RESEARCH-ARTICLE

Complex Event Recognition meets Hierarchical Conjunctive Queries

DANTE PINTO, Pontifical Catholic University of Chile, Santiago, RM, Chile

CRISTIAN RIVEROS, Pontifical Catholic University of Chile, Santiago, RM, Chile

Published: 07 November 2024

Citation in BibTeX format

Proceedings of the ACM on Management of Data, Volume 2, Issue 5 (November 2024)
https://doi.org/10.1145/3695834
EISSN: 2836-6573

Complex Event Recognition meets Hierarchical Conjunctive Queries

DANTE PINTO[*], Pontificia Universidad Católica de Chile, Chile, IMFD, Chile, and UHasselt, Belgium
CRISTIAN RIVEROS, Pontificia Universidad Católica de Chile, Chile and IMFD, Chile

Hierarchical conjunctive queries (HCQ) are a subclass of conjunctive queries (CQ) with robust algorithmic properties. Among others, Berkholz, Keppeler, and Schweikardt have shown that HCQ is the subclass of CQ (without projection) that admits dynamic query evaluation with constant update time and constant delay enumeration. On a different but related setting stands Complex Event Recognition (CER), a prominent technology for evaluating sequence patterns over streams. Since one can interpret a data stream as an unbounded sequence of inserts in dynamic query evaluation, it is natural to ask to which extent CER can take advantage of HCQ to find a robust class of queries that can be evaluated efficiently.

In this paper, we search to combine HCQ with sequence patterns to find a class of CER queries that can get the best of both worlds. To reach this goal, we propose a class of complex event automata model called Parallelized Complex Event Automata (PCEA) for evaluating CER queries with correlation (i.e., joins) over streams. This model allows us to express sequence patterns and compare values among tuples, but it also allows us to express conjunctions by incorporating a novel form of non-determinism that we call parallelization. We show that for every HCQ (under bag semantics), we can construct an equivalent PCEA. Further, we show that HCQ is the biggest class of full CQ that this automata model can define. Then, PCEA stands as a sweet spot that precisely expresses HCQ (i.e., among full CQ) and extends them with sequence patterns. Finally, we show that PCEA also inherits the good algorithmic properties of HCQ by presenting a streaming evaluation algorithm under sliding windows with logarithmic update time and output-linear delay for the class of PCEA with equality predicates.

CCS Concepts: • **Theory of computation → Database theory**; *Automata over infinite objects*; *Automata extensions*; • **Information systems → Data streams**.

Additional Key Words and Phrases: Query evaluation, conjunctive queries, streams, complex event recognition.

ACM Reference Format:
Dante Pinto and Cristian Riveros. 2024. Complex Event Recognition meets Hierarchical Conjunctive Queries. *Proc. ACM Manag. Data* 2, 5 (PODS), Article 216 (November 2024), 26 pages. https://doi.org/10.1145/3695834

1 Introduction

Hierarchical Conjunctive Queries [12] (HCQ) are a subclass of Conjunctive Queries (CQ) with good algorithmic properties for dynamic query evaluation [9, 18]. In this scenario, users want to continuously evaluate a CQ over a database that receives insertion, updates, or deletes of tuples, and to efficiently retrieve the output after each modification. A landmark result by Berkholz, Keppeler, and Schweikardt [5] shows that HCQ are the subfragment among CQ for dynamic query evaluation.

---

[*] This work was done as a member of PUC Chile and IMFD.

Authors' Contact Information: Dante Pinto, Pontificia Universidad Católica de Chile, Santiago, Chile and IMFD, Santiago, Chile and UHasselt, Hasselt, Belgium, dante.pintoaraya@uhasselt.be; Cristian Riveros, Pontificia Universidad Católica de Chile, Santiago, Chile and IMFD, Santiago, Chile, cristian.riveros@uc.cl.

ACM 2836-6573/2024/11-ART216
https://doi.org/10.1145/3695834

Specifically, they show one can evaluate every HCQ with constant update time and constant-delay enumeration. Furthermore, they show that HCQ are the only class of full CQ (i.e., CQ without projection) with such guarantees, namely, under fined-grained complexity assumptions, a full CQ can be evaluated with constant update time and constant delay enumeration if, and only if, the query is hierarchical. Therefore, HCQ stand as the fragment for efficient evaluation under a dynamic scenario (see also [18]).

Data stream processing is another dynamic scenario where we want to evaluate queries continuously but now over an unbounded sequence of tuples (i.e., a data stream). *Complex Event Recognition* (CER) is one such technology for processing information flow [11, 14]. CER systems read high-velocity streams of data, called events, and evaluate expressive patterns for detecting complex events, a subset of relevant events that witness a critical case for a user. A singular aspect of CER compared to other frameworks is that the order of the stream's data matters, reflecting the temporal order of events in reality (see [29]). For this reason, sequencing operators are first citizens on CER query languages, which one combines with other operators, like filtering, disjunction, and correlation (i.e., joins), among others [4].

Similar to dynamic query evaluation, this work aims to find a class of CER query languages with efficient streaming query evaluation. Our strategy to pursue this goal is simple but effective: we use HCQ as a starting point to guide our search for CER query languages with good algorithmic properties. Since one can interpret a data stream as an unbounded sequence of inserts in dynamic query evaluation, we want to extend HCQ with sequencing while maintaining efficient evaluation. We plan this strategy from an algorithmic point of view. Instead of studying which CER query language fragments have such properties, we look for automata models that can express HCQ. By finding such a model, we can later design our CER query language to express these queries [17].

With this goal and strategy in mind, we start from the proposal of Chain Complex Event Automata (CCEA), an automata model for CER expressing sequencing queries with correlation, but that cannot express simple HCQ [16]. We extend this model with a new sort of non-deterministic power that we call *parallelization*. This feature allows us to run several parallel executions that start independently and to gather them together when reading new data items. We define the class of *Parallelized Complex Event Automata* (PCEA), the extension of CCEA with parallelization. As an extension, PCEA can express patterns with sequencing, disjunction, iteration, and correlation but also allows conjunction. In particular, we can show that PCEA can express a full CQ Q if, and only if, Q is hierarchical. Then, PCEA is a sweet spot that precisely expresses HCQ (i.e., among full CQ) and extends them with sequencing and other operations. Moreover, we show that PCEA inherits the good algorithmic properties of HCQ by presenting a streaming evaluation algorithm under sliding windows, reaching our desired goal.

Example 1.1. To get a feeling of the setting and the queries definable by PCEA, consider the following scenario (see Section 2 and 4 for the formal details). Suppose that we have a stream:

$$\underbrace{S(2,11)}_{0}\ \underbrace{T(2)}_{1}\ \underbrace{R(1,10)}_{2}\ \underbrace{S(2,11)}_{3}\ \underbrace{T(3)}_{4}\ \underbrace{R(2,11)}_{5}\ \underbrace{S(1,10)}_{6}\ \underbrace{T(1)}_{7}\ \ldots$$

where $R(a,b)$, $S(a,b)$, or $T(a)$ are data tuples with relation names R, S, and T over some schema, and a, b are data values in $\mathbb{N}$. One may think of this stream as a sequence of *events* $t_0 t_1 \ldots$ (e.g., sensors measures, messages in a social network, insertions in a database, etc), where each event t_i happens before t_{i+1}. Further, suppose that we want to evaluate the CQ:

$$Q_0(x,y) \leftarrow T(x),\ S(x,y),\ R(x,y)$$

namely, find all events (i.e., tuples) t_i, t_j, and t_k such that $t_i = T(a)$, $t_j = S(a, b)$, and $t_k = R(a, b)$ for some data values $a, b \in \mathbb{N}$. For instance, $S(2, 11)$, $T(2)$, and $R(2, 11)$ at positions 0, 1, and 5, respectively, satisfies Q_0, and tuples $R(1, 10)$, $S(1, 10)$, and $T(1)$ at positions 2, 6, and 7 do so as well.

Suppose now that we want to further restrict Q_0 in such a way that, in addition to the previous conditions, $R(x, y)$ must arrive after $T(x)$ and $S(x, y)$. Informally, the restricted query Q_0' could look like[1]:

$$Q_0'(x, y) \leftarrow T(x),\ S(x, y),\ R(x, y),\ T < R, S < R$$

For example, tuples at position 0, 1, and 5 will be in the output of Q_0' and tuples at position 2, 6, and 7 will not (since $T(1)$ is after $R(1, 10)$).

We want to evaluate Q_0' in a streaming fashion, enumerating the outputs as soon as a new tuple arrives. Notice that Q_0' is not a hierarchical CQ (and not even a CQ), so we cannot evaluate it efficiently by using the techniques of dynamic query evaluation. As we will see, we can define Q_0' with a PCEA (see $\mathcal{P}_0$ in Figure 1), and then we can evaluate it with the techniques developed in this work.

Contributions. The technical contributions and outline of the paper are the following.

In Section 2, we provide some basic definitions plus recalling the definition of CCEA.

In Section 3, we introduce the concept of parallelization for standard non-deterministic NFA, called PFA, and study their properties. We show that PFA can be determinized in exponential time (similar to NFA) (Proposition 3.2). We then apply this notion to CER and define the class of PCEA, showing that it is strictly more expressive than CCEA (Proposition 3.4).

Section 4 compares PCEA with HCQ under bag semantics. Given that PCEA runs over streams and HCQ over relational databases, we must revisit the semantics of HCQ and formalize in which sense an HCQ and a PCEA define the same query. We show that under such comparison, every HCQ Q under bag semantics can be expressed by a PCEA with equality predicates of exponential size in $|Q|$ and of quadratic size if Q does not have self-joins (Theorem 4.1). Furthermore, if Q is not hierarchical, then Q cannot be defined by any PCEA (Theorem 4.2).

In Section 5, we study the evaluation of PCEA in a streaming scenario. Specifically, we present a streaming evaluation algorithm under a sliding window with logarithmic update time and output-linear delay for the class of unambiguous PCEA with equality predicates (Theorem 5.1).

Related work. Dynamic query evaluation of HCQ and acyclic CQ has been studied in [5, 18, 20, 30]. This research line did not study HCQ or acyclic CQ in the presence of order predicates. [28, 31] studied CQ under comparisons (i.e., θ-joins) but in a static setting (i.e., no updates). The closest work is [19], which studied dynamic query evaluation of CQ with comparisons; however, this work did not study well-behaved classes of HCQ with comparisons, and, further, their algorithms have update time linear in the data.

Complex event recognition and, more generally, data stream processing have studied the evaluation of joins over streams (see, e.g., [21, 32, 33]). To the best of our knowledge, no work in this research line optimizes queries focused on HCQ or provides guarantees regarding update time or enumeration delay in this setting. We base our work on [16], which we will discuss extensively.

2 Preliminaries

Strings and NFA. A *string* is a sequence of elements $\bar{s} = a_0 \dots a_{n-1}$. For presentation purposes, we make no distinction between a *sequence* or a string and, thus, we also write $\bar{s} = a_0, \dots, a_{n-1}$ for denoting a string. We will denote strings using a bar and its i-th element by $\bar{s}[i] = a_i$. We use $|\bar{s}| = n$

---

[1] We use the notation $<$ in this example to explain Q_0'. Since we will use PCEA to define queries, we will not use this notation later in the paper.

for the length of $\bar{s}$ and $\{\bar{s}\} = \{a_0, \ldots, a_{n-1}\}$ to consider $\bar{s}$ as a set. Given two strings $\bar{s}$ and $\bar{s}'$, we write $\bar{s}\bar{s}'$ for the *concatenation* of $\bar{s}$ followed by $\bar{s}'$. Further, we say that $\bar{s}'$ is a *prefix* of $\bar{s}$, written as $\bar{s'} \leq_p \bar{s}$, if $|\bar{s}'| \leq |\bar{s}|$ and $\bar{s}'[i] = \bar{s}[i]$ for all $i < |\bar{s}'|$. Given a non-empty set Σ we denote by Σ^* the set of all strings from elements in Σ, where $\epsilon \in \Sigma^*$ denotes the 0-length string. For a function $f : \Sigma \to \Omega$ and $\bar{s} \in \Sigma^*$, we write $f(\bar{s}) = f(a_0) \ldots f(a_{n-1})$ to denote the point-wise application of f over $\bar{s}$.

A *Non-deterministic Finite Automaton* (NFA) is a tuple $\mathcal{A} = (Q, \Sigma, \Delta, I, F)$ such that Q is a finite set of states, Σ is a finite alphabet, $\Delta \subseteq Q \times \Sigma \times Q$ is the transition relation, and I and F are the set of initial and final states, respectively. A *run* of $\mathcal{A}$ over a string $\bar{s} = a_0 \ldots a_{n-1} \in \Sigma^*$ is a non-empty sequence $p_0 \ldots p_n$ such that $p_0 \in I$, and $(p_i, a_i, p_{i+1}) \in \Delta$ for every $i < n$. We say that $\mathcal{A}$ *accepts* a string $\bar{s} \in \Sigma^*$ iff there exists such a run of $\mathcal{A}$ over $\bar{s}$ such that $p_n \in F$. We define the language $\mathcal{L}(\mathcal{A}) \subseteq \Sigma^*$ of all strings accepted by $\mathcal{A}$. Finally, we say that $\mathcal{A}$ is a *Deterministic Finite Automaton* (DFA) iff Δ is given as a partial function $\Delta : Q \times \Sigma \to Q$ and $|I| = 1$.

Schemas, tuples, and streams. Fix a set $\mathbf{D}$ of *data values.* A *relational schema* σ (or just schema) is a pair $(\mathbf{T}, \text{arity})$ where $\mathbf{T}$ are the *relation names* and arity $: \mathbf{T} \to \mathbb{N}$ maps each name to a number, that is, its *arity.* An *R-tuple* of σ (or just a tuple) is an object $R(a_0, \ldots, a_{k-1})$ such that $R \in \mathbf{T}$, each $a_i \in \mathbf{D}$, and $k = \text{arity}(R)$. We will write $R(\bar{a})$ to denote a tuple with values $\bar{a}$. We denote by $\text{Tuples}[\sigma]$ the set of all R-tuples of all $R \in \mathbf{T}$. We define the size of a tuple $R(\bar{a})$ as $|R(\bar{a})| = \sum_{i=0}^{k-1} |\bar{a}[i]|$ with $k = \text{arity}(R)$ where $|\bar{a}[i]|$ is the size of the data value $\bar{a}[i] \in \mathbf{D}$, which depends on the domain.

A *stream* $\mathcal{S}$ over σ is an infinite sequence of tuples $\mathcal{S} = t_0 t_1 t_2 \ldots$ such that $t_i \in \text{Tuples}[\sigma]$ for every $i \geq 0$. For a running example, consider the schema σ_0 with relation names $\mathbf{T} = \{R, S, T\}$, $\text{arity}(R) = \text{arity}(S) = 2$ and $\text{arity}(T) = 1$. A stream $\mathcal{S}_0$ over σ_0 could be the following (same stream as in Example 1.1):

$$\mathcal{S}_0 := \underbrace{S(2,11)}_{0} \; \underbrace{T(2)}_{1} \; \underbrace{R(1,10)}_{2} \; \underbrace{S(2,11)}_{3} \; \underbrace{T(3)}_{4} \; \underbrace{R(2,11)}_{5} \; \underbrace{S(1,10)}_{6} \; \underbrace{T(1)}_{7} \; \ldots$$

where we add an index (i.e., the position) below each tuple (for simplification, we use $\mathbf{D} = \mathbb{N}$).

Predicates. For a fixed k, a *k-predicate* P is a subset of $\text{Tuples}[\sigma]^k$. Further, we say that $\bar{t} = (t_1, \ldots, t_n)$ *satisfies* P iff $\bar{t} \in P$. We say that P is *unary* if $k = 1$ and *binary* if $k = 2$. In the following, we denote any class of unary or binary predicates by $\mathbf{U}$ or $\mathbf{B}$, respectively.

Although we define our automata models for any class of unary and binary predicates, the following two predicate classes will be relevant for algorithmic purposes (see Section 4 and 5). Let σ be a schema. We denote by $\mathbf{U}_{\text{lin}}$ the class of all unary predicates U such that, for every $t \in \text{Tuples}[\sigma]$, one can decide in linear time over $|t|$ whether t satisfies U or not. In addition, we denote by $\mathbf{B}_{\text{eq}}$ the class of all equality predicates defined as follows: a binary predicate B is an *equality predicate* iff there exist partial functions $\overleftarrow{B}$ and $\overrightarrow{B}$ over $\text{Tuples}[\sigma]$ such that, for every $t_1, t_2 \in \text{Tuples}[\sigma]$, $(t_1, t_2) \in B$ iff $\overleftarrow{B}(t_1)$ and $\overrightarrow{B}(t_2)$ are defined and $\overleftarrow{B}(t_1) = \overrightarrow{B}(t_2)$. Further, we require that one can compute $\overleftarrow{B}(t_1)$ and $\overrightarrow{B}(t_2)$ in linear time over $|t_1|$ and $|t_2|$, respectively. For example, recall our schema σ_0 and consider the binary predicate $(Tx, Sxy) = \{(T(a), S(a,b)) \mid a, b \in \mathbf{D}\}$. Then by using the functions $\overleftarrow{B}(T(a)) = a$ and $\overrightarrow{B}(S(a,b)) = a$, one can check that (Tx, Sxy) is an equality predicate.

Note that $\mathbf{B}_{\text{eq}}$ is a more general class of equality predicates compared with the ones used in [16], that will serve in our automata models for comparing tuples by "equality" in different subsets of attributes. We take here a more semantic presentation, where the equality comparison between tuples is directly given by the functions $\overleftarrow{B}$ and $\overrightarrow{B}$ and not symbolically by some formula.

Chain complex event automata. A *Chain Complex Event Automaton* (CCEA) [16] is a tuple $\mathcal{C} = (Q, \mathbf{U}, \mathbf{B}, \Omega, \Delta, I, F)$ where Q is a finite set of states, $\mathbf{U}$ is a set of unary predicates, $\mathbf{B}$ is a set of binary predicates, Ω is a finite set of labels, $I : Q \to \mathbf{U} \times (2^{\Omega} \setminus \{\varnothing\})$ is a partial initial function, $F \subseteq Q$

is the set of final states, and Δ is a finite transition relation of the form: $\Delta \subseteq Q \times \mathbf{U} \times \mathbf{B} \times (2^{\Omega} \setminus \{\varnothing\}) \times Q$. Let $\mathcal{S} = t_0 t_1 \ldots$ be a stream. A *configuration* of $\mathcal{C}$ over $\mathcal{S}$ is a tuple $(p, i, L) \in Q \times \mathbb{N} \times (2^{\Omega} \setminus \{\varnothing\})$, representing that the automaton $\mathcal{C}$, is at state p after having read and marked t_i with the set of labels L. For $\ell \in \Omega$, we say that (p, i, L) *marked* position i with ℓ iff $\ell \in L$. Given a position $n \in \mathbb{N}$, we say that a configuration is *accepting* iff it is of the form (p, n, L) and $p \in F$. Then a *run* ρ of $\mathcal{C}$ over $\mathcal{S}$ is a sequence of configurations:

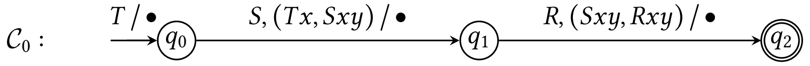
$$\rho \ := \ (p_0, i_0, L_0), (p_1, i_1, L_1), \ldots, (p_n, i_n, L_n)$$

such that $i_0 < i_1 < \ldots < i_n$, $I(p_0) = (U, L_0)$ is defined and $t_{i_0} \in U$, and there exists a transition $(p_{j-1}, U_j, B_j, L_j, p_j) \in \Delta$ such that $t_{i_j} \in U_j$ and $(t_{i_{j-1}}, t_{i_j}) \in B_j$ for every $j \in [1, n]$. Intuitively, a run of a CCEA is a subsequence of the stream that can follow a path of transitions, where each transition checks a local condition (i.e., the unary predicate U_j) and a join condition (i.e., the binary predicate B_j) with the previous tuple. For the first tuple, a CCEA can only check a local condition (i.e., there is no previous tuple).

Given a run ρ like above, we define its *valuation* $\nu_\rho : \Omega \to 2^{\mathbb{N}}$ such that $\nu_\rho(\ell)$ is the set consisting of all positions in ρ marked by ℓ, formally, $\nu_\rho(\ell) = \{i_j \mid j \leq n \wedge \ell \in L_j\}$. Further, given a position $i_n \in \mathbb{N}$, we say that ρ is an *accepting run at position n* iff (p_n, i_n, L_n) is an accepting configuration. Then the *output* of $\mathcal{C}$ over $\mathcal{S}$ at position n is defined as:

$$[\![\mathcal{C}]\!]_n(\mathcal{S}) \ = \ \{\nu_\rho \mid \rho \text{ is an accepting run at position } n \text{ of } \mathcal{C} \text{ over } S\}.$$

Example 2.1. Below, we show an example of a CCEA over the schema σ_0 with $\Omega = \{\bullet\}$:



We use T to denote the predicate $T = \{T(a) \mid a \in \mathbf{D}\}$ and similar for S and R. Further, we use (Tx, Sxy) and (Sxy, Rxy) to denote equality predicates as defined above. An accepting run of $\mathcal{C}_0$ over $\mathcal{S}_0$ is $\rho = (q_0, 1, \{\bullet\}), (q_1, 3, \{\bullet\}), (q_2, 5, \{\bullet\})$ which produces the valuation $\nu_\rho = \{\bullet \mapsto \{1, 3, 5\}\}$ that represents the subsequence $T(2), S(2, 11), R(2, 11)$ of $\mathcal{S}_0$. Intuitively, $\mathcal{C}_0$ defines all subsequences of the form $T(a), S(a, b), R(a, b)$ for every $a, b \in \mathbf{D}$.

Note that the definition of CCEA above differs from [16] to fit our purpose better. Specifically, we use a set of labels Ω to annotate positions in the streams and define valuations in the same spirit as the model of *annotated automata* used in [3, 23]. One can see this extension as a generalization to the model in [16], where $|\Omega| = 1$. This extension will be helpful to enrich the outputs of our models for comparing them with hierarchical conjunctive queries with self-joins (see Section 4).

Computational model. For our algorithms, we assume the computational model of Random Access Machines (RAM) with uniform cost measure, and addition as its basic operation [1, 15]. This RAM has read-only registers for the input, read-writes registers for the work, and write-only registers for the output. This computation model is a standard assumption in the literature [5, 6].

3 Parallelized complex event automata

This section presents our automata model for specifying CER queries with conjunction called Parallelized Complex Event Automata (PCEA), which strictly generalized CCEA by adding a new feature called *parallelization*. For the sake of presentation, we first formalize the notion of parallelization for NFA to extend the idea to CCEA. Before this, we need the notation of labeled trees that will be useful for our definitions and proofs.

Labeled trees. As it is common in the area [24], we define (unordered) *trees* as a finite set of strings $t \subseteq \mathbb{N}^*$ that satisfies two conditions: (1) t contains the empty string, (i.e., $\varepsilon \in t$), and (2) t is a *prefix-closed set*, namely, if $a_1 ... a_n \in t$, then $a_1 ... a_j \in t$ for every $j < n$. We will refer to the strings of t as *nodes*, and the *root* of a tree, $\mathtt{root}(t)$, will be the empty string ε.

Let $\bar{u}, \bar{v} \in t$ be nodes. The *depth* of $\bar{u}$ will be given by its length $\mathtt{depth}_t(\bar{u}) = |\bar{u}|$. We say that $\bar{u}$ is the *parent* of $\bar{v}$ and write $\mathtt{parent}_t(\bar{v}) = \bar{u}$ if $\bar{v} = \bar{u} \cdot n$ for some $n \in \mathbb{N}$. Likewise, we say that $\bar{v}$ is a *child* of $\bar{u}$ if $\bar{u}$ is the parent of $\bar{v}$ and define $\mathtt{children}_t(\bar{u}) = \{\bar{v} \in t \mid \mathtt{parent}_t(\bar{v}) = \bar{u}\}$. Similarly, we define the *descendants* of $\bar{u}$ as $\mathtt{desc}_t(\bar{u}) = \{\bar{v} \in t \mid \bar{u} \leq_p \bar{v}\}$ and the *ancestors* as $\mathtt{ancst}_t(\bar{u}) = \{\bar{v} \in t \mid \bar{v} \leq_p \bar{u}\}$; note that $\bar{u} \in \mathtt{desc}_t(\bar{u})$ and $\bar{u} \in \mathtt{ancst}_t(\bar{u})$. A node $\bar{u}$ is a *leaf* of t if $\mathtt{desc}_t(\bar{u}) = \{\bar{u}\}$, and an *inner node* if it is not a leaf node. We define the *set of leaves* of $\bar{u}$ as $\mathtt{leaves}_t(\bar{u}) = \{\bar{v} \in \mathtt{desc}_t(\bar{u}) \mid \bar{v} \text{ is a leaf node}\}$.

A *labeled tree* τ is a function $\tau : t \to L$ where t is a tree and L is any finite set of labels. We use $\mathrm{dom}(\tau)$ to denote the underlying tree structure t of τ. Given that τ is a function, we can write $\tau(\bar{u})$ to denote the label of node $\bar{u} \in \mathrm{dom}(\tau)$. To simplify the notation, we extend all the definitions above for a tree t to labeled tree τ, changing t by $\mathrm{dom}(\tau)$. For example, we write $\bar{u} \in \tau$ to refer to $\bar{u} \in \mathrm{dom}(\tau)$, or $\mathtt{parent}_\tau(\bar{u})$ to refer to $\mathtt{parent}_{\mathrm{dom}(\tau)}(\bar{u})$. Finally, we say that two labeled trees τ and τ' are *isomorphic* if there exists a bijection $f : \mathrm{dom}(\tau) \to \mathrm{dom}(\tau')$ such that $\bar{u} \leq_p \bar{v}$ iff $f(\bar{u}) \leq_p f(\bar{v})$ and $\tau(\bar{u}) = \tau'(f(\bar{u}))$ for every $\bar{u}, \bar{v} \in \mathrm{dom}(\tau)$. We will usually say that τ and τ' are equal, meaning they are isomorphic.

Parallelized finite automata. A *Parallelized Finite Automaton* (PFA) is a tuple $\mathcal{P} = (Q, \Sigma, \Delta, I, F)$ where Q is a finite set of states, Σ is a finite alphabet, $I, F \subseteq Q$ are the sets of initial and accepting states, respectively, and $\Delta \subseteq 2^Q \times \Sigma \times Q$ is the transition relation. We define the size of $\mathcal{P}$ as $|\mathcal{P}| = |Q| + \sum_{(P,a,q) \in \Delta}(|P| + 1)$, namely, the number of states plus the size of encoding the transitions.

A *run tree* of a PFA $\mathcal{P}$ over a string $\bar{s} = a_1 \ldots a_n \in \Sigma^*$ is a labeled tree $\tau : t \to Q$ such that $\mathtt{depth}_\tau(\bar{u}) = n$ for every leaf $\bar{u} \in \tau$; in other words, every node of τ is labeled by a state of $\mathcal{P}$ and all branches have the same length n. In addition, τ must satisfy the following two conditions: (1) every leaf node $\bar{u}$ of t is labeled by an initial state (i.e., $\tau(\bar{u}) \in I$) and (2) for every inner node $\bar{v}$ at depth i (i.e., $\mathtt{depth}_\tau(\bar{v}) = i$) there must be a transition $(P, a_{n-i}, q) \in \Delta$ such that $\tau(\bar{v}) = q$, $|\mathtt{children}_\tau(\bar{v})| = |P|$ and $P = \{\tau(\bar{u}) \mid \bar{u} \in \mathtt{children}_\tau(\bar{v})\}$, that is, children have different labels and P is the set of labels in the children of $\bar{v}$. We say that τ is an accepting run of $\mathcal{P}$ over $\bar{s}$ iff τ is a run of $\mathcal{P}$ over $\bar{s}$ and $\tau(\varepsilon) \in F$ (recall that $\varepsilon = \mathtt{root}(\tau)$). We say that $\mathcal{P}$ accepts a string $\bar{s} \in \Sigma^*$ if there is an accepting run of $\mathcal{P}$ over $\bar{s}$ and we define the language recognized by $\mathcal{P}$, $\mathcal{L}(\mathcal{P})$, as the set of strings that $\mathcal{P}$ accepts.

Example 3.1. In Figure 1 (left), we show the example of a PFA $\mathcal{F}_0$ over the alphabet $\Sigma = \{T, S, R\}$. Intuitively, the upper part (i.e., p_0, p_1) looks for a symbol T, the lower part (i.e., p_2, p_3) for a symbol S, and both runs join together in p_4 when they see a symbol R. Then, $\mathcal{F}_0$ defines all strings that contain symbols T and S (in any order) before a symbol R.

One can see that PFA is a generalization of an NFA. Indeed, NFA is a special case of an PFA where each run tree τ is a line. Nevertheless, PFA do not add expressive power to NFA, given that PFA is another model for recognizing regular languages, as the next result shows.

PROPOSITION 3.2. *For every PFA $\mathcal{P}$ with n states there exists a DFA $\mathcal{A}$ with at most 2^n states such that $\mathcal{L}(\mathcal{P}) = \mathcal{L}(\mathcal{A})$. In particular, all languages defined by PFA are regular.*

Intuitively, one could interpret a PFA as an *Alternating Finite Automaton* (AFA) [7] that runs backward over the string (however, they still process the string in a forward direction). It was shown in [7, Theorem 5.2 and 5.3] that for every AFA that defines a language L with n states, there exists an equivalent DFA with 2^{2^n} states in the worst case that recognizes L. Nevertheless, they argued that the reverse language $L^R = \{a_1 a_2 \ldots a_n \in \Sigma^* \mid a_n \ldots a_2 a_1 \in L\}$ can always be accepted by a DFA with at most 2^n states. Then, one can see Proposition 3.2 as a consequence of reversing

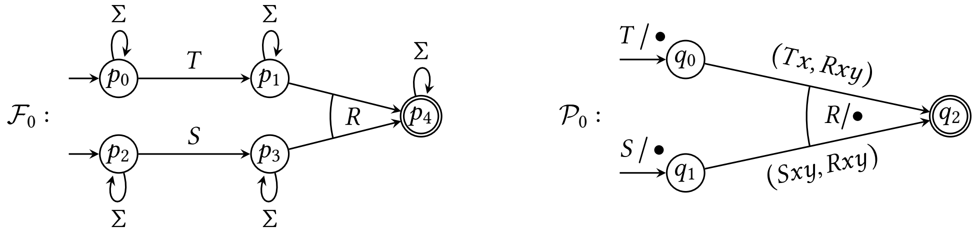


Fig. 1. On the left, an example of a PFA and, on the right, an example of a PCEA.

an alternating automaton. Despite this connection, we use here PFA as a proper automata model, which was not studied or used in [7]. Another related proposal is the *parallel finite automata* model presented in [27]. Indeed, one can consider PFA as a restricted case of this model, although it was not studied in [27]. For this reason, we decided to name the PFA model with the same acronym but a slightly different name as in [27].

Parallelized complex event automata. A *Parallelized Complex Event Automaton* (PCEA) is the extension of CCEA with the idea of parallelization as in PFA. Specifically, a PCEA is a tuple $\mathcal{P} = (Q, \mathbf{U}, \mathbf{B}, \Omega, \Delta, F)$, where Q, $\mathbf{U}$, $\mathbf{B}$, Ω, and F are the same as for CCEA, and Δ is a finite transition relation of the form:

$$\Delta \subseteq 2^Q \times \mathbf{U} \times \mathbf{B}^Q \times (2^\Omega \setminus \{\varnothing\}) \times Q.$$

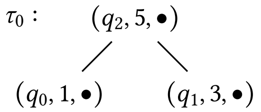
where $\mathbf{B}^Q$ are all partial functions $\mathcal{B} : Q \to \mathbf{B}$, that associate a state q to a binary predicate $\mathcal{B}(q)$. We define the *size of* $\mathcal{P}$ as $|\mathcal{P}| = |Q| + \sum_{(P,U,\mathcal{B},L,q)\in\Delta}(|P| + |L|)$. Note that $\mathcal{P}$ does not define the initial function explicitly. As we will see, transitions of the form $(\varnothing, U, \mathcal{B}, L, q)$ will play the role of the initial function on a run of $\mathcal{P}$.

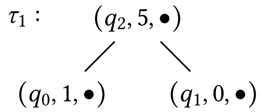
Next, we extend the notion of a run from CCEA to its parallelized version. Let $\mathcal{S} = t_0 t_1 \dots$ be a stream. A *run tree* of $\mathcal{P}$ over $\mathcal{S}$ is now a labeled tree $\tau : t \to (Q \times \mathbb{N} \times (2^\Omega \setminus \{\varnothing\}))$ where each node $\bar{u} \in \tau$ is labeled with a configuration $\tau(\bar{u}) = (q, i, L)$ such that, for every child $\bar{v} \in \mathtt{children}_\tau(\bar{u})$ with $\tau(\bar{v}) = (p, j, M)$, it holds that $j < i$. In other words, the positions of τ-configurations increase towards the root of τ, similar to the runs of a CCEA. In addition, $\bar{u}$ must satisfy the transition relation Δ, that is, there must exist a transition $(P, U, \mathcal{B}, L, q) \in \Delta$ such that (1) $t_i \in U$, (2) $|\mathtt{children}_\tau(\bar{u})| = |P|$ and $P = \{p \mid \exists \bar{v} \in \mathtt{children}_\tau(\bar{u}). \tau(\bar{v}) = (p, j, M)\}$, and (3) for every $\bar{v} \in \mathtt{children}_\tau(\bar{u})$ with $\tau(\bar{v}) = (p, j, M)$, $(t_j, t_i) \in \mathcal{B}(p)$. Similar to PFA, condition (2) forces that there exists a bijection between P and the states at the children of $\bar{u}$. Instead, condition (3) forces that two consecutive configurations (p, j, M) and (q, i, L) must satisfy the binary predicate in $\mathcal{B}(p)$ associated with p. Notice that, if $\bar{u}$ is a leaf node in τ, then it must hold that $P = \varnothing$ and condition (3) is trivially satisfied. Also, note that we do not assume that all leaves are at the same depth.

Given a position $n \in \mathbb{N}$, we say that τ is an *accepting run at position n* iff the root configuration $\tau(\varepsilon)$ is accepting at position n. Further, we define the output of a run τ as the valuation $\nu_\tau : \Omega \to 2^{\mathbb{N}}$ such that $\nu_\tau(\ell) = \{i \mid \exists \bar{u} \in \tau.\ \tau(\bar{u}) = (q, i, L) \wedge \ell \in L\}$ for every label $\ell \in \Omega$. Finally, the output of a PCEA $\mathcal{P}$ over $\mathcal{S}$ at the position n is defined as:

$$[\![\mathcal{P}]\!]_n(\mathcal{S}) \ = \ \{\nu_\tau \mid \tau \text{ is an accepting run at position } n \text{ of } \mathcal{P} \text{ over } S\}.$$

Example 3.3. In Figure 1 (right), we show an example of a PCEA $\mathcal{P}_0$ over schema σ_0 with $\Omega = \{\bullet\}$. We use the same notation as in Example 2.1 to represent unary and equality predicates. If we run $\mathcal{P}_0$ over $\mathcal{S}_0$, we have the following two run trees at position 5:





that produces the valuation $\nu_{\tau_0} = \{\bullet \mapsto \{1, 3, 5\}\}$ and $\nu_{\tau_1} = \{\bullet \mapsto \{0, 1, 5\}\}$ representing the subsequences $T(2), S(2, 11), R(2, 11)$ and $S(2, 11), T(2), R(2, 11)$ of $\mathcal{S}_0$, respectively. Note that the former is an output of $\mathcal{C}_0$ in Example 2.1, but the latter is not.

Proposition 3.4. *PCEA is strictly more expressive than CCEA.*

The proof for this proposition can be found in the appendix, but it is easy to see that every CCEA is a PCEA where every transition $(P, U, \mathcal{B}, L, q) \in \Delta$ satisfies that $|P| \leq 1$. Additionally, the previous example gives evidence that PCEA is a strict generalization of CCEA, namely, there exists no CCEA that can define $\mathcal{P}_0$. Intuitively, since a CCEA can only compare the current tuple to the last tuple, for a stream like $\mathcal{S} = R(a, b), T(a), S(a, b)$ it would be impossible to check conditions over the second attribute of tuples $R(a, b)$ and $S(a, b)$.

Unambiguous PCEA. We end this section by introducing a subclass of PCEA relevant to our algorithmic results. Let $\mathcal{P}$ be a PCEA and τ a run of $\mathcal{P}$ over some stream. We say that τ is *simple* iff for every two different nodes $\bar{u}, \bar{u}' \in \tau$ with $\tau(\bar{u}) = (q, i, L)$ and $\tau(\bar{u}') = (q', i', L')$, if $i = i'$, then $L \cap L' = \emptyset$. In other words, τ is simple if all positions of the valuation ν_τ are uniquely represented in τ. We say that $\mathcal{P}$ is *unambiguous* if (1) every accepting run of $\mathcal{P}$ is simple and (2) for every stream $\mathcal{S}$ and accepting run τ of $\mathcal{P}$ over $\mathcal{S}$ with valuation ν_τ, there is no other run τ' of $\mathcal{P}$ with valuation $\nu_{\tau'}$ such that $\nu_\tau = \nu_{\tau'}$. For example, the reader can check that $\mathcal{P}_0$ is unambiguous.

Condition (2) of unambiguous PCEA ensures that each output is witnessed by exactly one run. This condition is common in MSO enumeration [2, 22] for a one-to-one correspondence between outputs and runs. Condition (1) forces a correspondence between the size of the run and the size of the output it represents. As we will see, both conditions will be helpful for our evaluation algorithm, and satisfied by our translation of hierarchical conjunctive queries into PCEA in the next section.

4 Representing hierarchical conjunctive queries

This section studies the connection between PCEA and hierarchical conjunctive queries (HCQ) over streams. For this purpose, we must first define the semantics of HCQ over streams and how to relate their expressiveness with PCEA. We connect them by using a *bag semantics* of CQ. We start by introducing bags that will be useful throughout this section.

Bags. A bag (also called a multiset) is usually defined in the literature as a function that maps each element to its multiplicity (i.e., the number of times it appears). In this work, we use a different but equivalent representation of a bag where each element has its *own identity*. This representation will be helpful in our context to deal with duplicates in the stream and define the semantics of hierarchical CQ in the case of self-joins.

We define a *bag* (with own identity) B as a surjective function $B : I \to U$ where I is a finite set of identifiers (i.e., the identity of each element) and U is the underlying set of the bag. Given any bag B, we refer to these components as $I(B)$ and $U(B)$, respectively. For example, a bag $B = \{\!\{a, a, b\}\!\}$ (where a is repeated twice) can be represented with a surjective function $B_0 = \{0 \mapsto a, 1 \mapsto a, 2 \mapsto b\}$ where $I(B_0) = \{0, 1, 2\}$ and $U(B_0) = \{a, b\}$. In general, we will use the standard notation for bags $\{\!\{a_0, \ldots, a_{n-1}\}\!\}$ to denote the bag B whose identifiers are $I(B) = \{0, \ldots, n-1\}$ and $B(i) = a_i$ for each $i \in I(B)$. Note that if $B : I \to U$ is injective, then B encodes a set (i.e., no repetitions). We write $a \in B$ if $B(i) = a$ for some $i \in I(B)$ and define the empty bag $\emptyset$ such that $I(\emptyset) = \emptyset$ and $U(\emptyset) = \emptyset$.

For a bag B and an element a, we define the *multiplicity* of a in B as $\text{mult}_B(a) = |\{i \mid B(i) = a\}|$. Then, we say that a bag B' is *contained* in B, denoted as $B' \subseteq B$, iff $\text{mult}_{B'}(a) \leq \text{mult}_B(a)$ for every a. We also say that two bags B' and B are *equal*, and write $B = B'$, if $B' \subseteq B$ and $B \subseteq B'$. Note that two bags can be equal although the set of identifiers can be different (i.e., they are equal up to a renaming of the identifiers). Given a set A, we say that B *is a bag from elements of* A (or just a bag of A) if $U(B) \subseteq U(A)$.

Relational databases. Recall that $\mathbf{D}$ is our set of data values and let $\sigma = (\mathbf{T}, \text{arity})$ be a schema. A *relational database* D (with duplicates) over σ is a bag of Tuples$[\sigma]$. Given a relation name $R \in \mathbf{T}$, we write R^D as the bag of D containing only the R-tuples of D, formally, $I(R^D) = \{i \in I(D) \mid D(i) = R(\bar{a})$ for some $\bar{a}\}$ and $R^D(i) = D(i)$ for every $i \in I(R^D)$. For example, consider again the schema σ_0. Then a database D_0 over σ_0 is the bag:

$$D_0 \;:=\; \{\!\{\, S(2,11), T(2), R(1,10), S(2,11), T(3), R(2,11) \,\}\!\}.$$

Here, one can check that $T^{D_0} = \{\!\{T(2), T(3)\}\!\}$ and $S^{D_0} = \{\!\{S(2,11), S(2,11)\}\!\}$.

Conjunctive queries. Fix a schema $\sigma = (\mathbf{T}, \text{arity})$ and a set of variables $\mathbf{X}$ disjoint from $\mathbf{D}$ (i.e., $\mathbf{X} \cap \mathbf{D} = \emptyset$). A *Conjunctive Query* (CQ) over relational schema σ is a syntactic structure of the form:

$$Q(\bar{x}) \;\leftarrow\; R_0(\bar{x}_0), \ldots, R_{m-1}(\bar{x}_{m-1}) \tag{†}$$

such that Q is a relational name not in $\mathbf{T}$, $R_i \in \mathbf{T}$, $\bar{x}_i$ is a sequence of variables in $\mathbf{X}$ and data values in $\mathbf{D}$, and $|\bar{x}_i| = \text{arity}(R_i)$ for every $i < m$. Further, $\bar{x}$ is a sequence of variables in $\bar{x}_0, \ldots, \bar{x}_{m-1}$. We will denote a CQ like (†) by Q, where $Q(\bar{x})$ and $R_0(\bar{x}_0), \ldots, R_{m-1}(\bar{x}_{m-1})$ are called the *head* and the *body* of Q, respectively. Furthermore, we call each $R_i(\bar{x}_i)$ an *atom* of Q. For example, the following are two conjunctive queries Q_0 (the same CQ of Example 1.1) and Q_1 over the schema σ_0:

$$Q_0(x,y) \leftarrow T(x),\, S(x,y),\, R(x,y) \qquad\qquad Q_1(x,y) \leftarrow T(x),\, R(x,y),\, S(2,y),\; T(x)$$

Note that a query can repeat atoms. For this reason, we will regularly consider Q as a bag of atoms, where $I(Q)$ are the positions of Q and $U(Q)$ is the set of distinct atoms. For instance, we can consider Q_1 above as a bag of atoms, where $I(Q_1) = \{0, 1, 2, 3\}$ (i.e., the position of the atoms) and $Q_1(0) = T(x)$, $Q_1(1) = R(x,y)$, $Q_1(2) = S(2,y)$, $Q_1(3) = T(x)$. We say that a CQ Q has *self-joins* if there are two atoms with the same relation name. We can see in the previous example that Q_1 has self-joins, while Q_0 does not.

Homomorphisms and CQ bag semantics. Let Q be a CQ, and D be a database over the same schema σ. A *homomorphism* is any function $h : \mathbf{X} \cup \mathbf{D} \to \mathbf{D}$ such that $h(a) = a$ for every $a \in \mathbf{D}$. We extend h as a function from atoms to tuples such that $h(R(\bar{x})) := R(h(\bar{x}))$ for every atom $R(\bar{x})$. We say that h is a homomorphism from Q to D if h is a homomorphism and $h(R(\bar{x})) \in D$ for every atom $R(\bar{x})$ in Q. We denote by $\text{Hom}(Q, D)$ the set of all homomorphisms from Q to D.

To define the bag semantics of CQ, we need a more refined notion of homomorphism that specifies the correspondence between atoms in Q and tuples in D. Formally, a *tuple-homomorphism* from Q to D (or *t-homomorphism* for short) is a function $\eta : I(Q) \to I(D)$ such that there exists a homomorphism h_η from Q to D satisfying that $h_\eta(Q(i)) = D(\eta(i))$ for every $i \in I(Q)$. For example, consider again Q_0 and D_0 above, then $\eta_0 = \{0 \mapsto 1, 1 \mapsto 3, 2, \mapsto 5\}$ and $\eta_1 = \{0 \mapsto 1, 1 \mapsto 0, 2 \mapsto 5\}$ are two t-homomorphism from Q_0 to D_0.

Intuitively, a t-homomorphism is like a homomorphism, but it additionally specifies the correspondence between atoms (i.e., $I(Q)$) and tuples (i.e., $I(D)$) in the underlying bags. One can easily check that if η is a t-homomorphism, then h_η (restricted to the variables of Q) is unique. For this reason, we usually say that h_η is the homomorphism associated to η. Note that the converse does not hold: for h from Q to D, there can be several t-homomorphisms η such that $h = h_\eta$.

Let $Q(\bar{x})$ be the head of Q. We define the output of a CQ Q over a database D as:

$$[\![Q]\!](D) \;=\; \{\!\{Q(h_\eta(\bar{x})) \mid \eta \text{ is a t-homomorphism from } Q \text{ to } D\}\!\}.$$

Note that the result is another relation where each $Q(h_\eta(\bar{x}))$ is witnessed by a t-homomorphism from Q to D. In other words, there is a one-to-one correspondence between tuples in $[\![Q]\!](D)$ and t-homomorphisms from Q to D.

Discussion. In the literature, homomorphisms are usually used to define the set semantics of a CQ Q over a database D. They are helpful for set semantics but "inconvenient" for bag semantics since it does not specify the correspondence between atoms and tuples; namely, they only witness the existence of such correspondence. In [8], Chaudhuri and Vardi introduced the bag semantics of CQ by using homomorphisms, which we recall next. Let Q be a CQ like (†) and D a database over the same schema σ, and let $h \in \mathrm{Hom}(Q, D)$. We define the *multiplicity* of h with respect to Q and D by:

$$\mathrm{mult}_{Q,D}(h) \;=\; \prod_{i=0}^{m-1} \mathrm{mult}_D(h(R_i(\bar{x}_i)))$$

Chaudhuri and Vardi defined the bag semantics $[Q]$ of Q over D as the bag $[Q](D)$ such that each tuple $Q(\bar{a})$ has multiplicity equal to:

$$\mathrm{mult}_{[Q](D)}(Q(\bar{a})) \;=\; \sum_{h \in \mathrm{Hom}(Q,D)\,:\, h(\bar{x})=\bar{a}} \mathrm{mult}_{Q,D}(h)$$

In the appendix, we prove that for every CQ Q and database D it holds that $[\![Q]\!](D) = [Q](D)$, namely, the bag semantics introduced here (i.e., with t-homomorphisms) is equivalent to the standard bag semantics of CQ. The main difference is that the standard bag semantics of CQ are defined in terms of homomorphisms and multiplicities, and there is no direct correspondence between outputs and homomorphisms. For this reason, we redefine the bag semantics of CQ in terms of t-homomorphism that will connect the outputs of CQ with the outputs of PCEA over streams.

CQ over streams. Now, we define the semantics of CQ over streams, formalizing its comparison with queries in complex event recognition. For this purpose, we must show how to interpret streams as databases and encode CQ's outputs as valuations. Fix a schema σ and a stream $\mathcal{S} = t_0 t_1 \cdots$ over σ. Given a position $n \in \mathbb{N}$, we define the database of $\mathcal{S}$ at position n as the σ-database $D_n[\mathcal{S}] = \{\!\{t_0, t_1, \ldots, t_n\}\!\}$. For example, $D_5[\mathcal{S}_0] = D_0$. One can interpret here that $\mathcal{S}$ is a sequence of inserts, and then $D_n[\mathcal{S}]$ is the database version at position n. Since $D_n[\mathcal{S}]$ is a bag, the identifiers $I(D_n[\mathcal{S}])$ coincide with the positions of the sequence $t_0 \ldots t_n$.

Let Q be a CQ over σ, and let $\eta : I(Q) \to I(D_n[\mathcal{S}])$ be a t-homomorphism from Q to $D_n[\mathcal{S}]$. If we consider $\Omega = I(Q)$, we can interpret η as a *valuation* $\hat{\eta} : \Omega \to 2^{\mathbb{N}}$ that maps each atom of Q to a set with a single position; formally, $\hat{\eta}(i) = \{\eta(i)\}$ for every $i \in I(Q)$. Then, we define the semantics of Q over stream $\mathcal{S}$ at position n as:

$$[\![Q]\!]_n(\mathcal{S}) \;=\; \{\hat{\eta} \mid \eta \text{ is a t-homomorphism from } Q \text{ to } D_n[\mathcal{S}]\}$$

Note that $[\![Q]\!]_n(\mathcal{S})$ is equivalent to evaluating Q over $D_n[\mathcal{S}]$ where instead of outputting a bag of tuples $[\![Q]\!](D_n[\mathcal{S}])$, we output the t-homomorphisms (i.e., as valuations) that are in a one-to-one correspondence with the tuples in $[\![Q]\!](D_n[\mathcal{S}])$.

Hierarchical conjunctive queries and main results. Let Q be a CQ of the form (†). Given a variable $x \in \mathbf{X}$, define *atoms*(x) as the bag of all atoms $R_i(\bar{x}_i)$ of Q such that x appears in $\bar{x}_i$. We say that Q is *full* if every variable appearing in $\bar{x}_0, \ldots, \bar{x}_{m-1}$ also appears in $\bar{x}$. Then, Q is a *Hierarchical Conjunctive Query* (HCQ)[12] iff Q is full and for every pair of variables $x, y \in \mathbf{X}$ it holds that *atoms*$(x) \subseteq$ *atoms*(y), *atoms*$(y) \subseteq$ *atoms*(x) or *atoms*$(x) \cap$ *atoms*$(y) = \emptyset$. For example, one can check that Q_0 is an HCQ, but Q_1 is not.

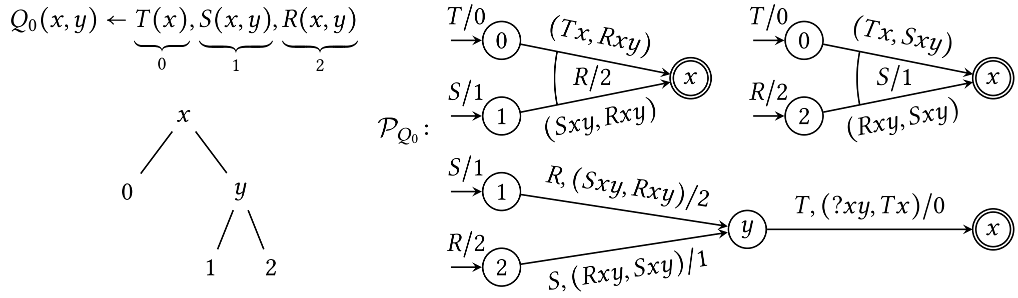


Fig. 2. An illustration of constructing an PCEA from an HCQ. On the left, the HCQ Q_0 and its q-tree. On the right, a PCEA $\mathcal{P}_{Q_0}$ equivalent to Q_0. For presentation purposes, states are repeated several times and $?xy$ means a binary relation with any relation name (i.e., R or S).

HCQ is a subset of CQ that can be evaluated with constant-delay enumeration under updates [5, 18]. Moreover, it is the greatest class of full conjunctive queries that can be evaluated with such guarantees under fine-grained complexity assumptions. Therefore, HCQ is the right yardstick to measure the expressive power of PCEA for defining queries with strong efficiency guarantees. Given a PCEA $\mathcal{P}$ and a CQ Q over the same schema σ, we say that $\mathcal{P}$ is *equivalent* to Q (denoted as $\mathcal{P} \equiv Q$) iff for every stream $\mathcal{S}$ over σ and every position n it holds that $[\![\mathcal{P}]\!]_n(\mathcal{S}) = [\![Q]\!]_n(\mathcal{S})$.

THEOREM 4.1. *Let σ be a schema. For every HCQ Q over σ, there exists a PCEA $\mathcal{P}_Q$ over σ with unary predicates in $\boldsymbol{U}_{\text{lin}}$ and binary predicates in $\boldsymbol{B}_{\text{eq}}$ such that $\mathcal{P}_Q \equiv Q$. Furthermore, $\mathcal{P}_Q$ is unambiguous and of at most exponential size with respect to Q. If Q does not have self-joins, then $\mathcal{P}_Q$ is of quadratic size.*

PROOF SKETCH. We give an example of the construction to provide insights on the expressive power of PCEA for defining HCQ (the full technical proof is in the extended version [25]). For this construction, we rely on a q-tree of an HCQ, a structure introduced in [5]. Formally, let Q be an HCQ and assume, for the sake of simplification, that Q is connected (i.e., the Gaifman graph associated to Q is connected). A *q-tree* for Q is a labeled tree, $\tau_Q : t \to I(Q) \cup \{\bar{x}\}$, where for every $x \in \{\bar{x}\}$ there is a unique inner node $\bar{u} \in t$ such that $\tau_Q(\bar{u}) = x$, and for every atom $i \in I(Q)$ there is a unique leaf node $\bar{v} \in t$ such that $\tau_Q(\bar{v}) = i$. Further, if $\bar{u}_1, \ldots, \bar{u}_k$ are the inner nodes of the path from the root until $\bar{v}$, then $\{\bar{x}_i\} = \{\tau_Q(\bar{u}_1), \ldots, \tau_Q(\bar{u}_k)\}$. In [5], it was shown that a CQ Q is hierarchical and connected iff there exists a q-tree for Q. For instance, in Figure 2 (left) we display again the HCQ Q_0, labeled with the identifiers of the atoms, and below a q-tree for Q_0.

For a connected HCQ without self-joins, the idea of the construction is to use the q-tree of Q as the underlying structure of the PCEA $\mathcal{P}_Q$. Indeed, the nodes of the q-tree will be the states of $\mathcal{P}_Q$. For example, in Figure 2 (right) we present a PCEA $\mathcal{P}_{Q_0}$ equivalent to Q_0, where we use multiple copies of the states for presentation purposes (i.e., if two states have the same label, they are the same state in the figure). As you can check, the states are $\{0, 1, 2, x, y\}$, which are the nodes of the q-tree. Furthermore, the leaves of the q-trees (i.e., the atoms) are the initial states $\{0, 1, 2\}$ where $\mathcal{P}_{Q_0}$ uses a unary predicate to check that the tuples have arrived and annotates with the corresponding identifier.

For every atom $R_i(\bar{x}_i)$ and every variable $x \in \{\bar{x}_i\}$, $\mathcal{P}_Q$ jumps with a transition to the state x which is a node in the q-tree and joins with all the atoms and variables "hanging" from the path from x to the leave i in the q-tree. For example, consider the first component (i.e., top-left) of $\mathcal{P}_{Q_0}$ in Figure 2. When $\mathcal{P}_{Q_0}$ reads a tuple $R(a, b)$, it jumps to state x and joins with all the atoms hanging from the path from x to 2, namely, the atoms T and S. Similarly, consider the last component (i.e., below) of $\mathcal{P}_{Q_0}$ in Figure 2. When $\mathcal{P}_{Q_0}$ reads a tuple $R(a, b)$, it also jumps to state y, but now the only atom hanging from the path from y to 2 in the q-tree is 1, which corresponds to a single transition from 1 to y joining with the atom $S(x, y)$. Finally, when $\mathcal{P}_{Q_0}$ reads a tuple $T(a)$, the only variable that hangs in the path from the root to 0 is the variable y, and then there is a single transition from y to x, joining with an equality predicate $(?xy, Tx)$ where $?xy$ means a binary relation with any relational name (i.e., R or S). Finally, the root of the q-tree serves as the final state of the $\mathcal{P}_{Q_0}$, namely, all atoms were found. Note that an accepting run tree of $\mathcal{P}_{Q_0}$ serves as a witness that the q-tree is complete. The construction of HCQ with self-joins is more involved, and we present the details in the extended version [25]. □

The previous result shows that PCEA has the expressive power to specify every HCQ. Given that HCQ characterize the full CQ that can be evaluated in a dynamic setting (under complexity assumptions), a natural question is to ask whether PCEA has the *right* expressive power, in the sense that it cannot define CQ that cannot be evaluated efficiently (i.e., non-hierarchical CQ). We answer this question positively by focusing on full CQ.

Theorem 4.2. *Let σ be a schema. For every full CQ Q over σ, if $\mathcal{P} \equiv Q$ for some PCEA $\mathcal{P}$ over σ, then Q is hierarchical.*

The reader can find the proof of this theorem in the appendix. By combining Theorem 4.1 and 4.2, we get the following result stating that PCEA exactly captures the expressive power of HCQ.

Corollary 4.3. *Let σ be a schema. For every full CQ Q over σ, Q is hierarchical if, and only if, $\mathcal{P} \equiv Q$ for some PCEA $\mathcal{P}$ over σ.*

We note that, although PCEA can only define full CQ that are hierarchical, it can define queries that are not CQ. For instance, $\mathcal{P}_0$ in Example 3.3 cannot be defined by any CQ, since a CQ cannot express that the R-tuple must arrive after T and S. Therefore, the class of queries defined by PCEA is strictly more expressive than HCQ.

By Corollary 4.3, PCEA capture the expressibility of HCQ among full CQ. In the next section, we show that they also share their good algorithmic properties for streaming evaluation.

5 An evaluation algorithm for PCEA

Below, we present our evaluation algorithm for unambiguous PCEA with equality predicates. We do this in a streaming setting where the algorithm reads a stream sequentially, and at each position, we can enumerate the new outputs fired by the last tuple. Furthermore, our algorithm works under a *sliding window* scenario, where we only want to enumerate the outputs inside the last w items for some window size w. This scenario is motivated by CER [6, 11, 14], where the importance of data decreases with time, and then, we want the outputs inside some relevant time window.

In the following, we start by defining the evaluation problem and stating the main theorem, followed by describing our data structure for storing valuations. We end this section by explaining the algorithm and stating its correctness.

The streaming evaluation problem. Let σ be a fixed schema. For a valuation $\nu : \Omega \to 2^{\mathbb{N}}$, we define $\min(\nu) = \min\{i \mid \exists \ell \in \Omega.\ i \in \nu(\ell)\}$, namely, the minimum position appearing in ν. In this section, we study the following evaluation problem of PCEA over streams:

Problem:	*EvalPCEA*$[\sigma]$
Input:	An unambiguous PCEA $\mathcal{P} = (Q, \mathbf{U}_{\text{lin}}, \mathbf{B}_{\text{eq}}, \Omega, \Delta, F)$ over σ, a window size $w \in \mathbb{N}$, and a stream $\mathcal{S} = t_0 t_1 \ldots$
Output:	At each position i, enumerate all valuations $\nu \in [\![\mathcal{P}]\!]_i(\mathcal{S})$ such that $\lvert i - \min(\nu) \rvert \leq w$.

The goal is to output the set $[\![\mathcal{P}]\!]_i^w(\mathcal{S}) = \{\nu \in [\![\mathcal{P}]\!]_i(\mathcal{S}) \mid |i - \min(\nu)| \leq w\}$ by reading the stream $\mathcal{S}$ tuple-by-tuple sequentially. We assume here a method `yield`$[\mathcal{S}]$ such that each call retrieves the next tuple, that is, the i-th call to `yield`$[\mathcal{S}]$ retrieves t_i for each $i \geq 0$.

For solving *EvalPCEA*$[\sigma]$, we desire to find a streaming evaluation algorithm [16, 18] that, for each tuple t_i, updates its internal state quickly and enumerates the set $[\![\mathcal{P}]\!]_i^w(\mathcal{S})$ with output-linear delay. More precisely, let $f : \mathbb{N}^3 \to \mathbb{N}$. A *streaming enumeration algorithm* $\mathcal{E}$ with f-update time for *EvalPCEA*$[\sigma]$ works as follows. Before reading the stream $\mathcal{S}$, $\mathcal{E}$ receives as input a PCEA $\mathcal{P}$ and $w \in \mathbb{N}$, and does some preprocessing. By calling `yield`$[\mathcal{S}]$, $\mathcal{E}$ reads $\mathcal{S}$ sequentially and processes the next tuple t_i in two phases called the *update phase* and *enumeration phase*, respectively. In the update phase, $\mathcal{E}$ updates a data structure DS with t_i taking time $\mathcal{O}(f(|\mathcal{P}|, |t_i|, w))$. In the enumeration phase, $\mathcal{E}$ uses DS for enumerating $[\![\mathcal{P}]\!]_i^w(\mathcal{S})$ with *output-linear delay*[2]. Formally, if $[\![\mathcal{P}]\!]_i^w(\mathcal{S}) = \{\nu_1, \ldots, \nu_k\}$ (i.e., in arbitrary order), the algorithm prints $\#\nu_1\#\nu_2\#\ldots\#\nu_k\#$ to the output registers, sequentially. Furthermore, $\mathcal{E}$ prints the first and last symbols # when the enumeration phase starts and ends, respectively, and the time difference (i.e., the *delay*) between printing the #-symbols surrounding ν_i is in $\mathcal{O}(|\nu_i|)$. Finally, if such an algorithm exists, we say that *EvalPCEA*$[\sigma]$ admits a *streaming evaluation algorithm* with f-update time and output-linear delay.

In the following, we prove the following algorithmic result for evaluating PCEA.

Theorem 5.1. *EvalPCEA*$[\sigma]$ *admits a streaming evaluation algorithm with* $(|\mathcal{P}| \cdot |t| + |\mathcal{P}| \cdot \log(|\mathcal{P}|) + |\mathcal{P}| \cdot \log(w))$*-update time and output-linear delay.*

Note that the update time does not depend on the number of outputs seen so far, and regarding data complexity (i.e., assuming that $\mathcal{P}$ and the size of the tuples, $|t|$ are fixed), the update time is logarithmic in the size of the sliding window. Theorem 5.1 improves with respect to [16] by considering a more general class of queries and evaluating over a sliding window. In contrast, Theorem 5.1 is incomparable to the algorithms for dynamic query evaluation of HCQ in [5, 18]. On the one hand, [5, 18] show constant update time algorithms for HCQ under insertions and deletions. On the other hand, Theorem 5.1 is for sliding windows (i.e., insertions "on the right" and deletions "on the left") and works for CER queries that include disjunction, iteration (i.e., loops), and can consider the order of tuples. If we restrict to HCQ, the algorithms in [5, 18] have better complexity, given that there is no need to maintain and check the order in which the tuples are inserted or deleted.

It is important to note that we base the algorithm of Theorem 5.1 on the ideas introduced in [16]. Nevertheless, it has several new insights that are novel and are not present in [16]. First, our algorithm evaluates PCEA, which is a generalization of CCEA, and then the approach in [16] requires several changes. Second, the data structure for our algorithm must manage the evaluation of a sliding window and simultaneously combine parallel runs into one. This challenge requires a new strategy for enumeration that combines cross-products with checking a time condition. Finally, maintaining the runs that are valid inside the sliding window with logarithmic update time requires the design of a new data structure based on the principles of a heap, which is novel. We believe this data structure is interesting in its own right, which could lead to new advances in streaming evaluation algorithms with enumeration.

We dedicate the rest of this section to explaining the streaming evaluation algorithm of Theorem 5.1, starting by describing the data structure DS.

The data structure. Fix a set of labels Ω. For representing sets of valuations $\nu : \Omega \to 2^{\mathbb{N}}$, we use a data structure composed of nodes, where each node stores a position, a set of labels, and pointers to

---

[2] For PCEA, output-linear delay is different from the notion of constant delay [26], even if we restrict to data complexity (i.e., if the input PCEA is fixed). Since a PCEA can have loops, an output could be of size proportional to the size of the data.

other nodes. Formally, the *data structure* DS is composed by a set of nodes, denoted by Nodes(DS), where each node n has a set $L(\mathsf{n}) \subseteq \Omega$, a position $i(\mathsf{n}) \in \mathbb{N}$, a set $\text{prod}(\mathsf{n}) \subseteq \text{Nodes}(\text{DS})$, and two links to other nodes $\text{uleft}(\mathsf{n}), \text{uright}(\mathsf{n}) \in \text{Nodes}(\text{DS})$. We assume that the directed graph G_{DS} with $V(G_{\text{DS}}) = \text{Nodes}(\text{DS})$ and $E(G_{\text{DS}}) = \{(\mathsf{n}_1, \mathsf{n}_2) \mid \mathsf{n}_2 \in \text{prod}(\mathsf{n}_1) \vee \mathsf{n}_2 = \text{uleft}(\mathsf{n}_1) \vee \mathsf{n}_2 = \text{uright}(\mathsf{n}_1)\}$ is acyclic. In addition, we assume a special node $\bot \in \text{Nodes}(\text{DS})$ that serves as a bottom node (i.e., all components above are undefined for $\bot$) and $\bot \notin \text{prod}(\mathsf{n})$ for every n.

Each node in DS represents a bag of valuations. To explain this representation, we need to first introduce some algebraic operations on valuations. Given two valuations $\nu, \nu' : \Omega \to 2^{\mathbb{N}}$, we define the product $\nu \oplus \nu' : \Omega \to 2^{\mathbb{N}}$ such that $[\nu \oplus \nu'](\ell) = \nu(\ell) \cup \nu'(\ell)$ for every $\ell \in \Omega$. Further, we extend this product to bags of valuations V and V' such that $V \oplus V' = \{\!\{\nu \oplus \nu' \mid \nu \in V, \nu' \in V'\}\!\}$. Note that $\oplus$ is an associative and commutative operation and, thus, we can write $\bigoplus_i V_i$ for referring to a sequence of $\oplus$-operations. Given a pair $(L, i) \in 2^{\Omega} \times \mathbb{N}$, we define the valuation $\nu_{L,i} : \Omega \to 2^{\mathbb{N}}$ such that $\nu_{L,i}(\ell) = \{i\}$ if $\ell \in L$, and $\nu_{L,i}(\ell) = \varnothing$, otherwise. With this notation, for every $\mathsf{n} \in \text{Nodes}(\text{DS})$ we define the bags $[\![\mathsf{n}]\!]_{\text{prod}}$ and $[\![\mathsf{n}]\!]$ recursively as follows:

$$[\![\mathsf{n}]\!]_{\text{prod}} := \{\!\{\nu_{L(\mathsf{n}),i(\mathsf{n})}\}\!\} \oplus \bigoplus_{\mathsf{n}' \in \text{prod}(\mathsf{n})} [\![\mathsf{n}']\!] \qquad [\![\mathsf{n}]\!] := [\![\mathsf{n}]\!]_{\text{prod}} \cup [\![\text{uleft}(\mathsf{n})]\!] \cup [\![\text{uright}(\mathsf{n})]\!].$$

For $\bot$, we define $[\![\bot]\!]_{\text{prod}} = [\![\bot]\!] = \varnothing$. Intuitively, the set $\text{prod}(\mathsf{n})$ represents the *product* of its nodes with the valuation $\nu_{L,i}$, and the nodes $\text{uleft}(\mathsf{n})$ and $\text{uright}(\mathsf{n})$ represent *unions* (for union-left and union-right, respectively). This interpretation is analog to the product and union nodes used in previous work of MSO enumeration [2, 22], but here we encode products and unions in a single node.

For efficiently enumerating $[\![\mathsf{n}]\!]$, we require that valuations in DS are represented without overlapping. To formalize this idea, define that the product $\nu \oplus \nu'$ is *simple* if for every $\ell \in \Omega$, $\nu(\ell)$ and $\nu'(\ell)$ are disjoint and $[\nu \oplus \nu'](\ell) = \nu(\ell) \cup \nu'(\ell)$. Accordingly, we extend this notion to bags of valuations: $V \oplus V'$ is simple if $\nu \oplus \nu'$ is simple for every $\nu \in V$ and $\nu' \in V'$. We say that DS is simple if $\{\!\{\nu_{L(\mathsf{n}),i(\mathsf{n})}\}\!\} \oplus \bigoplus_{\mathsf{n}' \in \text{prod}(\mathsf{n})} [\![\mathsf{n}']\!]$ is simple for every $\mathsf{n} \in \text{Nodes}(\text{DS})$. This notion is directly related to unambiguous PCEA in Section 3. Intuitively, the first condition of unambiguous PCEA will help us to force that DS is always simple.

The next step is to incorporate the window-size restriction to DS. For a node $\mathsf{n} \in \text{Nodes}(\text{DS})$, let $\max(\mathsf{n}) = \max\{i \in \nu(\ell) \mid \nu \in [\![\mathsf{n}]\!] \wedge \ell \in \Omega\}$. Then, given a position $i \geq \max(\mathsf{n})$ and a window size $w \in \mathbb{N}$, define the bag:

$$[\![\mathsf{n}]\!]_i^w \;:=\; \{\!\{\nu \in [\![\mathsf{n}]\!] \mid |i - \min(\nu)| \leq w\}\!\}.$$

We plan to represent $[\![\mathsf{n}]\!]_i^w$ and enumerate its valuations with output-linear delay. For this goal, from now on we fix a $w \in \mathbb{N}$ and write DS_w to denote the data structure with window size w. For the enumeration of $[\![\mathsf{n}]\!]_i^w$, in each node n we store the value:

$$\text{max-start}(\mathsf{n}) \;:=\; \max\left\{\min(\nu) \mid \nu \in [\![\mathsf{n}]\!]_{\text{prod}}\right\}$$

This value will be helpful to verify whether $[\![\mathsf{n}]\!]_i^w$ is non-empty or not; in particular, one can check that $[\![\mathsf{n}]\!]_i^w \neq \varnothing$ iff $|i - \text{max-start}(\mathsf{n})| \leq w$. We always assume that $|\max(\mathsf{n}) - \text{max-start}(\mathsf{n})| \leq w$ (otherwise $[\![\mathsf{n}]\!]_i^w = \varnothing$). In addition, we require an order with $\text{uleft}(\mathsf{n})$ and $\text{uright}(\mathsf{n})$ to discard empty unions easily. For every node $\mathsf{n} \in \text{Nodes}(\text{DS}_w)$, we require:

$$\text{max-start}(\mathsf{n}) \geq \text{max-start}(\text{uleft}(\mathsf{n})) \;\text{ and }\; \text{max-start}(\mathsf{n}) \geq \text{max-start}(\text{uright}(\mathsf{n})) \qquad (\ddagger)$$

whenever $\text{uleft}(\mathsf{n}) \neq \bot \neq \text{uright}(\mathsf{n})$. Intuitively, the binary tree formed by n and all nodes that can be reached by following $\text{uleft}(\cdot)$ and $\text{uright}(\cdot)$ is not strictly ordered; however, it follows the same principle ($\ddagger$) as a *heap* [10]. Note that it is not our goal to use DS_w as a priority queue (since removing the max element from a heap takes logarithmic time, and we need constant time), but to

Algorithm 1 Evaluation of an unambiguous PCEA $\mathcal{P} = (Q, \mathbf{U}_{\text{lin}}, \mathbf{B}_{\text{eq}}, \Omega, \Delta, F)$ with equality predicates over a stream $\mathcal{S}$ under a sliding window of size w.

```
procedure Evaluation($\mathcal{P}, w, \mathcal{S}$)
    $\mathsf{DS}_w \leftarrow \varnothing$
    $i \leftarrow -1$
    while $t \leftarrow \texttt{yield}[\mathcal{S}]$ do
        Reset()
        FireTransitions($t, i$)
        UpdateIndices($t, i$)
        for each $\mathsf{n} \in \bigcup_{p \in F} \mathsf{N}_p$
                $\wedge\ |\text{max-start}(\mathsf{n}) - i| \leq w$ do
            Enumerate($\mathsf{n}, i, w$)

procedure Reset()
    $i \leftarrow i + 1$
    for each $p \in Q$ do
        $\mathsf{N}_p \leftarrow \varnothing$

procedure FireTransitions($t, i$)
    for each $e = (P, U, \mathcal{B}, L, q) \in \Delta$ do
        if $t \in U \wedge \bigwedge_{p \in P} \mathsf{H}[e, p, \vec{\mathcal{B}}_p(t)] \neq \varnothing$ then
            $\mathsf{N} \leftarrow \{\, \mathsf{H}[e, p, \vec{\mathcal{B}}_p(t)] \mid p \in P \,\}$
            $\mathsf{N}_q \leftarrow \mathsf{N}_q \cup \{\texttt{extend}(L, i, \mathsf{N})\}$

procedure UpdateIndices($t$)
    for each $e = (P, U, \mathcal{B}, L, q) \in \Delta$ do
        for each $p \in P \wedge \mathsf{n} \in \mathsf{N}_p$ do
            if $\mathsf{H}[e, p, \overleftarrow{\mathcal{B}}_p(t)] = \varnothing$ then
                $\mathsf{H}[e, p, \overleftarrow{\mathcal{B}}_p(t)] \leftarrow \mathsf{n}$
            else
                $\mathsf{n}' \leftarrow \mathsf{H}[e, p, \overleftarrow{\mathcal{B}}_p(t)]$
                $\mathsf{H}[e, p, \overleftarrow{\mathcal{B}}_p(t)] \leftarrow \texttt{union}(\mathsf{n}', \mathsf{n})$
```

use condition (‡) to quickly check if there are more outputs to enumerate in $\text{uleft}(\mathsf{n})$ or $\text{uright}(\mathsf{n})$ by comparing the max-start value of a node with the start of the current location of the time window.

Theorem 5.2. *Let $w \in \mathbb{N}$ be a window size and assume that DS_w is simple. Then, for every $\mathsf{n} \in \text{Nodes}(\mathsf{DS}_w)$ and every position $i \geq \max(\mathsf{n})$, the valuations in $[\![\mathsf{n}]\!]_i^w$ can be enumerated with output-linear delay and without preprocessing (i.e., the enumeration starts immediately).*

We sketch the proof of this theorem in the appendix.

We require two procedures, called *extend* and *union*, for operating nodes in our algorithm. The first procedure $\texttt{extend}(L, i, \mathsf{N})$ receives as input a set $L \subseteq \Omega$, a position $i \in \mathbb{N}$, and $\mathsf{N} \subseteq \text{Nodes}(\mathsf{DS}_w)$ such that $i(\mathsf{n}) < i$ for every $\mathsf{n} \in \mathsf{N}$. The procedure outputs a fresh node n_e such that $[\![\mathsf{n}_e]\!]_i^w := \{\!\{\nu_{L,i}\}\!\} \oplus \bigoplus_{\mathsf{n} \in \mathsf{N}} [\![\mathsf{n}]\!]_i^w$. By the construction of DS_w, this operation is straightforward to implement by defining $L(\mathsf{n}_e) = L$, $i(\mathsf{n}_e) = i$, $\text{prod}(\mathsf{n}_e) = \mathsf{N}$, and $\text{uleft}(\mathsf{n}_e) = \text{uright}(\mathsf{n}_e) = \bot$. Further, we can compute $\text{max-start}(\mathsf{n}_e)$ from the set N as follows: $\text{max-start}(\mathsf{n}_e) = \min\{i, \min\{\text{max-start}(\mathsf{n}) \mid \mathsf{n} \in \mathsf{N}\}\}$. Overall, we can implement $\texttt{extend}(L, i, \mathsf{N})$ with running time $\mathcal{O}(|\mathsf{N}|)$.

The second procedure $\texttt{union}(\mathsf{n}_1, \mathsf{n}_2)$ receives as inputs two nodes $\mathsf{n}_1, \mathsf{n}_2 \in \text{Nodes}(\mathsf{DS}_w)$ such that $\max(\mathsf{n}_1) \leq i(\mathsf{n}_2)$ and $\text{uleft}(\mathsf{n}_2) = \text{uright}(\mathsf{n}_2) = \bot$. It outputs a fresh node n_u such that $[\![\mathsf{n}_u]\!]_i^w := [\![\mathsf{n}_1]\!]_i^w \cup [\![\mathsf{n}_2]\!]_i^w$. The implementation of this procedure is more involved since it requires inserting n_2 into n_1 by using $\text{uleft}(\mathsf{n}_1)$ and $\text{uright}(\mathsf{n}_1)$, and maintaining condition (‡). Furthermore, we require them to be fully persistent [13], namely, n_1 and n_2 are unmodified after each operation.

Proposition 5.3. *Let $k \in \mathbb{N}$ and assume that one performs $\texttt{union}(\mathsf{n}_1, \mathsf{n}_2)$ over DS_w with the same position $i = i(\mathsf{n}_2)$ at most k times. Then one can implement $\texttt{union}(\mathsf{n}_1, \mathsf{n}_2)$ with running time $\mathcal{O}(\log(k \cdot w))$ per call.*

In the appendix, we provide a proof of this proposition. An illustration on how this data structure works is included in Example 5.4.

The streaming evaluation algorithm. In Algorithm 1, we present the main procedures of the evaluation algorithm given a fixed schema σ. The algorithm receives as input a PCEA $\mathcal{P} = (Q, \mathbf{U}_{\text{lin}}, \mathbf{B}_{\text{eq}}, \Omega, \Delta, F)$ over σ, a window size $w \in \mathbb{N}$, and a reference to a stream $\mathcal{S}$. We assume that these inputs are globally accessible by all procedures. Recall that we can test if $t \in U$ in linear time for any $U \in \mathbf{U}_{\text{lin}}$. Further, recall that $\mathbf{B}_{\text{eq}}$ are equality predicates and, for every $B \in \mathbf{B}_{\text{eq}}$, there exists linear time computable partial functions $\overleftarrow{B}$ and $\vec{B}$ such that $(t_1, t_2) \in B$ iff $\overleftarrow{B}(t_1)$ and $\vec{B}(t_2)$ are defined and $\overleftarrow{B}(t_1) = \vec{B}(t_2)$, for every $t_1, t_2 \in \text{Tuples}[\sigma]$.

For the algorithm, we require some data structures. First, we use the previously described data structure DS_w and its nodes $\text{Nodes}(\mathsf{DS}_w)$. Second, we consider a *look-up table* H that maps triples of the form (e, p, d) to nodes in $\text{Nodes}(\mathsf{DS}_w)$ where $e \in \Delta$, $p \in Q$, and d is the output of any partial function $\overleftarrow{B}$ or $\vec{B}$. We write $\mathsf{H}[e, p, d]$ for accessing its node, and $\mathsf{H}[e, p, d] \leftarrow \mathsf{n}$ for updating a node n at entry (e, p, d). Also, we write $\mathsf{H}[e, p, d] = \emptyset$ or $\mathsf{H}[e, p, d] \neq \emptyset$ for checking whether there is a node or not at entry (e, p, d). We assume all entries are empty at the beginning. Intuitively, for $e = (P, U, \mathcal{B}, L, q) \in \Delta$ and $p \in P$, we use $\mathsf{H}[e, p, \cdot]$ to check if the equality predicate $\mathcal{B}_p$ is satisfied or not (here $\mathcal{B}_p = \mathcal{B}(p)$). As it is standard in the literature [5, 18] (i.e., by adopting the RAM model), we assume that each operation over look-up tables takes constant time. Finally, we assume a set of nodes N_p for each $p \in Q$ whose use will be clear later.

Algorithm 1 starts at the main procedure Evaluation. It initializes the data structure DS_w to empty (i.e., the only node it has is the special node $\bot$) and the index i for keeping the current position in the stream (lines 2-3). Then, the algorithm loops by reading the next tuple $\texttt{yield}[\mathcal{S}]$, performs the update phase (lines 5-7), followed by the enumeration phase (lines 8-9), and repeats the process over again. Next, we explain the update phase and enumeration phase separately.

The update phase is composed of three steps, encoded as procedures. The first one, Reset, is in charge of starting a new iteration by updating i to the next position and emptying the sets N_p (lines 12-14). The second step, FireTransitions, uses the new tuple t to fire all transitions $e = (P, U, \mathcal{B}, L, q) \in \Delta$ of $\mathcal{P}$ (lines 16-19). We do this by checking if t satisfies U and all equality predicates $\{\mathcal{B}_p\}_{p \in P}$ (line 17). The main intuition is that the algorithm stores partial runs in the look-up table H, whose outputs are represented by nodes in DS_w. Then the call $\mathsf{H}[e, p, \vec{\mathcal{B}}_p(t)]$ is used to verify the equality $\overleftarrow{\mathcal{B}}_p(t') = \vec{\mathcal{B}}_p(t)$ for some previous tuple t'. Furthermore, if $\mathsf{H}[e, p, \vec{\mathcal{B}}_p(t)]$ is non-empty, it contains the node that represents all runs that have reached p. If U and all predicates $\{\mathcal{B}_p\}_{p \in P}$ are satisfied, we collect all nodes at states P in the set N (line 18), and symbolically extend these runs by using the method $\texttt{extend}(L, i, \mathsf{N})$ of DS_w. We collect the output node of $\texttt{extend}$ in the set N_q for use in the next procedure UpdateIndices.

The last step of the update phase, UpdateIndices, is to update the look-up table H by using t and the nodes stored at the sets $\{\mathsf{N}_p\}_{p \in Q}$ (lines 22-28). Intuitively, the nodes in N_p represent new runs (i.e., valuations) that reached state p when reading t. Then, for every transition $e = (P, U, \mathcal{B}, L, q) \in \Delta$ such that $p \in P$, we want to update the entry $(e, p, \overleftarrow{\mathcal{B}}_p(t))$ of H with the nodes from N_p, to be ready to be fired for future tuples. For this goal, we check each $\mathsf{n} \in \mathsf{N}_p$ and, if $\mathsf{H}[e, p, \overleftarrow{\mathcal{B}}_p(t)]$ is empty, we just place n at the entry $(e, p, \overleftarrow{\mathcal{B}}_p(t))$ (lines 23-25). Otherwise, we use the union operator of DS_w, to combine the previous outputs with the new ones of n (lines 26-28). Note that the call to $\texttt{union}(\mathsf{n}', \mathsf{n})$ satisfies the requirements of this operator, given that n was created recently.

Example 5.4. For getting some intuition on how the data structure DS_w and Algorithm 1 work, we show an example involving the $\texttt{extend}$ and the $\texttt{union}$ procedures. Consider the PCEA $\mathcal{P}_0$ from Figure 1 and the stream:

$$\underbrace{T(0)}_{0}\ \underbrace{S(0,0)}_{1}\ \underbrace{T(0)}_{2}\ \underbrace{R(0,0)}_{3}\ \ldots$$

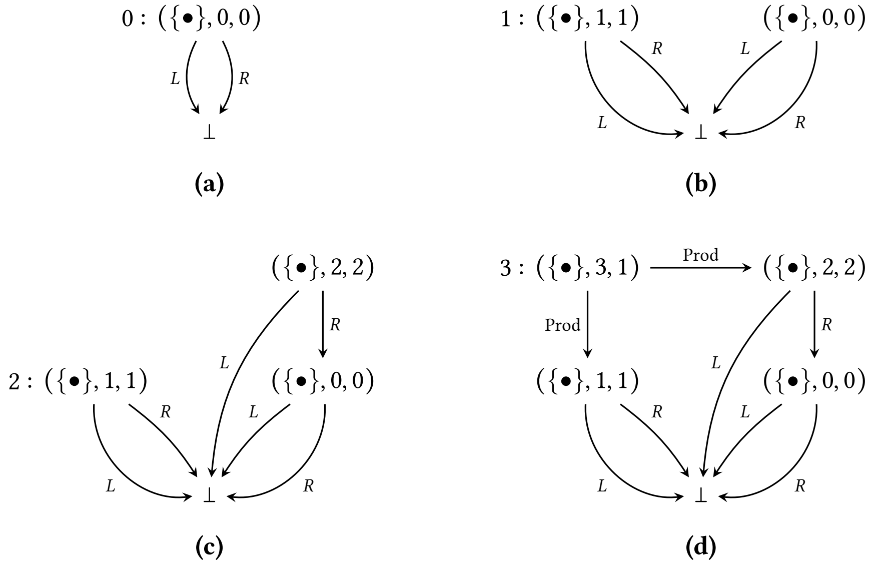


Fig. 3. (a), (b), (c), and (d) show the resulting data structure DS_w following Algorithm 1 after reading the tuples $T(0)$, $S(0,0)$, $T(0)$, and $R(0,0)$, respectively. We represent each node by a triple $(L(\mathsf{n}), i(\mathsf{n}), \text{max-start}(\mathsf{n}))$, with $\text{uleft}(\mathsf{n})$, $\text{uright}(\mathsf{n})$, and $\text{prod}(\mathsf{n})$ represented by labeled edges L, R and Prod, respectively.

In Figure 3, we show how the data structure DS_w with $w = 4$ evolves after reading the stream until position 3. To represent nodes we use tuples of the form $\mathsf{n} = (L(\mathsf{n}), i(\mathsf{n}), \text{max-start}(\mathsf{n}))$, with $\text{uleft}(\mathsf{n})$, $\text{uright}(\mathsf{n})$, and $\text{prod}(\mathsf{n})$ represented by labeled edges L, R and Prod on the figure, respectively. We label each new node with the position of the tuple that adds it.

Based on the previous description, the enumeration phase is straightforward. Given that the nodes in $\{\mathsf{N}_p\}_{p \in Q}$ represent new runs at the last position, $\bigcup_{p \in F} \mathsf{N}_p$ are all new runs that reached some final state. Then, for each node $\mathsf{n} \in \bigcup_{p \in F} \mathsf{N}_p$ satisfying $|\text{max-start}(\mathsf{n}) - i| \leq w$ we call the procedure $\textsc{Enumerate}(\mathsf{n}, i, w)$ that enumerates all valuations in $[\![\mathsf{n}]\!]_i^w$. Theorem 5.2 shows that this method exists with the desired guarantees given that $\mathcal{P}$ is unambiguous which implies that DS_w is simple. Note that, for enforcing output-linear delay, we assume that the **for each** routine is done wisely by removing the nodes $\mathsf{n} \in \bigcup_{p \in F} \mathsf{N}_p$ that does not satisfy $|\text{max-start}(\mathsf{n}) - i| \leq w$ before starting the enumeration phase. Further, runs correspond with valuations, namely, $[\![\mathsf{n}]\!]_i^w$ is a set, and, thus, we enumerate the outputs without repetitions.

Proposition 5.5. *For every unambiguous PCEA $\mathcal{P}$ with equality predicates, $w \in \mathbb{N}$, stream $\mathcal{S}$, and position $i \in \mathbb{N}$, Algorithm 1 enumerates all valuations $[\![\mathcal{P}]\!]_i^w(\mathcal{S})$ without repetitions.*

We end by discussing the update time of Algorithm 1. By inspection, one can check that we performed a linear pass over Δ during the update phase, where each iteration takes linear time over each transition. Overall, we made at most $\mathcal{O}(|\mathcal{P}|)$ calls to unary predicates, the look-up table, or the data structure DS_w. Each call to a unary predicate takes $\mathcal{O}(|t|)$-time and, thus, at most $\mathcal{O}(|\mathcal{P}| \cdot |t|)$-time in total. The operations to the look-up table or `extend` take constant time. Instead, we performed at most $\mathcal{O}(|\mathcal{P}|)$ unions over the same position i. By Proposition 5.3, each `union` takes time $\mathcal{O}(\log(|\mathcal{P}| \cdot w))$. Summing up, the updating time is $\mathcal{O}(|\mathcal{P}| \cdot |t| + |\mathcal{P}| \cdot \log(|\mathcal{P}|) + |\mathcal{P}| \cdot \log(w))$.

6 Future work

We present an automata model for CER that expresses HCQ and can be evaluated in a streaming fashion under a sliding window with a logarithmic update time and output-linear delay. These results achieve the primary goal of this paper but leave several directions for future work. First, defining a query language that characterizes the expressive power of PCEA will be interesting. Second, one would like to understand a disambiguation procedure to convert any PCEA into an unambiguous PCEA or to decide when this is possible. Last, we study here algorithms for PCEA with equality predicates, but the model works for any binary predicate. Then, it would be interesting to understand for which other predicates (e.g., inequalities) the model still admits efficient streaming evaluation. On this line, an interesting problem is to study how to extend the algorithms for PCEA to include deletions everywhere in the stream.

Acknowledgments

This work was funded by ANID Fondecyt Regular project 1230935 and ANID – Millennium Science Initiative Program – Code ICN17_002.

References

[1] A. V. Aho and J. E. Hopcroft. *The design and analysis of computer algorithms.* Addison-Wesley, 1974.

[2] A. Amarilli, P. Bourhis, L. Jachiet, and S. Mengel. A circuit-based approach to efficient enumeration. In *ICALP*, volume 80 of *LIPIcs*, pages 111:1–111:15, 2017.

[3] A. Amarilli, L. Jachiet, M. Muñoz, and C. Riveros. Efficient enumeration for annotated grammars. In *PODS*, pages 291–300. ACM, 2022.

[4] A. Artikis, A. Margara, M. Ugarte, S. Vansummeren, and M. Weidlich. Complex event recognition languages: Tutorial. In *DEBS*, pages 7–10. ACM, 2017.

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

[6] M. Bucchi, A. Grez, A. Quintana, C. Riveros, and S. Vansummeren. CORE: a complex event recognition engine. *VLDB*, 15(9):1951–1964, 2022.

[7] A. K. Chandra, D. C. Kozen, and L. J. Stockmeyer. Alternation. *Journal of the ACM (JACM)*, 28(1):114–133, 1981.

[8] S. Chaudhuri and M. Y. Vardi. Optimization of real conjunctive queries. In *PODS*, pages 59–70, 1993.

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

[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms.* MIT press, 2022.

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

[12] N. N. Dalvi and D. Suciu. The dichotomy of conjunctive queries on probabilistic structures. In *PODS*, pages 293–302, 2007.

[13] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. In *STOC*, pages 109–121, 1986.

[14] N. Giatrakos, E. Alevizos, A. Artikis, A. Deligiannakis, and M. N. Garofalakis. Complex event recognition in the big data era: a survey. *VLDB J.*, 29(1):313–352, 2020.

[15] E. Grandjean and L. Jachiet. Which arithmetic operations can be performed in constant time in the RAM model with addition? *CoRR*, abs/2206.13851, 2022.

[16] A. Grez and C. Riveros. Towards Streaming Evaluation of Queries with Correlation in Complex Event Processing. In *ICDT*, volume 155, pages 14:1–14:17, 2020.

[17] A. Grez, C. Riveros, M. Ugarte, and S. Vansummeren. A formal framework for complex event recognition. *ACM Trans. Database Syst.*, 46(4):16:1–16:49, 2021.

[18] M. Idris, M. Ugarte, and S. Vansummeren. The dynamic yannakakis algorithm: Compact and efficient query processing under updates. In *SIGMOD*, pages 1259–1274, 2017.

[19] M. Idris, M. Ugarte, S. Vansummeren, H. Voigt, and W. Lehner. General dynamic yannakakis: conjunctive queries with theta joins under updates. *VLDB J.*, 29(2-3):619–653, 2020.

[20] A. Kara, M. Nikolic, D. Olteanu, and H. Zhang. Pods. pages 375–392. ACM, 2020.

[21] Q. Lin, B. C. Ooi, Z. Wang, and C. Yu. Scalable distributed stream join processing. In *SIGMOD*, pages 811–825, 2015.

[22] M. Muñoz and C. Riveros. Streaming enumeration on nested documents. In *ICDT*, volume 220 of *LIPIcs*, pages 19:1–19:18, 2022.

[23] M. Muñoz and C. Riveros. Constant-delay enumeration for slp-compressed documents. In *ICDT*, volume 255 of *LIPIcs*, pages 7:1–7:17, 2023.

[24] F. Neven. Automata theory for XML researchers. *SIGMOD Record*, 31(3):39–46, 2002.

[25] D. Pinto and C. Riveros. Complex event recognition meets hierarchical conjunctive queries. *arXiv preprint arXiv:2408.01652*, 2024.

[26] L. Segoufin. Constant delay enumeration for conjunctive queries. *ACM SIGMOD Record*, 44(1):10–17, 2015.

[27] P. D. Stotts and W. W. Pugh. Parallel finite automata for modeling concurrent software systems. *J. Syst. Softw.*, 27(1):27–43, 1994.

[28] N. Tziavelis, W. Gatterbauer, and M. Riedewald. Beyond equi-joins: Ranking, enumeration and factorization. *VLDB*, 14(11):2599–2612, 2021.

[29] M. Ugarte and S. Vansummeren. On the difference between complex event processing and dynamic query evaluation. In *AMW*, volume 2100, 2018.

[30] Q. Wang, X. Hu, B. Dai, and K. Yi. Change propagation without joins. *VLDB*, 16(5):1046–1058, 2023.

[31] Q. Wang and K. Yi. Conjunctive queries with comparisons. In *SIGMOD*, pages 108–121. ACM, 2022.

[32] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *SIGMOD*, pages 407–418, 2006.

[33] J. Xie and J. Yang. A survey of join processing in data streams. *Data Streams: Models and Algorithms*, pages 209–236, 2007.

A Proofs of Section 3

Proof of Proposition 3.2

Proof. To prove this statement, we follow the same principle used in the subset construction. To simulate all possible run trees of a PFA with a DFA, we start at the leaves, with all initial states. Then for each symbol we move up on the tree, firing all transitions that used a subset of the current set of states. At the end of the string, if the last set has a final state, then it means that one can construct a run tree that accepts the input.

Let $\mathcal{P} = (Q, \Sigma, \Delta, I, F)$ be a parallelized finite automata. We build the DFA $\mathcal{A} = (2^Q, \Sigma, \delta, I, F')$ such that $F' = \{P \mid P \cap F \neq \varnothing\}$ and $\delta(P, a) = \{q \mid \exists P' \subseteq P. (P', a, q) \in \Delta)\}$ for every $P \subseteq Q$ and $a \in \Sigma$. We now prove that both automata define the same language.

$\mathcal{L}(\mathcal{P}) \subseteq \mathcal{L}(\mathcal{A})$. Let $\bar{s} = a_1 \dots a_n \in \Sigma^*$ be a string such that $\bar{s} \in \mathcal{L}(\mathcal{P})$ and let $\tau : t \to Q$ be an accepting run tree of $\mathcal{P}$ over $\bar{s}$. We need to prove that the run $\rho : S_n \xrightarrow{a_1} S_{n-1} \xrightarrow{a_2} \dots \xrightarrow{a_n} S_0$ is an accepting run of $\mathcal{A}$ over $\bar{s}$, i.e. $S_n \in F'$. To this end, we define $L_i = \{\tau(\bar{u}) \mid \mathtt{depth}_\tau(\bar{u}) = i\}$ as the set of states labeling τ at depth i and prove that $L_i \subseteq S_i$ for all $0 \leq i \leq n$. Since $L_0 = \{\tau(\varepsilon)\}$, this in return means that $S_n \cap F \neq \varnothing$ and $S_n \in F'$.

For every leaf node $\bar{u}$ it holds that $\mathtt{depth}_\tau(\bar{u}) = n$ and $\tau(\bar{u}) \in I$, meaning $L_n \subseteq S_n = I$. Let us assume that $L_{i-1} \subseteq S_{i-1}$; for every inner node $\bar{v}$ at depth i there must be a transition $(P, a_{n-i}, q) \in \Delta$ such that $\tau(\bar{v}) = q$ and $P = \{\tau(\bar{u}) \mid \bar{u} \in \mathtt{children}_\tau(\bar{v})\}$. Following the definition of δ, it is clear that $q \in \delta(P, a)$, and since this is true for every node at depth i, we have that $L_i \subseteq S_i$.

Given that $L_0 \subseteq S_0$, we know that $S_n \in F'$, which means that ρ is an accepting run of $\mathcal{A}$ over $\bar{s}$ and therefore $\mathcal{L}(\mathcal{P}) \subseteq \mathcal{L}(\mathcal{A})$.

$\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{P})$. Let $\bar{s} = a_1 \dots a_n \in \Sigma^*$ be a string such that $\bar{s} \in \mathcal{L}(\mathcal{A})$ and let $\rho : S_n \xrightarrow{a_1} S_{n-1} \xrightarrow{a_2} \dots \xrightarrow{a_n} S_0$ be the run of $\mathcal{A}$ over $\bar{s}$. We can now construct a run tree of $\mathcal{P}$ over $\bar{s}$.

Since ρ is an accepting run, we know that $S_0 \cap F \leq \varnothing$. We define $\tau : t \to Q$ such that $\tau(\varepsilon) = f$ with $f \in S_0 \cap F$. If we consider a node $\bar{v} \in t$ at depth i, such that $\tau(\bar{v}) = q$ and $q \in S_i$, we can follow the definition of δ, and inductively add nodes to τ according to the transition $(P, a_{n-i}, q) \in \Delta$ so that $|\mathtt{children}_\tau(\bar{v})| = |P|$ and $P = \{\tau(\bar{u}) \mid \bar{u} \in \mathtt{children}_\tau(\bar{v})\}$. For every leaf node $\bar{v}$ it holds that $\mathtt{depth}_\tau(\bar{v}) = n$ and since $S_n = I$ all of them will be labeled by initial states.

The labeled tree τ we just constructed is an accepting run of $\mathcal{P}$ over $\bar{s}$, meaning $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{P})$ and, therefore, $\mathcal{L}(\mathcal{P}) = \mathcal{L}(\mathcal{A})$. □

Proof of Proposition 3.4

Proof. To prove this statement we just need to find a Parallelized-CEA $\mathcal{P}$ with no CCEA equivalent, i.e. there is no CCEA $\mathcal{C}$ such that $[\![\mathcal{P}]\!](\mathcal{S}) = [\![\mathcal{C}]\!](\mathcal{S})$ for every stream $\mathcal{S}$. Let $\mathcal{P}$ be the PCEA represented in Figure 2, then $\mathcal{P} = \big(Q, \mathbf{U}, \mathbf{B}, \Omega, \Delta, F\big)$, with $Q = \{R(x,y), S(x,y), T(x), x, y\}$, $\Omega = \{R, S, T\}$, $F = \{x\}$ and:

$$\begin{array}{rcl}
\Delta & = & \big\{(\varnothing, U_{R(x,y)}, \varnothing, \{R(x,y)\}, R(x,y)), \\
& & (\varnothing, U_{S(x,y)}, \varnothing, \{S(x,y)\}, S(x,y)), \\
& & (\varnothing, U_{T(x)}, \varnothing, \{T(x)\}, T(x)), \\
& & (\{R(x,y), T(x)\}, U_{S(x,y)}, \{(R(x,y), B_{R(x,y),S(x,y)}), (T(x), B_{T(x),S(x,y)})\}, \{S(x,y)\}, x), \\
& & (\{S(x,y), T(x)\}, U_{R(x,y)}, \{(S(x,y), B_{S(x,y),R(x,y)}), (T(x), B_{T(x),R(x,y)})\}, \{R(x,y)\}, x), \\
& & (\{R(x,y)\}, U_{S(x,y)}, \{(R(x,y), B_{R(x,y),S(x,y)})\}, \{S(x,y)\}, y), \\
& & (\{S(x,y)\}, U_{R(x,y)}, \{(S(x,y), B_{S(x,y),R(x,y)})\}, \{R(x,y)\}, y), \\
& & (\{y\}, U_{T(x)}, \{(y, B_{y,T(x,y)})\}, \{T(x)\}, x)\big\}
\end{array}$$

with the predicates $U_{R(\bar{x})}$ and $B_{R(\bar{x}),S(\bar{y})}$ defined as:

$$U_{R(\bar{x})} \;:=\; \{R(\bar{a}) \in \text{Tuples}[\sigma] \mid \exists h \in \text{Hom}\,.\, h(R(\bar{x})) = R(\bar{a})\}$$

and:

$$B_{R(\bar{x}),S(\bar{y})} \;:=\; \{(R(\bar{a}), S(\bar{b})) \mid \exists h \in \text{Hom}\,.\, h(R(\bar{x})) = R(\bar{a}) \wedge h(S(\bar{y})) = S(\bar{b})\}.$$

Let $\mathcal{S}_i = \{\!\{R(0,i), T(0), S(0,i), \ldots\}\!\}$ be a family of streams over the set of data values $\mathbf{D} = \mathbb{N}$ with $i \in \mathbb{N}$. It is clear that the valuation $\{\!\{0,1,2\}\!\} \in [\![\mathcal{P}]\!](\mathcal{S}_i)$ for every $i \in \mathbb{N}$. Let $\mathcal{C} = (Q', \mathbf{U}', \mathbf{B}', \Omega', \Delta', I', F')$ be a deterministic CCEA such that $[\![\mathcal{C}]\!](\mathcal{S}_i) = [\![\mathcal{P}]\!](\mathcal{S}_i)$ for every $i \in \mathbb{N}$. This means that for every stream $\mathcal{S}_i$, there is an accepting run of $\mathcal{C}$ over of the form $\rho_i : q_{i,0} \xrightarrow{R(0,i)} q_{i,1} \xrightarrow{T(0)} q_{i,2} \xrightarrow{S(0,i)} q_{i,3}$.

Since $\mathcal{C}$ has a finite number of states, we know that there must be two streams, $\mathcal{S}_j$ and $\mathcal{S}_k$ with $j \neq k$ with accepting runs $\rho_j : q_{j,0} \xrightarrow{R(0,j)} q_{j,1} \xrightarrow{T(0)} q_{j,2} \xrightarrow{S(0,j)} q_{j,3}$ and $\rho_k : q_{k,0} \xrightarrow{R(0,k)} q_{k,1} \xrightarrow{T(0)} q_{k,2} \xrightarrow{S(0,k)} q_{k,3}$, respectively, such that $q_{j,i} = q_{k,i}$ for every $0 \leq i \leq 3$.

Given the run ρ_k of $\mathcal{C}$, we know that there must be a transition $(q_{k,2}, U, B, \omega, q_{k,3}) \in \Delta'$ such that $S(0,k) \in U$ and $(T(0), S(0,k)) \in B$ and since $q_{k,2} = q_{j,2}$ and $q_{k,3} = q_{j,3}$ the following will be an accepting run of $\mathcal{C}$ over the stream $S_{j,k} = \{\{R(0,j), T(0), S(0,k)\}\}$: $\rho_{j,k} : q_{j,0} \xrightarrow{R(0,j)} q_{j,1} \xrightarrow{T(0)} q_{j,2} \xrightarrow{S(0,k)} q_{j,3}$.

We can easily check that there are no accepting runs of $\mathcal{P}$ over $\mathcal{S}_{j,k}$, meaning $[\![\mathcal{P}]\!](\mathcal{S}_{j,k}) \neq [\![\mathcal{C}]\!](\mathcal{S}_{j,k})$ and therefore there is no CCEA $\mathcal{C}$ such that $[\![\mathcal{P}]\!](\mathcal{S}) = [\![\mathcal{C}]\!](\mathcal{S})$ for every stream $\mathcal{S}$. □

B Proofs of Section 4

Proof of equivalence between CQ bag-semantics

Fix a schema σ, a relational database D over σ, and a CQ Q over σ of the form:

$$Q(\bar{x}) \;\leftarrow\; R_0(\bar{x}_0), \ldots, R_{m-1}(\bar{x}_{m-1}).$$

Further, without loss of generality, assume that $\mathbf{X} = \bigcup_i \{\bar{x}_i\}$ and, thus, all homomorphisms $h : \mathbf{X} \cup \mathbf{D} \to \mathbf{D}$ in $\text{Hom}(Q, D)$ are restricted to the variables of Q. To prove that $[\![Q]\!](D) = [Q](D)$, we need to prove that both bags have equal multiplicity, namely, $\text{mult}_{[\![Q]\!](D)}(Q(\bar{a})) = \text{mult}_{[Q](D)}(Q(\bar{a}))$ for every Q-tuple $Q(\bar{a})$.

Towards this goal, let t-Hom(Q, D) be the set of all t-homomorphism from Q to D. Recall that for every t-homomorphism $\eta \in$ t-Hom(Q, D) there exists a unique homomorphism $h_\eta \in \text{Hom}(Q, D)$ associated to η. Conversely, one can easily see that, for every $h \in \text{Hom}(Q, D)$, there are exactly:

$$\text{mult}_{Q,D}(h) \;=\; \prod_{i=0}^{m-1} \text{mult}_D(R_i(h(\bar{x}_i)))$$

t-homomorphisms η such that $h = h_\eta$. Indeed, for having $h = h_\eta$ we must map each $i \in I(Q)$ (i.e., $R_i(\bar{x}_i)$) to a tuple $j \in I(D)$ of the form $R_i(h(\bar{x}_i))$ and we only have $\text{mult}_D(R_i(h(\bar{x}_i)))$ copies of it. Then, for each $i \in I(Q)$ we can choose $\eta(i)$ in $\text{mult}_D(h(R_i(\bar{x}_i)))$ possible ways, independently. In other words, for every $h \in \text{Hom}(Q, D)$ it holds that:

$$|\{\eta \in \text{t-Hom}(Q, D) \mid h = h_\eta\}| \;=\; \text{mult}_{Q,D}(h).$$

Using the previous equation, for any Q-tuple $Q(\bar{a})$ we can show that $\text{mult}_{[\![Q]\!](D)}(Q(\bar{a})) = \text{mult}_{[Q](D)}(Q(\bar{a}))$. By following the definition for the multiplicities of a bag $[\![Q]\!](D)$, we know that:

$$\text{mult}_{[\![Q]\!](D)}(Q(\bar{a})) = \left|\{j \mid [\![Q]\!](D)(j) = Q(\bar{a})\}\right|.$$

We also know that, if $[\![Q]\!](D)(j) = Q(\bar{a})$ holds, then there must be a unique t-homomorphism η for j such that $h_\eta(\bar{x}) = \bar{a}$ (i.e., there is a one-to-one correspondence between identifiers in $I([\![Q]\!](D))$ and t-homomorphisms in $\text{t-Hom}(Q, D)$). Then the following equivalences follows:

$$\begin{aligned}
\mathrm{mult}_{[\![Q]\!](D)}(Q(\bar{a})) &= \left|\{j \mid [\![Q]\!](D)(j) = Q(\bar{a})\}\right| \\
&= |\{\eta \in \text{t-Hom}(Q, D) \mid h_\eta(\bar{x}) = \bar{a}\}| \\
&= \sum_{\substack{h \in \mathrm{Hom}(Q,D): \\ h(\bar{x}) = \bar{a}}} |\{\eta \in \text{t-Hom}(Q, D) \mid h = h_\eta\}| \\
&= \sum_{\substack{h \in \mathrm{Hom}(Q,D): \\ h(\bar{x}) = \bar{a}}} \mathrm{mult}_{Q,D}(h) \\
&= \mathrm{mult}_{[Q](D)}(Q(\bar{a})).
\end{aligned}$$

□

Proof of Theorem 4.2

We prove that, if Q is non-hierarchical, then $\mathcal{P} \not\equiv Q$ for every PCEA $\mathcal{P}$ over σ. So, assume that Q is a full non-hierarchical CQ over σ. To better explain the proof, we will start with a specific case and then explain how to extend it to every conjunctive query.

Fix a schema σ and, for the sake of simplification, assume that $\mathbf{D} = \mathbb{N}$. Let $Q(x, y)$ be the following full CQ:

$$Q(x, y) \ \leftarrow \ R_0(x, y), R_1(x), R_2(y)$$

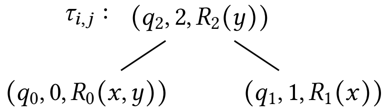
Note that Q is not a hierarchical query, since $atoms(x) \nsubseteq atoms(y)$, $atoms(y) \nsubseteq atoms(x)$, and $atoms(y) \cap atoms(z) \neq \emptyset$.

By contradiction, suppose that there exists PCEA $\mathcal{P} = (P, \mathbf{U}, \mathbf{B}, \Omega, \Delta, F)$ such that $\mathcal{P} \equiv Q$. For every $i, j \in \mathbb{N}$ we define the stream $\mathcal{S}_{i,j}$ such that $\mathcal{S}_{i,j} = R_0(i, j), R_1(i), R_2(j), \ldots$. Note that after the third tuple, the tuples of $\mathcal{S}_{i,j}$ are not relevant. It is easy to see that the valuation:

$$\nu \ = \ \{R_0(x, y) \to 0, R_1(x) \to 1, R_2(y) \to 2\}$$

satisfies that $\nu \in [\![Q]\!]_2(\mathcal{S}_{i,j})$ and, then, $\nu \in [\![\mathcal{P}]\!]_2(\mathcal{S}_{i,j})$ for every $i, j \in \mathbb{N}$. Let $\tau_{i,j}$ be the run of $\mathcal{P}$ over $\mathcal{S}_{i,j}$ that ends at position 2 and produces ν. Clearly, each $\tau_{i,j}$ has three nodes and it is either of the following two shapes:

$$\tau_{i,j} : (q_0, 0, R_0(x, y)) - (q_1, 1, R_1(x)) — (q_2, 2, R_2(y))$$



for some states $q_0, q_1, q_2 \in P$. We call the former a *line* shape and the latter a *tree* shape. Note that although some runs $\tau_{i,j}$ may coincide in the shape, they can use different states or transitions of $\mathcal{P}$. However, given that $\mathcal{P}$ is finite, there are a finite number of such runs. Then, let N be the number of different runs $\tau_{i,j}$ of $\mathcal{P}$.

For every $i \leq (N + 1) \cdot N$, there must exists $j_i, j'_i \leq N$ such that $j_i \neq j'_i$ and τ_{i,j_i} and τ_{i,j'_i} are equivalent, namely, they have the same shape, states, and transitions. We need to consider two possible scenarios.

(1) Assume that for some $i \leq (N + 1) \cdot N$, both τ_{i,j_i} and τ_{i,j'_i} have the shape of a line like above (left). Then, it is easy to see that $\mathcal{P}$ would have the same run tree for the stream:

$$\mathcal{S}^* \ = \ R_0(i, j_i), R_1(i), R_2(j'_i), \ldots$$

This implies that $\nu \in [\![\mathcal{P}]\!]_2(\mathcal{S}^*)$, however, $\nu \notin [\![Q]\!]_2(\mathcal{S}^*)$ which is a contradiction.

(2) Otherwise, assume that, for every $i \leq (N+1)\cdot N$, τ_{i,j_i} and τ_{i,j'_i} have not the shape of a line, namely, they are tree shape. By the pigeonhole principle, there must exists $i_0 < i_1 < \ldots < i_N \leq (N+1)\cdot N$ such that $j_{i_0} = j_{i_1} = \ldots = j_{i_N} = j^*$. Therefore, all the runs $\tau_{i_0,j^*}, \ldots, \tau_{i_N,j^*}$ are tree shape. Applying again the pigeonhole principle, we know that there must exist $k, \ell \leq N$ with $k \neq \ell$ such that τ_{i_k,j^*} and τ_{i_ℓ,j^*} are equivalent and have tree shape like above (right). Then, it is easy to see that $\mathcal{P}$ would have the same run tree for stream:

$$\mathcal{S}^* \ = \ R_0(i_k, j^*), R_1(i_\ell), R_2(j^*), \ldots$$

Again, this implies that $\nu \in [\![\mathcal{P}]\!]_2(\mathcal{S}^*)$, however, $\nu \notin [\![Q]\!]_2(\mathcal{S}^*)$ which is a contradiction.

Given that in both scenarios we found a stream $\mathcal{S}^*$ where $[\![\mathcal{P}]\!]_2(\mathcal{S}^*) \neq [\![Q]\!]_2(\mathcal{S}^*)$, we conclude that $\mathcal{P} \not\equiv Q$ for every PCEA $\mathcal{P}$ over σ.

For the general case, we consider any full CQ Q of the form:

$$Q(\bar{x}) \ \leftarrow \ R_0(\bar{x}_0), \ldots, R_{m-1}(\bar{x}_{m-1})$$

that is non-hierarchical, meaning there is a pair of variables $x, y \in \mathbf{X}$ such that $atoms(x) \nsubseteq atoms(y)$, $atoms(y) \nsubseteq atoms(x)$, and $atoms(x) \cap atoms(y) \neq \emptyset$. For every $i < m$, we say that $R_i(\bar{x}_i)$ is an *x-atom* if $x \in \{\bar{x}newc_i\}$ and $y \notin \{\bar{x}_i\}$; an *y-atom* if $x \notin \{\bar{x}_i\}$ and $y \in \{\bar{x}_i\}$; an *xy-atom* if $x, y \in \{\bar{x}_i\}$; and an $\emptyset$-atom if $\{x, y\} \cap \{\bar{x}_i\} = \emptyset$. Given that Q is non-hierarchical, Q has at least one x-atom, one y-atom, and one xy-atom (note that it could have no $\emptyset$-atom). Without loss of generality, we can reorder the atoms in Q and assume that there exist numbers m_{xy}, m_x, and m_y such that $0 \leq m_{xy} < m_x < m_y < m$ and for every $i < m$: if $i < m_{xy}$, then $R_i(\{\bar{x}_i\})$ is an $\emptyset$-atom; if $m_{xy} \leq i < m_x$, then $R_i(\{\bar{x}_i\})$ is an xy-atom; if $m_x \leq i < m_y$, then $R_i(\{\bar{x}_i\})$ is an x-atom; and if $m_y \leq i$, then $R_i(\{\bar{x}_i\})$ is an y-atom. In other words, Q is of the form:

$$Q(\bar{x}) \ \leftarrow \ \underbrace{R_0(\bar{x}_0), \ldots,}_{\emptyset\text{-atoms}} \underbrace{R_{m_{xy}}(\bar{x}_{m_{xy}}), \ldots,}_{xy\text{-atoms}} \underbrace{R_{m_x}(\bar{x}_{m_x}), \ldots,}_{x\text{-atoms}} \underbrace{R_{m_y}(\bar{x}_{m_y}), \ldots, R_{m-1}(\bar{x}_{m-1})}_{y\text{-atoms}} \qquad (\ddagger)$$

Similar than for the simple case, for every $i, j \in \mathbb{N}$ we define the stream:

$$\mathcal{S}_{i,j} \ = \ R_0(\bar{a}_0), \ldots, R_{m-1}(\bar{a}_{m-1}), \ldots$$

where for every tuple $\bar{a}_k$ each of its variables will be mapped to zero, except for x and y, which will be mapped to i and j, respectively. It is clear that the valuation $\nu = \{R_k(\bar{x}_k) \to k \mid k < m\} \in [\![Q]\!]_{m-1}(\mathcal{S}_{i,j})$ for every $i, j \in \mathbb{N}$.

From now, we follow the same strategy to the simple case presented above. Once again, assume there exists a PCEA $\mathcal{P} = (P, \mathbf{U}, \mathbf{B}, \Omega, \Delta, F)$ such that $\mathcal{P} \equiv Q$. Then, $\nu \in [\![\mathcal{P}]\!]_{m-1}(\mathcal{S}_{i,j})$ for every $i, j \in \mathbb{N}$. Let $\tau_{i,j}$ be the run tree of $\mathcal{P}$ over $\mathcal{S}_{i,j}$ that ends at position $m-1$ and produces ν. Given that $\mathcal{P}$ is finite, let N be the number of different runs $\tau_{i,j}$ of $\mathcal{P}$.

By the reordering of Q like ($\ddagger$) and the definition of $\mathcal{S}_{i,j}$, we know that all run trees $\tau_{i,j}$ have at the root the y-atom $R_{m-1}(\bar{x}_{m-1})$. Given that Q has at least one xy-atom, then every run tree τ_{ij} has at least one xy-atom in a node. For every $i, j \in \mathbb{N}$, let $\bar{u}_{i,j}$ be a node in $\tau_{i,j}$ that is the closest to the root of $\tau_{i,j}$ and its labeled by an xy-atom. We know that $\bar{u}_{i,j}$ exists and the path from $\bar{u}_{i,j}$ to $\mathtt{root}(\tau_{i,j})$ has zero or more x-atoms, followed by only y-atoms until the root. Indeed, by the construction of $\mathcal{S}_{i,j}$ it cannot be a switch from an y-atom to an x-atom in the path from $\bar{u}_{i,j}$ to the root. If this path contains at least one x-atom, we say that $\tau_{i,j}$ has a *line shape*. Otherwise, we say that $\tau_{i,j}$ has a *tree shape*. Note that if there is no x-atom from $\bar{u}_{i,j}$ to the root, then there must exists a node in $\tau_{i,j}$ labeled by an x-atom whose path to the root only contains y-atoms. For this reason, it makes sense to name it as tree shape (i.e., like in the simple case above).

Finally, we can use the same argument as for the simple case to prove that $\mathcal{P} \not\equiv \mathcal{Q}$. Namely, for every $i \leq (N+1) \cdot N$, there must exists $j_i, j'_i \leq N$ such that $j_i \neq j'_i$ and τ_{i,j_i} and τ_{i,j'_i} are equivalent, namely, they have the same shape, states, and transitions. Then we must distinguish between the two possible scenarios: (1) τ_{i,j_i} and τ_{i,j'_i} are line shape for some $i \leq (N+1) \cdot N$, and (2) τ_{i,j_i} and τ_{i,j'_i} are tree shape for every $i \leq (N+1) \cdot N$. In both cases, we apply the same argument like in the simple case by constructing a stream $\mathcal{S}^*$ such that $\nu \in [\![\mathcal{P}]\!]_{m-1}(\mathcal{S}^*)$, but $\nu \notin [\![\mathcal{Q}]\!]_{m-1}(\mathcal{S}^*)$ which will lead to a contradiction. From there, we conclude that $\mathcal{P} \not\equiv \mathcal{Q}$ for every PCEA $\mathcal{P}$ over σ.

C Proofs of Section 5

Proof of Theorem 5.2

Proof. Let $w \in \mathbb{N}$ be a window size, DS_w be a simple data structure and $\mathsf{n} \in \mathrm{Nodes}(\mathsf{DS}_w)$ be a node of the data structure. The valuations in $[\![\mathsf{n}]\!]_i^w$ are defined as:

$$[\![\mathsf{n}]\!]_i^w \;:=\; \{\!\{\nu \in [\![\mathsf{n}]\!] \mid |i - \min(\nu)| \leq w\}\!\}.$$

with

$$[\![\mathsf{n}]\!]_{\text{prod}} := \{\!\{\nu_{L(\mathsf{n}),i(\mathsf{n})}\}\!\} \oplus \bigoplus_{\mathsf{n}' \in \text{prod}(\mathsf{n})} [\![\mathsf{n}']\!] \qquad [\![\mathsf{n}]\!] := [\![\mathsf{n}]\!]_{\text{prod}} \cup [\![\text{uleft}(\mathsf{n})]\!] \cup [\![\text{uright}(\mathsf{n})]\!].$$

Following the definitions used in [22], we will say that the algorithm enumerates the results $\nu \in [\![\mathsf{n}]\!]_i^w$ by writing $\#\nu_1\#\nu_2\#\ldots\#\nu_m\#$ to the output registers, where $\# \notin \Omega$ is a separator symbol. Let $\mathsf{time}(i)$ be the time in the enumeration when the algorithm writes the i-th symbol #, we define the $\mathsf{delay}(i) = \mathsf{time}(i+1) - \mathsf{time}(i)$ for each $i \leq m$. We say that the enumeration has *output-linear delay* if there is a constant k such that for every $i \leq m$ it holds that $\mathsf{delay}(i) \leq k \cdot |\nu_i|$.

To output the first valuation of $[\![\mathsf{n}]\!]_i^w$ we need to (1) determine if $[\![\mathsf{n}]\!]_i^w = \varnothing$ and (2) build the valuation by calculating the products in $[\![\mathsf{n}]\!]$. We can know that $[\![\mathsf{n}]\!]_i^w \neq \varnothing$ iff $|i - \text{max-start}(\mathsf{n})| \leq w$, and since the value of $\text{max-start}(\mathsf{n}) = \max\{\min(\nu) \mid \nu \in [\![\mathsf{n}]\!]_{\text{prod}}\}$ is stored in every node n and we are doing a simple calculation with constants, we can check (1) in constant time. Note that it is not necessary to recursively check the max-start of the rest of the nodes in $[\![\mathsf{n}]\!]_{\text{prod}}$ since they are considered in the definition.

On the other hand, the product of two bags of valuations V, V' is defined as the bag $V \oplus V' = \{\!\{\nu \oplus \nu' \mid \nu \in V, \nu' \in V'\}\!\}$, where $\nu \oplus \nu'$ is the product of two valuations, defined as a valuation such that $[\nu \oplus \nu'](\ell) = \nu(\ell) \cup \nu'(\ell)$ for every $\ell \in \Omega$. With these definitions, we can enumerate a single valuation $\nu \in [\![\mathsf{n}]\!]_{\text{prod}}$ by calculating the union between a valuation $\nu_{\mathsf{n}} \in U(\{\!\{\nu_{L(\mathsf{n}),i(\mathsf{n})}\}\!\})$ and $\nu_{\mathsf{n}'} \in \text{prod}(\mathsf{n}')$ for each $n' \in \text{prod}(\mathsf{n}')$. It is easy to see that we can complete (2) by both calculating and writing this valuation in linear time. It is worth noting that we can make sure that we find valuations inside of the time window in constant time by traversing every bag in reverse order (starting from the valuations with a higher to lower $\min\{\nu\}$).

After enumerating the first output, we can continue traversing the bags of valuations, checking in constant time if $|i - \min\{\nu_{\mathsf{n}'}\} \leq w|$. In the worst case, which will occur right after writing the last valuation in the output, we will have to check that $|i - \min\{\nu_{\mathsf{n}'}\} \leq w|$ for every node $\mathsf{n}' \in \text{prod}(n)$, but since each check takes constant time and there is one node for each valuation we are adding to the output, this step can also be done in linear time with respect to $|\nu|$. Finally, after enumerating every output of $\text{prod}(n)$ inside the time window, we can recursively start the enumeration for $\text{uleft}(\mathsf{n})$ and $\text{uright}(\mathsf{n})$ in constant time, which will maintain an output-linear delay. ◻

Proof of Proposition 5.3

Proof. Fix $k, w \in \mathbb{N}$ and assume that one performs $\texttt{union}(\mathsf{n}_1, \mathsf{n}_2)$ over DS_w with the same position $i = i(\mathsf{n}_2)$ at most k times. In the following, we first prove the proposition with an implementation of the `union` operation that is not fully persistent and then show how to modify the implementation to maintain this property.

Let $\mathsf{n}_1, \mathsf{n}_2 \in \mathrm{Nodes}(\mathrm{DS}_w)$ be two nodes such that $\max(\mathsf{n}_1) \leq i(\mathsf{n}_2)$ and $\mathrm{uleft}(\mathsf{n}_2) = \mathrm{uright}(\mathsf{n}_2) = \perp$. We say that $\mathsf{n}_1 \leq \mathsf{n}_2$ iff (1) $\text{max-start}(\mathsf{n}_1) \leq \text{max-start}(\mathsf{n}_2)$ and (2) if $\text{max-start}(\mathsf{n}_1) = \text{max-start}(\mathsf{n}_2)$ then $i(\mathsf{n}_1) \leq i(\mathsf{n}_2)$.

Recall that this operation requires inserting n_2 into n_1 and it outputs a fresh node n_u such that $[\![\mathsf{n}_u]\!]_i^w := [\![\mathsf{n}_1]\!]_i^w \cup [\![\mathsf{n}_2]\!]_i^w$.

If $|\text{max-start}(\mathsf{n}_1 - i(n_2))| > w$ then all of the outputs from n_1 are now out of the time window, so $[\![\mathsf{n}_1]\!]_i^w \cup [\![\mathsf{n}_2]\!]_i^w = [\![\mathsf{n}_2]\!]_i^w$ and therefore $\texttt{union}(\mathsf{n}_1, \mathsf{n}_2) = \mathsf{n}_2$. The time it will take to do this operation will be the time necessary to insert n_2 in DS_w, which we will analyze later.

On the other hand, if $|\text{max-start}(\mathsf{n}_1 - i(\mathsf{n}_2))| \leq w$, we have to consider the outputs of both nodes, n_2 and n_1. First check how n_1 compares with n_2. If $\mathsf{n}_1 \leq \mathsf{n}_2$, then we have to create the new node $\mathsf{n} = \mathsf{n}_2$ such that $\mathrm{uleft}(\mathsf{n}) = \texttt{union}(\mathrm{uleft}(\mathsf{n}_2), \mathsf{n}_1)$ and, similarly, if $\mathsf{n}_1 > \mathsf{n}_2$, we need to create the new node $\mathsf{n} = \mathsf{n}_1$ such that $\mathrm{uleft}(\mathsf{n}) = \texttt{union}(\mathrm{uleft}(\mathsf{n}_1), \mathsf{n}_2)$. In both cases, we are only creating one node and switching or adding pointers between nodes a constant number of times. Although it might seem like a recursive operation at first glance, we know beforehand that $\text{max-start}(\mathsf{n}) \geq \text{max-start}(\mathrm{uleft}(\mathsf{n}))$, so there will be at most one other union process generated. Once again since we can do this part of the operation in constant time and the bulk of the operation will be the time necessary for the insertion of the new node.

We know that for every position $i = i(\mathsf{n}_2)$ we will perform a `union` operation at most k times. Starting with an empty data structure, there will be at most $k \cdot w$ nodes in DS_w given a time window w. Assuming DS_w is a perfectly balanced binary tree, this means that the tree has a depth of $\log_2(k \cdot w)$.

To ensure that the tree will always be balanced, we can add one bit of information to every node, which we will call *the direction bit*, that indicates which of the children of the node we need to visit for the insertion. If $bit(\mathsf{n}) = 0$, we must go to its left child and we must go to the right one otherwise. After each insertion, we need to change the value of the direction bit of every node in the path from the root to the newly inserted one, to avoid repeating the same path on the next insertion. This operation can be done in constant time for each node, so the time it will take to update all of the direction bits for each insertion will be exactly the depth of the tree.

Since one performs $\texttt{union}(\mathsf{n}_1, \mathsf{n}_2)$ over DS_w with the same position $i = i(\mathsf{n}_2)$ at most k times, if we start with an empty data structure, it will have at most $k \times w$ nodes after reading w tuples from the stream. To insert the next node n', by following the direction bits, we will end up in the oldest node of the tree n, but it is clear that $i(\mathsf{n}) \leq i(\mathsf{n}') + w$, meaning that $\text{max-start}(\mathsf{n}) - i(\mathsf{n}) \leq w$ which indicates that all of the outputs of n are outside of the time window and therefore we can safely remove n from the tree and replace it with n' without losing outputs. Given that the depth of DS_w is at most $k \cdot w$, and all of our previous operations take time proportional to the depth of the tree, we can conclude that the running time of the `union` operation is in $\mathcal{O}(\log(k \cdot w))$ for each call.

As we stated before, although the method we just discussed works and has a running time in $\mathcal{O}(\log(k \cdot w))$ for each call, it is not a fully persistent implementation, since we are removing nodes from the leaves when they are not producing an output and we are also modifying the direction bits of the nodes. To solve this problem, we can use the *path copying method*. With this method, whenever we need to modify a node, we create a copy with the modifications applied instead.

In our case, for every insertion we will create a copy of the entire path from the root to the new node, since we will modify the direction bit of each of these nodes, setting the modified copy of the root as the new root of the data structure. It is easy to see that with clever use of pointers, the copying of a node can be done in constant time, so the usage of this method does not increase the overall running time of the `union` operation. □

Proof of Proposition 5.5

Proof. Fix a time window size $w \in \mathbb{N}$ a stream $\mathcal{S}$, a position $i \in \mathbb{N}$ and an PCEA with equality predicates $\mathcal{P} = (Q, \mathbf{U}_{\text{lin}}, \mathbf{B}_{\text{eq}}, \Omega, \Delta, F)$. The output of the automaton $\mathcal{P}$ over $\mathcal{S}$ at position i with time window w is defined as the set of valuations:

$$[\![\mathcal{P}]\!]_i^w(\mathcal{S}) \;=\; \{\nu_\rho \mid \rho \text{ is an accepting run of } \mathcal{P} \text{ over } S \text{ at position } i \wedge |i - \min(\nu)| \leq w\}$$

We need to prove that Algorithm 1 enumerates every valuation ν_ρ without repetitions. One way to do this is showing that at any position in the stream the indices in H contain the information of every single run of $\mathcal{P}$ so far showing that the outputs for each of these runs can be enumerated.

First, let $i = 0$ and suppose that $\mathcal{S} = \{\!\{R(\bar{x}), \ldots\}\!\}$. H trivially contains the information of all the runs up to this point, so we need to show that this condition still holds after the first tuple.

Looking at the algorithm, after the Reset call, we start with $i = 0$, $\mathrm{DS}_w = \emptyset$, $N_p = \emptyset$ for every $p \in Q$. Calling FireTransitions$(R(\bar{x}), 0)$ we check each transitions satisfied by $R(\bar{x})$ and we register them in nodes for DS_w. Since$\mathcal{P}$ is unambiguous, there is only one transition that can lead to an accepting state, $e_f = (\emptyset, U, \emptyset, L_f, p_f) \in \Delta$ with $p_f \in F$ and for e_f we have $N = \{\}$ and $N_{p_f} = \texttt{extend}(L_f, 0, \{\})$. In addition, $\mathcal{P}$ can take (several)transitions that do not lead to a final state; these would be transitions of the form $e = (\emptyset, U, \emptyset, L, p) \in \Delta$ with $N_p = \texttt{extend}(L, 0, \{\})$.

On the other hand, UpdateIndices$(R(\bar{x}))$ uses the nodes created in FireTransitions and assigns them to every possible transition that could be satisfied by them. In particular, for every reached state p, we add each node in N_p to the data structure $H[e, p, \vec{\mathcal{B}}_p(t)]$, registering every incomplete run of $\mathcal{P}$.

Finally, we enumerate the outputs of each run that reached a final state. Since $\mathcal{P}$ is unambiguous, this enumeration will not have duplicates. It is easy to see that enumerate will output our only valuation since $p_f \in F$.

For the general case, suppose that H contains the information of every single run of $\mathcal{P}$ up until position $i - 1$ and that $\mathcal{S}[i] = S(\bar{y})$ and that we can enumerate every valuation in $[\![\mathcal{P}]\!]_{i-1}^w(\mathcal{S})$. We want to prove that after calling FireTransitions$(S(\bar{y}), i)$ and UpdateIndices$(S(\bar{y}))$, H will also contain the information of the runs up until i.

Once again we start with $\mathsf{N}_p = \emptyset$ for each $p \in Q$, but this time $H[e, p, \vec{\mathcal{B}}_p(t)]$ is not empty. Similar to the previous case, upon calling FireTransitions$(S(\bar{y}), i)$, we create a new node for every new state reached by any of the runs and it is clear by the definition of Δ and $\vec{\mathcal{B}}_p(t)$ that $S(\bar{y}) \in U$ and $\bigwedge_{p \in P} \mathsf{H}[e, p, \vec{\mathcal{B}}_p(t)] \neq \emptyset$ for a transition $e = (P, U, \mathcal{B}, L, q) \in \Delta$ iff there is a run tree ρ and a node $\bar{u}$ such that $\rho(\bar{u}) = (q, i, L)$.

In the same fashion, UpdateIndices$(S(\bar{y}))$ will thoroughly calculate for each transition and each state in those transitions the left projection of the binary predicate for $S(\bar{y})$, maintaining the data structure in H updated with the runs of $\mathcal{P}$.

Finally, the algorithm was already capable of enumerating every valuation that ends in a position $j < i$, and we get from FireTransitions$(S(\bar{y}), i)$ that every new accepting run will have its associated nodes. □

Received May 2024; revised August 2024; accepted September 2024