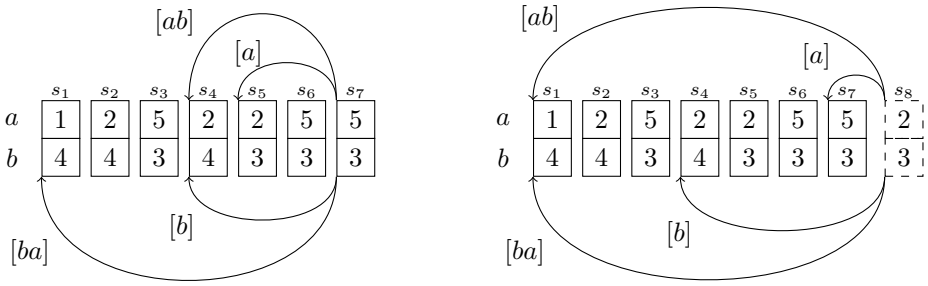

Figure 3 A list of tuples $s_1 \dots s_7$ with the additional bookkeeping to support disequalities.

Case with disequalities. If e includes disequalities (i.e. $B_{\neq}$ is non-trivial), then we need to extend our lists $\texttt{DS}[t]$ to support insertions (v_i, t_i, s_i) and queries $\text{Index}_e^i[t, r]$. For this, extend $\texttt{DS}[t]$ as a list of triples $(u_0, s_0, i_0), \dots, (u_n, s_n, i_n)$ where u_k and i_k are as before, and s_k is the tuple for supporting disequalities. Similar to the case without disequalities, for every new insertion (v_i, t_i, s_i) at phase i we go into the list $\texttt{DS}[t_i]$ and insert the triple (v_i, s_i, i) at the end of the list. Then for every query $\text{Index}_e^i[t, r]$ we can jump into the list $\texttt{DS}[t]$, jump into the triple (u_k, s_k, i_k) with $i_k = i$ and enumerate all u_l with $l \leq k$ such that $s_l \not\equiv r$ (i.e. s_l and r are totally different). Of course, this last enumeration step cannot be done with constant delay, unless some extra bookkeeping is added to the data structure. The rest of this section is then devoted to do this.

For the sake of simplification, from now on assume that each list $\texttt{DS}[t]$ is composed only by tuples $s_1, \dots, s_n$. Then the problem is reduced to, given a tuple r and position i, enumerate the set $\{s_k \mid k \leq i \wedge s_k \not\equiv r\}$. Without loss of generality, assume also that all $s_1, \dots, s_n$ have the same set of attributes A, i.e. $\text{att}(s_k) = A$, and define $d = |A|$. If not, complete each tuple s_k with the missing attributes and a fresh value for each new attribute. For example, at the left of Figure 3 we give a list $s_1, \dots, s_7$ with attributes $A = \{a, b\}$ and $d = 2$ where each column is a tuple (over integers) and each row is an attribute.

Let $\bar{a} = a_1 a_2 \dots a_m$ be a sequence of non-repeating attributes of A, and define $\bar{A}$ to be the set of all $\bar{a}$. For each tuple s_k and each $\bar{a}$, we define a tuple $s_k[\bar{a}] = s_j$ with $j < k$. Strictly speaking, $s_k[\bar{a}]$ will be a (backward) pointer from s_k to s_j that allows us to jump to $s_k[\bar{a}]$ in constant time. Given that our analysis is in data complexity, $|\bar{A}|$ is of constant size, so we only store a constant number of pointers in each tuple s_k (although exponential in d). In Figure 3, the pointers $[a]$, $[b]$, $[ab]$, and $[ba]$ of s_7 are displayed with arrows.

Now, for each s_k in the list $\texttt{DS}[t] = s_1, \dots, s_n$, the tuple $s_k[\bar{a}]$ is defined recursively as follows. First, for every attribute $a \in A$, $s_k[a]$ points to the maximum $j < k$ such that $s_k.a \neq s_j.a$. Next, for each sequence $\bar{a} = a_1 a_2 \dots a_m$, $s_k[\bar{a}]$ points to the maximum $j < k$ such that, for all $1 \leq l \leq m$, $s_j.a_l \neq s_k[a_1 \dots a_{l-1}].a_l$ where $s_k[\epsilon] = s_k$ (ϵ is the empty sequence in $\bar{A}$). In the case that there is no such tuple s_j, then $s_k[\bar{a}]$ is not defined, which means we reached the beginning of $\texttt{DS}[t]$.

▶ **Example 14.** Consider the list $s_1, \dots, s_7$ at the left of Figure 3 and consider tuple s_7. Then $s_7[a] = s_5$ is the last tuple before s_7 with a value different than 5, and $s_7[ab] = s_4$ is the last before s_7 with $s_4.a = 2 \neq 5 = s_7.a$ and $s_4.b = 4 \neq 3 = s_5.b$. Similarly, $s_7[b] = s_4$ is the last node before s_7 with $s_4.b = 4 \neq 3 = s_7.b$, and $s_7[ab] = s_1$ is the last before s_7 with $s_1.b = 4 \neq 3 = s_7.b$ and $s_1.a = 1 \neq 2 = s_4.a$.

With the previous structure over $s_1, \ldots, s_n$, we show how to enumerate with constant delay the set $\{s_k \mid k \leq i \wedge s_k \not\equiv r\}$ given a tuple r and index i. For this, we define a procedure $\texttt{findNext}(s_k, r)$ that returns the last tuple s_j with $j < k$ such that $s_j \not\equiv r$ (and false if s_j does not exist). Note that, if `findNext` runs in constant time, then we can enumerate the set $\{s_k \mid k \leq i \wedge s_k \not\equiv r\}$ with constant delay: first, if $s_i \not\equiv r$ then we enumerate s_i; then for every last node s_k we enumerated, we call $\texttt{findNext}(s_k, r)$ to get the next one, until `findNext` returns false. For computing $\texttt{findNext}(s_k, r)$, let $s := s_{k-1}$ be the node immediately before s_k in $\texttt{DS}[t]$. In the first step we check if $s[\epsilon]$ fulfills the condition, namely, if $s \not\equiv r$. If so, we return $s[\epsilon]$; otherwise, there must be some attribute a_1 such that $s[\epsilon].a_1 = r.a_1$. In the next step we consider $s[a_1]$ and check if $s[a_1].a \neq r.a$ for each $a \in \text{att}(R) \setminus \{a_1\}$; if so, we return $s[a_1]$. Notice we do not need to compare r with all tuples between $s[a_1]$ and $s[\epsilon]$ because, by definition, each tuple s' between both satisfy $s'.a_1 = s[\epsilon].a_1 = r.a_1$. Furthermore, we no longer need to check the value of a_1 in $s[a_1]$ because $s[a_1].a_1 \neq s[\epsilon].a_1 = r.a_1$. We repeat this procedure inductively. If we are in step $1 \leq m < d$ and failed in all previous steps, then for $\bar{a} = a_1 \ldots a_m \in \bar{A}$, assume $s[a_1 \ldots a_{l-1}].a_l = r.a_l$ for every $l \leq m$. If $s[\bar{a}] \not\equiv r$, return $s[\bar{a}]$; otherwise consider some attribute $a_{m+1} \in A \setminus \{a_1, \ldots, a_m\}$ such that $s[\bar{a}].a_{m+1} = r.a_{m+1}$. Then we consider $s[\bar{a} \cdot a_{m+1}]$ in the next step. Again, we do not need to compare r with all elements between $s[\bar{a} \cdot a_{m+1}]$ and $s[\bar{a}]$: each tuple s' between both satisfies $s'.a_{m+1} = s[\bar{a}].a_{m+1} = r.a_{m+1}$. Also we do not need to compare $s[\bar{a} \cdot a_{m+1}]$ with r on $\{a_1, \ldots, a_{m+1}\}$ given that, by induction, $s[\bar{a} \cdot a_{m+1}].a_{m+1} \neq s[\bar{a}].a_{m+1} = r.a_{m+1}$ and $s[\bar{a} \cdot a_{m+1}].a_l \neq s[a_1 \ldots a_{l-1}].a_l = r.a_l$. At some point we will find some tuple that fulfills the conditions; in the worst-case scenario we iterate d times, in which case we are sure by definition that $s[a_1 \ldots a_d]$ satisfies the condition or is undefined (i.e. it does not exists). All in all, the procedure takes $O(d)$ steps, which is constant. Moreover, this procedure does not use the pointers of s_k, but the ones of s_{k-1}. This is an important property that we use next when we want to insert a new node in $\texttt{DS}[t]$.

It is left only to show how to update $\texttt{DS}[t] = s_1, \ldots, s_n$ when we read a new tuple s_{n+1}. For this, we add s_{n+1} to the end of the list and define $s_{n+1}[\bar{a}]$ for each $\bar{a} \in \bar{A}$ in the following way. If the list is empty, then $s_{n+1}[\bar{a}]$ is undefined for all $\bar{a} \in \bar{A}$. Otherwise, for each $\bar{a} = a_1 \ldots a_m$ we define $s_{n+1}[\bar{a}]$ incrementally over the length m. Suppose that, $s_{n+1}[a_1 \ldots a_l]$ is already defined for every $l < m$. Define the tuple r such that $r.a_l = s_{n+1}[a_1 \ldots a_{l-1}].a_l$ for all $l < m$. Then, define $s_{n+1}[a_1 \ldots a_m] := \texttt{findNext}(s_{n+1}, r)$. In other words, we collect all values $c_1 = s_{n+1}[\epsilon].a_1$, $c_2 = s_{n+1}[a_1].a_2, \ldots, c_m = s_{n+1}[a_1 \ldots a_m].a_m$ and find the last tuple s such that $s.a_l \neq c_l$ for every $l \leq m$. As it was mentioned above, since `findNext` only uses the pointers of s_n, and not of s_{n+1} itself, the function is well-defined. Moreover, given that $\texttt{findNext}(s_{n+1}, r)$ can be found in constant time, then $s_{n+1}[a_1 \ldots a_m]$ is computed in constant time as well.

▶ **Example 15.** Suppose that we want to add the node $s_8 = \{a \to 2, b \to 3\}$ to the list on the left of Figure 3. The result is shown on the right of Figure 3 where s_8 is the last dashed column. We define $s_8[\bar{a}]$ incrementally using `findNext`. For a, we call $\texttt{findNext}(\mathtt{n}_8, \{a \to 2\})$, which tries with the last tuple s_7 and, because $s_7.a \neq 2$, we set $s_8[a] := s_7$. For b, we call $\texttt{findNext}(s_8, \{b \to 3\})$, which first tries with s_7, but $s_7.b = 3$, so it tries with $s_7[b] = s_4$; since $s_4.b \neq s_7.b$, we set $s_8[b] = s_4$. For sequence ab, we have $s_8.a = 2$ and $s_8[a].b = 3$, so we call $\texttt{findNext}(s_8, \{a \to 2, b \to 3\})$. As s_7 conflicts in b, it tries with $s_7[b] = s_4$, but this time it conflicts with a, so it tries with $s_7[ba] = s_1$. As $s_1.a \neq 2$ and $s_1.b \neq 3$, we set $s_8[ab] = s_1$. The same procedure is done for ba, resulting in $s_8[ba] = s_1$.

By combining the key-value index `DS` where the keys are tuples and the values are the extended list with the additional bookkeeping mentioned above, we get properties (1) and(2) needed for Algorithm 1 to have constant update time and constant-delay enumeration.

7 Future work

This work rises several research opportunities regarding streaming evaluation of queries with correlation in CEP. The first problem is to find a unified class of queries that includes chain-CEA and hierarchical queries. Indeed, there are simple hierarchical queries (e.g. $R(x) \wedge S(y) \wedge T(x)$) that are not definable by chain-CEA. Another relevant question is whether partition-by queries with projection can be evaluated efficiently. Chain-CEA forbid the use of projection and it is not clear how to extend Algorithm 1 to support it. In particular, it is not clear how to extend this algorithm to support selection strategies [17], an important operator in CEP to filter the number of outputs. Finally, this work studies the streaming evaluation of equality and disequality predicates in CEP, but leaves open the evaluation of other predicates for correlation, like inequalities.

References

1 S. Abiteboul, R. Hull, and V. Vianu. *Foundations of databases: the logical level*. Addison-Wesley, 1995.

2 Yanif Ahmad, Oliver Kennedy, Christoph Koch, and Milos Nikolic. Dbtoaster: Higher-order delta processing for dynamic, frequently fresh views. *Proceedings of the VLDB Endowment*, 5(10):968–979, 2012.

3 A. Aho and J. Hopcroft. *The design and analysis of computer algorithms*. Pearson Education India, 1974.

4 E. Alevizos, A. Artikis, and G. Paliouras. Symbolic Automata with Memory: a Computational Model for CEP. *arXiv preprint arXiv:1804.09999*, 2018.

5 A. Arasu, S. Babu, and J. Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *The VLDB Journal*, 2006.

6 Guillaume Bagan. MSO queries on tree decomposable structures are computable with linear delay. In *International Workshop on Computer Science Logic*, pages 167–181. Springer, 2006.

7 C. Berkholz, J. Keppeler, and N. Schweikardt. Answering conjunctive queries under updates. In *PODS*, pages 303–318, 2017.

8 Stefano Ceri and Jennifer Widom. Deriving Production Rules for Incremental View Maintenance. In *VLDB*, 1991.

9 Rada Chirkova, Jun Yang, et al. Materialized views. *Foundations and Trends® in Databases*, 4(4):295–405, 2012.

10 T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to algorithms*. MIT press, 2009.

11 Bruno Courcelle. Linear delay enumeration and monadic second-order logic. *Discrete Applied Mathematics*, 157(12):2675–2700, 2009.

12 G. Cugola and A. Margara. Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys*, 2012.

13 Nilesh N. Dalvi and Dan Suciu. The dichotomy of conjunctive queries on probabilistic structures. In *Proceedings of the Twenty-Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 11-13, 2007, Beijing, China*, pages 293–302, 2007.

14 James R Driscoll, Neil Sarnak, Daniel D Sleator, and Robert E Tarjan. Making data structures persistent. *Journal of computer and system sciences*, 38(1):86–124, 1989.

15 Esper Enterprise Edition website. `http://www.espertech.com/`. Accessed: 2018-12-21.

16 Opher Etzion, Peter Niblett, and David C Luckham. *Event processing in action*. Manning Greenwich, 2011.

17 A. Grez, C. Riveros, and M. Ugarte. A formal framework for Complex Event Processing. In *ICDT*, 2019.

18 A. Grez, C. Riveros, M. Ugarte, and S. Vansummeren. A Second-Order Approach to Complex Event Recognition. *arXiv preprint arXiv:1712.01063*, 2017.

19 M. Groover. *Automation, production systems, and computer-integrated manufacturing*. Prentice Hall, 2007.

20 Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In *Proceedings of the forty-seventh annual ACM symposium on Theory of computing*, pages 21–30. ACM, 2015.

21 Martin Hirzel, Guillaume Baudart, Angela Bonifati, Emanuele Della Valle, Sherif Sakr, and Akrivi Akrivi Vlachou. Stream processing languages in the big data era. *ACM SIGMOD Record*, 47(2):29–40, 2018.

22 M. Idris, M. Ugarte, and S. Vansummeren. The dynamic Yannakakis algorithm: Compact and efficient query processing under updates. In *SIGMOD*, 2017.

23 M. Idris, M. Ugarte, S. Vansummeren, H. Voigt, and W. Lehner. Conjunctive queries with inequalities under updates. *VLDB*, 11(7):733–745, 2018.

24 M. Kaminski and N. Francez. Finite-memory automata. *Theoretical Computer Science*, 134(2):329–363, 1994.

25 Christoph Koch. Incremental query evaluation in a ring of databases. In *Proceedings of the twenty-ninth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 87–98. ACM, 2010.

26 Paraschos Koutris and Dan Suciu. Parallel evaluation of conjunctive queries. In *Proceedings of the thirtieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 223–234. ACM, 2011.

27 B. Mukherjee, T. Heberlein, and K. Levitt. Network intrusion detection. *IEEE network*, 1994.

28 B. Sahay and J. Ranjan. Real time business intelligence in supply chain analytics. *Information Management & Computer Security*, 2008.

29 L. Segoufin. Automata and logics for words and trees over an infinite alphabet. In *CSL*, 2006.

30 L. Segoufin. Enumerating with constant delay the answers to a query. In *Proceedings of the 16th International Conference on Database Theory*, pages 10–20. ACM, 2013.

31 E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *SIGMOD*, 2006.