Streaming Enumeration on Nested Documents

Martín Muñoz
Pontificia Universidad Católica de Chile, Santiago, Chile
Millennium Institute for Foundational Research on Data, Santiago, Chile

Cristian Riveros
Pontificia Universidad Católica de Chile, Santiago, Chile
Millennium Institute for Foundational Research on Data, Santiago, Chile

Abstract

Some of the most relevant document schemas used online, such as XML and JSON, have a nested format. In the last decade, the task of extracting data from nested documents over streams has become especially relevant. We focus on the streaming evaluation of queries with outputs of varied sizes over nested documents. We model queries of this kind as Visibly Pushdown Transducers (VPT), a computational model that extends visibly pushdown automata with outputs and has the same expressive power as MSO over nested documents. Since processing a document through a VPT can generate a massive number of results, we are interested in reading the input in a streaming fashion and enumerating the outputs one after another as efficiently as possible, namely, with constant-delay. This paper presents an algorithm that enumerates these elements with constant-delay after processing the document stream in a single pass. Furthermore, we show that this algorithm is worst-case optimal in terms of update-time per symbol and memory usage.

2012 ACM Subject Classification Theory of computation → Database theory

Keywords and phrases Streaming, nested documents, query evaluation, enumeration algorithms

Digital Object Identifier 10.4230/LIPIcs.ICDT.2022.19

Related Version *Extended Version*: `https://arxiv.org/abs/2010.06037`

Funding This work was funded by ANID - Millennium Science Initiative Program - Code `ICN17_002`.

1 Introduction

Streaming query evaluation [2, 9] is the task of processing queries over data streams in one pass and with a limited amount of resources. This approach is especially useful on the web, where servers share data, and they have to extract the relevant content as they receive it. For structuring the data, the de facto structure on the web are nested documents, like XML or JSON. For querying, servers use languages designed for these purposes, like XPath, XQuery, or JSON query languages. As an illustrative example, suppose our data server (e.g. Web API) is continuously receiving XML documents of the form:

```
<doc> <a> <b/> <c/> <b/> </a> <c> <b/> <b/> </c> </doc> ...
```

and for each document it has to evaluate the query $\mathcal{Q} = //a/b$ (i.e., to extract all b-tags that are surrounded by an a-tag). The streaming query evaluation problem consists on reading these documents and finding all b-tags without storing the entire document on memory, i.e., by making one pass over the data and spending constant time per tag. In our example, we need to retrieve the 3rd and 5th tag as soon as the last tag `</doc>` is received. One could consider here that the server has to read an infinite stream and perform the query evaluation continuously, where it must enumerate partial outputs as one of the XML documents ends.

Researchers have studied the streaming query evaluation problem in the past, focusing on reducing the processing time or memory usage (see, e.g. [13]). Hence, they spent less effort on understanding the enumeration time of such a problem, regarding delay guarantees between outputs. Constant-delay enumeration is a new notion of efficiency for retrieving outputs [23, 44]. Given an instance of the problem, an algorithm with constant-delay enumeration performs a preprocessing phase over the instance to build some indices and then continues with an enumeration phase. It retrieves each output, one-by-one, taking a delay that is constant between any two consecutive outcomes. These algorithms provide a strong guarantee of efficiency since a user knows that, after the preprocessing phase, she will access the output as if the algorithm had already computed it. These techniques have attracted researchers' attention, finding sophisticated solutions to several query evaluation problems [11, 15, 10, 5, 26, 6].

In this work, we investigate the streaming query evaluation problem over nested documents by including enumeration guarantees, like constant-delay. We study the evaluation of queries given by Visibly Pushdown Transducers (VPT) over nested documents. These machines are the natural "output extension" of visibly pushdown automata, and have the same expressive power as MSO over nested documents. In particular, VPT can define queries like $\mathcal{Q}$ above or any fragment of query languages for XML or JSON included in MSO. Therefore, VPT allow considering the streaming query evaluation from a more general perspective, without getting married to a specific language (e.g., XPath).

We study the evaluation of VPT over a nested document in a streaming fashion. Specifically, we want to find a streaming algorithm that reads the document sequentially and spends constant time per input symbol. Furthermore, whenever needed, the algorithm can enumerate all outputs with output-linear delay. The main contribution of the paper is an algorithm with such characteristics for the class of I/O-unambiguous VPT. We can extend this algorithm by determinization to all VPT (i.e., in data complexity). Regarding memory consumption, we bound the amount of memory used in terms of the nesting of the document and the output weight. We show that our algorithm is worst-case optimal in the sense that there are instances where the maximum amount of memory required by any streaming algorithm is at least one of these two measures.

Related work. The problem of streaming query evaluation has been extensively studied in the last decades. Some work considered streaming verification, like schema validation [45] or type-checking [38], where the output is true or false. Other proposals [19, 42, 36, 31, 41] provided streaming algorithms for XPath or XQuery's fragments; however, extending them for reaching constant-delay enumeration seems unlikely. Furthermore, most of these works [38, 30, 29] assumed outputs of fixed size (i.e., tuples). People have also considered other aspects of streaming evaluation with outputs like earliest query answering [29] or bounded delay [28] (i.e., given the first visit of a node, find the earliest event that permits its selection). These aspects are orthogonal to the problem studied here. Another line of research is [12, 13], which presents space lower bounds for evaluating fragments of XPath or XQuery over streams. These works do not consider restrictions on the delay to give outputs.

Visibly pushdown automata [4] are a model usually used for streaming evaluation of boolean queries [38]. In [24, 3], authors studied the evaluation of VPT in a streaming fashion, but none of them saw enumeration problems. Other extensions [27] for streaming evaluation have been analyzed but restricted to fixed-size outputs, and constant-delay was not included.

Constant-delay algorithms have been studied for several classes of query languages and structures [44], as we already discussed. In [10, 5], researchers considered query evaluation over trees (i.e., a different representation for nested documents), but their algorithms are for offline evaluation and it is not clear how to extend this algorithm for the online setting. This research is extended with updates in [7], which can encode streams by inserting new data items to the left. However, their update-time is logarithmic, and our proposal can do it with constant time (i.e., in data complexity). Furthermore, to the best of our knowledge it is unclear how to modify the work in [7] to get constant update-time in our scenario. Streaming evaluation with constant-delay enumeration was included in the context of dynamic query evaluation [34, 16, 40, 37] or complex event processing [33, 32]. In both cases, the input cannot encode nested documents, and their results do not apply.

2 Preliminaries

Well-nested words and streams. As usual, given a set Σ we denote by Σ^* all finite words with symbols in Σ where $\varepsilon \in \Sigma^*$ represents the empty word of length 0.

We will work over a *structured alphabet* $\Sigma = (\Sigma^<, \Sigma^>, \Sigma^|)$ comprised of three disjoint sets $\Sigma^<$, $\Sigma^>$, and $\Sigma^|$ that contain *open*, *close*, and *neutral* symbols respectively (in [4, 25] these sets are named *call*, *return*, and *local*, respectively). Furthermore, we will denote symbols in $\Sigma^<$, $\Sigma^>$ or $\Sigma^|$ by $<a$, $a>$, and a, respectively. Instead, we will use s to denote any symbol in $\Sigma^<$, $\Sigma^>$, or $\Sigma^|$. The set of *well-nested words* over Σ, denoted as $\Sigma^{<*>}$, is defined as the closure of the following rules: $\Sigma^| \cup \{\varepsilon\} \subseteq \Sigma^{<*>}$, if $w_1, w_2 \in \Sigma^{<*>} \setminus \{\varepsilon\}$ then $w_1 \cdot w_2 \in \Sigma^{<*>}$, and if $w \in \Sigma^{<*>}$ and $<a \in \Sigma^<$ and $b> \in \Sigma^>$ then $<a \cdot w \cdot b> \in \Sigma^{<*>}$. In addition, we will work with prefixes of well-nested words, that we call *prefix-nested words*. We denote the set of prefixes of $\Sigma^{<*>}$ as $\mathsf{prefix}(\Sigma^{<*>})$. Also, we will sometimes use $w[i]$ to refer to the i-th symbol in a word w.

A *stream* $\mathcal{S} = s_1 s_2 \cdots$ is an infinite sequence where $s_i \in \Sigma^< \cup \Sigma^> \cup \Sigma^|$. Given a stream $\mathcal{S} = s_1 s_2 \ldots$ and positions $i, j \in \mathbb{N}$ such that $i \leq j$, the word $\mathcal{S}[i, j]$ is the sequence $s_i \cdots s_j$. We also use this notation to refer to subsequences of infinite sequences that are not composed of symbols in Σ. For a stream $\mathcal{S}$, we will always assume that for each $i \in \mathbb{N}$, the word $\mathcal{S}[1, i]$ is a prefix of some nested word (i.e., it can be completed to form a nested word). We also consider a method $\texttt{yield}[\mathcal{S}]$ which can be called to access each element of $\mathcal{S}$ sequentially.

Visibly pushdown automata. A *visibly pushdown automaton* [4] (VPA) is a tuple $\mathcal{A} = (Q, \Sigma, \Gamma, \Delta, I, F)$ where Q is a finite set of states, $\Sigma = (\Sigma^<, \Sigma^>, \Sigma^|)$ is the input alphabet, Γ is the stack alphabet, $\Delta \subseteq (Q \times \Sigma^< \times Q \times \Gamma) \cup (Q \times \Sigma^> \times \Gamma \times Q) \cup (Q \times \Sigma^| \times Q)$ is the transition relation, $I \subseteq Q$ is a set of initial states, and $F \subseteq Q$ is a set of final states. A transition $(q, <a, q', \gamma)$ is a *push-transition* where on reading $<a \in \Sigma^<$, γ is pushed onto the stack and the current state switches from q to q'. Conversely, $(q, a>, \gamma, q')$ is a *pop-transition* where on reading $a> \in \Sigma^>$ from the input and γ from the top of the stack, the current state changes from q to q', and the symbol γ is popped. Lastly, we say that (q, a, q') is a *neutral transition* if $a \in \Sigma^|$, where there is no stack operation.

A stack is a finite sequence σ over Γ where the top of the stack is the first symbol on σ. For a well-nested word $w = s_1 \cdots s_n$ in $\Sigma^{<*>}$, a run of $\mathcal{A}$ on w is a sequence $\rho = (q_1, \sigma_1) \xrightarrow{s_1} \ldots \xrightarrow{s_n} (q_{n+1}, \sigma_{n+1})$, where each $q_i \in Q$, $\sigma_i \in \Gamma^*$, $q_1 \in I$, $\sigma_1 = \varepsilon$, and for every $i \in [1, n]$ the following holds: (1) if $s_i \in \Sigma^<$, then there is $\gamma \in \Gamma$ such that $(q_i, s_i, q_{i+1}, \gamma) \in \Delta$ and $\sigma_{i+1} = \gamma\sigma_i$, (2) if $s_i \in \Sigma^>$, then there is $\gamma \in \Gamma$ such that $(q_i, s_i, \gamma, q_{i+1}) \in \Delta$ and $\sigma_i = \gamma\sigma_{i+1}$, and (3) if $s_i \in \Sigma^|$, then $(q_i, s_i, q_{i+1}) \in \Delta$ and $\sigma_{i+1} = \sigma_i$. A run ρ is accepting if $q_{n+1} \in F$. A well-nested word $w \in \Sigma^{<*>}$ is accepted by a VPA $\mathcal{A}$ if there is an accepting run of $\mathcal{A}$ on w. The language $\mathcal{L}(\mathcal{A})$ is the set of well-nested words accepted by $\mathcal{A}$. Note that if ρ is an accepting run of $\mathcal{A}$ on a well-nested word w, then $\sigma_{n+1} = \varepsilon$. A set of well-nested words $\mathcal{L} \subseteq \Sigma^{<*>}$ is called a visibly pushdown language if there exists a VPA $\mathcal{A}$ such that $\mathcal{L} = \mathcal{L}(\mathcal{A})$.

A VPA $\mathcal{A} = (Q, \Sigma, \Gamma, \delta, I, F)$ is said to be *deterministic* if $|I| = 1$ and δ is a function subset of $(Q \times \Sigma^{<} \to Q \times \Gamma) \cup (Q \times \Sigma^{>} \times \Gamma \to Q) \cup (Q \times \Sigma^{|} \to Q)$. We also say that $\mathcal{A}$ is *unambiguous* if, for every $w \in \mathcal{L}(\mathcal{A})$, there exists exactly one accepting run of $\mathcal{A}$ on w. In [4], it is shown that for every VPA there exists an equivalent deterministic VPA of at most exponential size.

Model of computation. As it is common in the enumeration algorithms literature [10, 21, 44], for our algorithms we assume the computational model of Random Access Machines (RAM) with uniform cost measure, and addition and subtraction as basic operations [1]. We assume that a RAM has read-only input registers where the machine places the input, read-write work registers where it does the computation, and write-only output registers where it gives the output (i.e., the enumeration of the results).

3 Streaming evaluation with output-linear delay

We are interested in defining a notion of a streaming enumeration problem: to evaluate a query over a stream and to enumerate the outputs with bounded delay whenever there is such. Towards this goal, we want to restrict the amount of resources used (i.e., time and space) and impose strong guarantees on the delay. As our gold standard, we consider the notion of *output-linear delay* defined in [26]. This notion is a refinement of the definition of constant-delay [44] or linear-delay [21] enumeration that better fits our purpose. Altogether, our plan for this section is to define a streaming enumeration problem and then provide a notion of efficiency that a solution for this problem should satisfy.

We adopt the setting of relations to formalize a streaming enumeration problem [35, 8]. First, we need to define what is an enumeration problem outside the stream setting. Let Ω be an alphabet. An enumeration problem is a relation $R \subseteq (\Omega^* \times \Omega^*) \times \Omega^*$. For each pair $((q, x), y) \in R$ we view (q, x) as the input of the problem and y as a possible output for (q, x). Furthermore, we call q the query and x the data. This separation allows for a fine-grained analysis of the query complexity and data complexity of the problem. For an instance (q, x) we define the set $[\![q]\!]_R(x) = \{y \mid ((q, x), y) \in R\}$ of all outputs of evaluating q over x.

A streaming enumeration problem is an extension of an enumeration problem R where the input is a pair $(q, \mathcal{S})$ such that $\mathcal{S}$ is an infinite sequence of elements in Ω. We identify two ways of extending an enumeration problem R that differ in the output sets that are desired at each position in the stream:

1. The *streaming full-enumeration problem* for R is one where the objective is to enumerate the set $[\![q]\!]_R(\mathcal{S}[1, n])$ at each position $n \geq 1$.
2. A *streaming* Δ*-enumeration problem* for R is one where the objective is to enumerate the set $[\![q]\!]_R^{\Delta}(\mathcal{S}[1, n]) = [\![q]\!]_R(\mathcal{S}[1, n]) \setminus \bigcup_{i<n} [\![q]\!]_R(\mathcal{S}[1, i])$ at each position $n \geq 1$.

These versions give us two different ways of returning the outputs. These notions have been studied previously in the context of incremental view maintenance [20] and more recently, for dynamic query evaluation [34, 16]. For the sake of simplification, in the following we provide all definitions for the full-enumeration scenario. All definitions can be extended to Δ-enumeration by changing $[\![q]\!]_R$ to $[\![q]\!]_R^{\Delta}$.

We turn now to our notion of efficiency for solving a streaming enumeration problem. Let $f \colon \mathbb{N} \to \mathbb{N}$. We say that $\mathcal{E}$ is a *streaming evaluation algorithm* for R with f*-update-time* if $\mathcal{E}$ operates in the following way: it receives a query q and reads the stream $\mathcal{S}$ by calling the `yield`$[\mathcal{S}]$ method sequentially. After the n-th call to `yield`$[\mathcal{S}]$, the algorithm processes the n-th data symbol in two phases:

- In the first phase, called the *update* phase, the algorithm updates a data structure D with the read symbol and the time spent is bounded by $\mathcal{O}(f(|q|))$.
- The second phase, called the *enumeration* phase, occurs immediately after each update phase and outputs $[\![q]\!]_R(\mathcal{S}[1,n])$ using D. During this phase the algorithm: (1) writes $\#y_1\#y_2\#\cdots\#y_m\#$ to the output registers where $\#$ is a distinct separator symbol not contained in Ω, and $y_1, y_2, \ldots, y_m$ is an enumeration (without repetitions) of the set $[\![q]\!]_R(\mathcal{S}[1,n])$, (2) it writes the first $\#$ as soon as the enumeration phase starts, and (3) it stops immediately after writing the last $\#$.

The purpose of separating $\mathcal{E}$'s operation into an update and enumeration phase is to make an output-sensitive analysis of $\mathcal{E}$'s complexity. Moreover, from a user perspective, this separation allows running the enumeration phase without interrupting the update phase. That is, the user could execute the enumeration phase in a separate machine, and its running time only depends on how many outputs she wants to enumerate.

For the enumeration phase, we measure the delay between two outputs as follows: For an input $x \in \Omega^*$, let $\#y_1\#y_2\#\cdots\#y_m\#$ be the output of the algorithm during any call to the enumeration phase. Furthermore, let $\mathsf{time}_i(x)$ be the time in the enumeration phase when the algorithm writes the i-th $\#$ when running on x for $i \leq m+1$. Define $\mathsf{delay}_i(x) = \mathsf{time}_{i+1}(x) - \mathsf{time}_i(x)$ for $i \leq m$. Then we say that $\mathcal{E}$ has *output-linear delay* if there exists a constant k such that for every $x \in \Omega^*$ and $i \leq m$ it holds that $\mathsf{delay}_i(x) \leq k \cdot |y_i|$. In other words, the number of instructions executed by $\mathcal{E}$ between the time that the i-th and the $(i+1)$-th $\#$ are written is linear on the size of y_i. Note that, in particular, an output-linear delay implies that the enumeration phase ends in constant time if there is no output for enumerating.

As the last ingredient, we define how to measure the memory space of a streaming evaluation. Note that after the n-th call a streaming evaluation algorithm with f-update time will necessarily use at most $\mathcal{O}(n \cdot f(|q|))$ bits of space. As a refinement of this bound, we say that this algorithm uses g-space over a query q and stream $\mathcal{S}$ if the number of bits used by it after the n-th call is in $\mathcal{O}(g(|q|, \mathcal{S}[1,n]))$.

Given a streaming enumeration problem, we say that it can be solved with update-time f, output-linear delay, and in g-space if there exists an algorithm such as the one described above. For Δ-enumeration, the notion of streaming evaluation algorithm also applies, even though it could be the case that one can find such an algorithm for full-enumeration but not for Δ-enumeration, and vice versa. Finally, the enumeration problem and solutions provided here are a formal refinement of the algorithmic notions proposed in the literature of streaming evaluation [29], dynamic query evaluation [16, 34], and complex event processing [33, 32].

4 Visibly pushdown transducers and main result

In this section, we present the definition of visibly pushdown transducers [25] (VPT), which are an extension of visibly pushdown automata to produce outputs. We use VPT as our computational model to represent queries with output. This model is general enough to include any query language for nested documents, like XML or JSON, whose expressive power is in MSO. After the setting is formalized, we state the main result of the paper.

A *visibly pushdown transducer* (VPT) is a tuple $\mathcal{T} = (Q, \Sigma, \Gamma, \Omega, \Delta, I, F)$ where Q, Σ, Γ, I, and F are the same as for VPA, Ω is the output alphabet with $\varepsilon \notin \Omega$, and $\Delta \subseteq (Q \times \Sigma^{<} \times (\Omega \cup \{\varepsilon\}) \times Q \times \Gamma) \cup (Q \times \Sigma^{>} \times (\Omega \cup \{\varepsilon\}) \times \Gamma \times Q) \cup (Q \times \Sigma^{|} \times (\Omega \cup \{\varepsilon\}) \times Q)$ is the transition relation. As usual for transducers, a symbol $s \in \Sigma^{<} \cup \Sigma^{>} \cup \Sigma^{|}$ is an input symbol that the machine reads and $o \in \Omega \cup \{\varepsilon\}$ is a symbol that the machine prints in an output tape. Furthermore, ε represents that no symbol is printed for that transition. A run ρ of $\mathcal{T}$ over a well-nested word $w = s_1 s_2 \cdots s_n \in \Sigma^{<*>}$ is a sequence of the form $\rho = (q_1, \sigma_1) \xrightarrow{s_1/o_1} \ldots \xrightarrow{s_n/o_n} (q_{n+1}, \sigma_{n+1})$ where $q_i \in Q$, $\sigma_i \in \Gamma^*$, $q_1 \in I$, $\sigma_1 = \varepsilon$ and for every $i \in [1, n]$ the following holds: (1) if $s_i \in \Sigma^<$, then $(q_i, s_i, o_i, q_{i+1}, \gamma) \in \Delta$ for some $\gamma \in \Gamma$ and $\sigma_{i+1} = \gamma\sigma_i$, (2) if $s_i \in \Sigma^>$, then $(q_i, s_i, o_i, \gamma, q_{i+1}) \in \Delta$ for some $\gamma \in \Gamma$ and $\sigma_i = \gamma\sigma_{i+1}$, and (3) if $s_i \in \Sigma^|$, then $(p_i, s_i, o_i, q_{i+1}) \in \Delta$ and $\sigma_i = \sigma_{i+1}$. We say that the run is accepting if $q_{n+1} \in F$. We call a pair (q_i, σ_i) a configuration of ρ. Finally, the output of an accepting run ρ is defined as: $\mathsf{out}(\rho) = \mathsf{out}(o_1, 1) \cdot \ldots \cdot \mathsf{out}(o_n, n)$ where $\mathsf{out}(o, i) = \varepsilon$ when $o = \varepsilon$ and (o, i) otherwise. Note that in $o_1 \cdots o_n$ we use ε as a symbol, and in $\mathsf{out}(\rho)$ we use ε as the empty string. Given a VPT $\mathcal{T}$ and a $w \in \Sigma^{<*>}$, we define the set $[\![\mathcal{T}]\!](w)$ of all outputs of $\mathcal{T}$ over w as: $[\![\mathcal{T}]\!](w) = \{\mathsf{out}(\rho) \mid \rho \text{ is an accepting run of } \mathcal{T} \text{ over } w\}$.

Strictly speaking, our definition of VPT is richer than the one studied in [25]. In our definition of VPT each output element is a tuple (o, i) where o is the symbol and i is the output position, where for a standard VPT [25] an output element is just the symbol o. The extension presented here is indeed important for practical applications like in document spanners [26, 6] or in XML query evaluation [12, 46].

A first reasonable question is to understand what is the expressive power of VPT, namely, as a formalism for non-boolean query evaluation over nested words. For the Boolean case, it was shown [4] that VPA describe the same class of queries as MSO over nested words, called $\text{MSO}_{\mathsf{match}}$. Formally, fix a structured alphabet Σ and let $w \in \Sigma^{<*>}$ be a word of length n. We encode w as a structure:

$$\big([1, n], \leq, \{P_a\}_{a \in \Sigma}, \mathsf{match}\big)$$

where $[1, n]$ is the domain, $\leq$ is the total order over $[1, n]$, $P_a = \{i \mid w[i] = a\}$, and match is a binary relation over $[1, n]$ that corresponds to the matching relation of open and close symbols: $\mathsf{match}(i, j)$ is true iff $w[i]$ is an open symbol and $w[j]$ is its matching close symbol. A $\text{MSO}_{\mathsf{match}}$ formula φ over Σ is given by:

$$\varphi \;:=\; P_a(x) \;\mid\; x \in X \;\mid\; x \leq y \;\mid\; \mathsf{match}(x, y) \;\mid\; \neg\varphi \;\mid\; \varphi \vee \varphi \;\mid\; \exists x.\varphi \;\mid\; \exists X.\varphi$$

where $a \in \Sigma$, x and y are first-order variables and X is a monadic second order (MSO) variable. We write $\varphi(X_1, \ldots, X_n)$ where $X_1, \ldots, X_n$ are the free MSO variables of φ (first-order variables are a special case of MSO variables). Then we write $w \models \varphi(A_1, \ldots, A_n)$ for $A_1, \ldots, A_n \subseteq [1, n]$ when w satisfies φ by replacing each variable X_i with the set A_i. Here, we assume the standard semantics for MSO logic [39].

Given that VPT is an extension of VPA, it should not be a surprise that we can translate these results to VPT. In particular, the result in [4] can be easily extended to link VPT with formulas expressible in $\text{MSO}_{\mathsf{match}}$.

▶ **Proposition 1.** *Let $\varphi(X_1, \ldots, X_m)$ be a MSO_{match} formula with m free variables $X_1, \ldots, X_m$. There is a VPT $\mathcal{T}$ for which there is a one-to-one correspondence between the set $[\![\mathcal{T}]\!](w)$ and the set $\{(A_1, \ldots, A_m) \mid w \models \varphi(A_1, \ldots, A_m)\}$ for any word $w \in \Sigma^{<*>}$. Moreover, for every VPT $\mathcal{T}$ there is an MSO_{match} formula $\varphi(X_1, \ldots, X_m)$ for which the same one-to-one correspondence holds.*

In other words, VPT has the same expressive power as MSO over nested words. Given that fragments of query languages over nested documents (e.g., navigational XPath [47], JSON Navigational Logic [17]) are usually included in MSO, this shows that VPT is an expressive formalism for query evaluation over nested documents.

We say that a VPT $\mathcal{T} = (Q, \Sigma, \Gamma, \Omega, \Delta, I, F)$ is *input/output deterministic* (I/O-deterministic for short) if $|I| = 1$ and Δ is a partial function of the form $\Delta : (Q \times \Sigma^{<} \times \Omega \to Q \times \Gamma) \cup (Q \times \Sigma^{>} \times \Omega \times \Gamma \to Q) \cup (Q \times \Sigma^{|} \times \Omega \to Q)$. On the other hand, we say that $\mathcal{T}$ is *input/output unambiguous* (I/O-unambiguous for short) if for every $w \in \Sigma^{<*>}$ and every $\mu \in [\![\mathcal{T}]\!](w)$ there is exactly one accepting run ρ of $\mathcal{T}$ over w such that $\mu = \mathsf{out}(\rho)$. Notice that an I/O-deterministic VPT is also I/O-unambiguous and in both models for each output there exists at most one run. The definition of I/O-deterministic is in line with the notion of I/O-deterministic variable automata of [26] and I/O-unambiguous is a generalization of this idea that is enough for the purpose of our enumeration algorithm. One can show that for every VPT $\mathcal{T}$ there exists an equivalent I/O-deterministic VPT and, therefore, an equivalent I/O-unambiguous VPT.

▶ **Lemma 2.** *For every VPT $\mathcal{T}$ there exists an I/O-deterministic VPT $\mathcal{T}'$ of size $\mathcal{O}(2^{|Q|^2|\Gamma|})$ such that $[\![\mathcal{T}]\!](w) = [\![\mathcal{T}']\!](w)$ for every $w \in \Sigma^{<*>}$.*

In this paper, we are interested on the following streaming enumeration problem for VPT. Let $\mathcal{C}$ be a class of VPT (e.g. I/O-deterministic VPT).

Problem:	EnumVPT[$\mathcal{C}$]
Input:	a VPT $\mathcal{T} \in \mathcal{C}$ and $w \in \Sigma^{<*>}$
Output:	Enumerate $[\![\mathcal{T}]\!](w)$

The main result of the paper is that for the class of I/O-unambiguous VPT, the streaming full-enumeration version of this problem can be solved efficiently.

▶ **Theorem 3.** *The streaming full-enumeration problem of* EnumVPT *for the class of I/O-unambiguous VPT can be solved with update-time $\mathcal{O}(|Q|^2|\Delta|)$ and output-linear delay. For the general class of VPT, it can be solved with update-time $\mathcal{O}(2^{|Q|^2|\Delta|})$ and output-linear delay.*

The result for the class of all VPT is a consequence of Lemma 2 and the enumeration algorithm for I/O-unambiguous VPT (see Section 5 and 6). For both cases, if the VPT is fixed (i.e., in data complexity), then the update-time of the streaming algorithm is constant.

For the streaming version of EnumVPT, one can have Δ-enumeration with a small loss of efficiency by solving the full-enumeration problem. Specifically, one can show that for any I/O-unambiguous VPT $\mathcal{T}$ there is an I/O-unambiguous VPT $\mathcal{T}'$ of linear size with respect to $|\mathcal{T}|$ such that $[\![\mathcal{T}']\!](w) = [\![\mathcal{T}]\!](w) \setminus \bigcup\{[\![\mathcal{T}]\!](w[1,i]) \mid i < |w|, w[1,i] \in \Sigma^{<*>}\}$ for each $w \in \Sigma^{<*>}$.

▶ **Theorem 4.** *The streaming Δ-enumeration problem of* EnumVPT *for the class of I/O-unambiguous VPT can be solved with update-time $\mathcal{O}(|Q|^2|\Delta|)$ and output-linear delay. For the general class of VPT, it can be solved with update-time $\mathcal{O}(2^{|Q|^2|\Delta|})$ and output-linear delay.*

We could have considered a more general definition of VPT to produce outputs for prefix-nested words. This would be desirable for having some sort of *earliest query answering* [29] which is important in practical scenarios. We remark that the algorithm of Theorem 3 can be extended for this case at the cost of making the presentation more complicated. For the sake of presentation, we defer this extension to the full version of this paper.

Space lower bounds of evaluating a VPT. This subsection deals with the space used by the streaming evaluation algorithm of Theorem 3. Indeed, this algorithm could use linear space in the worst case. In the following we explore some lower bounds in the space needed by any algorithm, and show that this bound is tight for a certain type of VPT.

To study the minimum number of bits needed to solve EnumVPT we need to introduce some definitions. Fix a VPT $\mathcal{T}$ and $w \in \mathsf{prefix}(\Sigma^{<*>})$. Let $\mathsf{outputweight}(\mathcal{T}, w)$ be the number of positions less than $|w|$ that appear in some output of $[\![\mathcal{T}]\!](w \cdot w')$ for some $w \cdot w' \in \Sigma^{<*>}$. Furthermore, for a well-nested word u let $\mathsf{depth}(u)$ be the maximum number of nesting pairs inside u, formally, $\mathsf{depth}(a) = 0$ for $a \in \Sigma^{|} \cup \{\varepsilon\}$, $\mathsf{depth}(u_1 \cdot u_2) = \max\{\mathsf{depth}(u_1), \mathsf{depth}(u_2)\}$, and $\mathsf{depth}(<\!a \cdot u \cdot b\!>) = \mathsf{depth}(u) + 1$. For $w \in \mathsf{prefix}(\Sigma^{<*>})$, we define $\mathsf{depth}(w) = \min\{\mathsf{depth}(w') \mid w \text{ is a prefix of } w'\}$. We can now state some worst-case space lower bounds for EnumVPT.

▶ **Proposition 5.**
1. *There exists a VPT $\mathcal{T}$ such that every streaming evaluation algorithm for EnumVPT with input $\mathcal{T}$ and $\mathcal{S}$ requires at least $\Omega(\mathsf{depth}(\mathcal{S}[1,n]))$ bits of space.*
2. *There exists a VPT $\mathcal{T}$ such that every streaming evaluation algorithm for EnumVPT with input $\mathcal{T}$ and $\mathcal{S}$ requires at least $\Omega(\mathsf{outputweight}(\mathcal{T}, \mathcal{S}[1,n]))$ bits of space.*

In [12, 13], the authors provide lower bounds on the amount of space needed for evaluating XPath in terms of the nesting and the concurrency (see [12] for a definition). One can show that the output weight of $\mathcal{T}$ and w is always above the concurrency of $\mathcal{T}$ and w. Despite this, one can check that both notions coincide for the space lower bound given in Proposition 5.

The previous results show that, in the worst case, any streaming evaluation algorithm for VPT will require space of at least the depth of the document or the output weight. To show that Theorem 3 is optimal in the worst-case, we need to consider a further assumption of our VPT. We say that a VPT $\mathcal{T}$ is *trimmed* [18] if for every $w \in \mathsf{prefix}(\Sigma^{<*>})$ and every (partial) run ρ of $\mathcal{T}$ over w, there exists w' and an accepting run ρ' of $\mathcal{T}$ over $w \cdot w'$ such that ρ is a prefix of ρ'. This notion is the analog of trimmed non-deterministic automata. Similarly to Lemma 2, one can show that for every VPT $\mathcal{T}$ there exists a trimmed I/O-deterministic VPT $\mathcal{T}'$ equivalent to $\mathcal{T}$ (i.e., by extending the construction in [18] to VPT). The next result shows that, if the input to EnumVPT is a trimmed I/O-unambiguous VPT, then the memory footprint is at most the maximum between the depth and output weight of the input.

▶ **Proposition 6.** *The streaming enumeration problem of EnumVPT for the class of trimmed I/O-unambiguous VPT can be solved with update-time $\mathcal{O}(|Q|^2|\Delta|)$, output-linear delay and $\mathcal{O}(\max\{\mathsf{depth}(\mathcal{S}[1,n]), \mathsf{outputweight}(\mathcal{T}, \mathcal{S}[1,n])\} \times |Q|^2|\Delta|)$ space for every stream $\mathcal{S}$.*

Unfortunately, the algorithm provided in Theorem 3 is not *instance optimal*, in the sense of using the lowest number of bits needed for each specific VPT. Note that an instance optimal algorithm for the streaming enumeration problem of VPT will imply a solution to the weak evaluation problem, stated by Segoufin and Vianu [45]. This is an open problem in the area (see [14] for some recent results), so we leave this for future work.

5 Enumerable compact sets: a data structure for output-linear delay

This section presents a data structure, called Enumerable Compact Set (ECS), which is the cornerstone of our enumeration algorithm for VPT. This data structure is strongly inspired by the work in [5, 6]. Indeed, ECS can be considered a refinement of the d-DNNF circuits used in [5] or of the set circuits used in [6]. Several papers [43, 5, 7, 48] have considered circuits-like structures for encoding outputs and enumerate them with constant delay. The novelty of ECS is twofold. First, we use ECS for solving a streaming evaluation problem. Although people have studied streaming query evaluation with enumeration before [34, 16], this is the first work that uses a circuit-like data structure in an online setting. Second and more important, there is a difference in performance if we compare ECS to the previous approaches. In offline evaluation, constant delay algorithms usually create an initial circuit from the input, making several passes over the structure, building indices, and then running the enumeration process. Given time restrictions for the online evaluation, we cannot create a circuit and do this linear-time preprocessing before enumerating. On the contrary, we must extend the circuit-like data structure for each data item in constant time and then be ready to start the enumeration. This requirement justifies the need for a new data structure for representing and enumerating outputs. Therefore, ECS differs from previous proposals because each operation must take constant time, and we can run the enumeration process with output-linear delay, at any time and without any further preprocessing. In the following, we present ECS step-by-step to use it later in the next section.

Let Σ be a (possibly infinite) alphabet. We define an *Enumerable Compact Set* (ECS) as a tuple $\mathcal{D} = (\Sigma, V, I, \ell, r, \lambda)$ such that V and $I \subseteq V$ are finite sets of nodes, $\ell \colon I \to V$ and $r \colon I \to V$ are the *left* and *right* functions, and $\lambda \colon V \to \Sigma \cup \{\cup, \odot\}$ is a label function such that $\lambda(v) \in \{\cup, \odot\}$ if, and only if, $v \in I$. Further, we assume that the directed graph $(V, \{(v, \ell(v)), (v, r(v)) \mid v \in V\})$ is acyclic. We call the nodes in I *inner nodes* and the nodes in $V \setminus I$ *leaves*. Furthermore, for $v \in I$ we say that v is a *product node* if $\lambda(v) = \odot$, and a *union node* if $\lambda(v) = \cup$. We define the size of $\mathcal{D}$ as $|\mathcal{D}| = |V|$. For each node v in $\mathcal{D}$, we associate a set of words $\mathcal{L}_{\mathcal{D}}(v)$ recursively as follows: (1) $\mathcal{L}_{\mathcal{D}}(v) = \{a\}$ whenever $\lambda(v) = a \in \Sigma$, (2) $\mathcal{L}_{\mathcal{D}}(v) = \mathcal{L}_{\mathcal{D}}(\ell(v)) \cup \mathcal{L}_{\mathcal{D}}(r(v))$ whenever $\lambda(v) = \cup$, and (3) $\mathcal{L}_{\mathcal{D}}(v) = \mathcal{L}_{\mathcal{D}}(\ell(v)) \cdot \mathcal{L}_{\mathcal{D}}(r(v))$ whenever $\lambda(v) = \odot$, where $L_1 \cdot L_2 = \{w_1 \cdot w_2 \mid w_1 \in L_1 \text{ and } w_2 \in L_2\}$.

The size $|\mathcal{L}_{\mathcal{D}}(v)|$ can be exponential with respect to $|\mathcal{D}|$. For this reason, we say that $\mathcal{D}$ is a *compact* representation of $\mathcal{L}_{\mathcal{D}}(v)$ for any $v \in V$. Although $\mathcal{L}_{\mathcal{D}}(v)$ is very large, the goal is to enumerate all of its elements efficiently. Specifically, we consider the following problem:

Problem:	ENUM-ECS
Input:	An ECS $\mathcal{D} = (\Sigma, V, I, \ell, r, \lambda)$ and $v \in V$.
Output:	Enumerate the set $\mathcal{L}_{\mathcal{D}}(v)$ without repetitions.

Plus, we want to solve ENUM-ECS with output-linear delay. To reach this goal we need to impose two additional restrictions on $\mathcal{D}$. The first restriction is to guarantee that $\mathcal{D}$ is not ambiguous, namely, for each $w \in \mathcal{L}_{\mathcal{D}}(v)$ there is at most one way to retrieve w from $\mathcal{D}$. Formally, we say that $\mathcal{D}$ is *unambiguous* if $\mathcal{D}$ satisfies the following two properties: (1) for every union node v it holds that $\mathcal{L}_{\mathcal{D}}(\ell(v))$ and $\mathcal{L}_{\mathcal{D}}(r(v))$ are disjoint, and (2) for every product node v and for every $w \in \mathcal{L}_{\mathcal{D}}(v)$, there exists a unique way to decompose $w = w_1 \cdot w_2$ such that $w_1 \in \mathcal{L}_{\mathcal{D}}(\ell(v))$ and $w_2 \in \mathcal{L}_{\mathcal{D}}(r(v))$. Thus, if $\mathcal{D}$ is unambiguous, there will be no duplicates if we enumerate $\mathcal{L}_{\mathcal{D}}(v)$ directly, given that there is no way of producing the same element in two different ways.

The second restriction is to guarantee that, for each node v, there exists an output or, more specifically, a symbol of an output *close* to v, in the sense that it can be reached in a bounded number of steps. This is not always the case for an ECS. For example, consider a balanced tree of union nodes where all the outputs are at the leaves. One has to traverse a logarithmic number of nodes from the root to reach the first output. Note that product nodes do not pose this problem since the number of nodes that have to be traversed to produce a certain output is proportional to its length. For this reason, we define the notion of *k-bounded* ECS. Given an ECS $\mathcal{D}$, define the (left) output-depth of a node $v \in V$, denoted by $\mathsf{odepth}_{\mathcal{D}}(v)$, recursively as follows: $\mathsf{odepth}_{\mathcal{D}}(v) = 0$ whenever $\lambda(v) \in \Sigma$ or $\lambda(v) = \odot$, and $\mathsf{odepth}_{\mathcal{D}}(v) = \mathsf{odepth}_{\mathcal{D}}(\ell(v)) + 1$ whenever $\lambda(v) = \cup$. Then, for a fixed $k \in \mathbb{N}$ we say that $\mathcal{D}$ is k-bounded if $\mathsf{odepth}_{\mathcal{D}}(v) \leq k$ for all $v \in V$.

Given the definition of output-depth, we say that v is an output node of $\mathcal{D}$ if v is a leaf or a product node. Note that if $\mathcal{D}$ only has output nodes, then it is 0-bounded, and one can easily check that $\mathcal{L}_{\mathcal{D}}(v)$ can be enumerated with output-linear delay. Indeed, for a fixed k the same happens with every unambiguous and k-bounded ECS.

▶ **Proposition 7.** *Fix $k \in \mathbb{N}$. Let $\mathcal{D} = (\Sigma, V, I, \ell, r, \lambda)$ be an unambiguous and k-bounded ECS. Then the set $\mathcal{L}_{\mathcal{D}}(v)$ can be enumerated with output-linear delay for any $v \in V$.*

The enumeration algorithm above does not require any preprocessing over $\mathcal{D}$ and the main idea is to perform some sort of DFS traversal over the nodes. By this proposition, from now we assume that all ECS are unambiguous and k-bounded for some fixed k.

The next step is to provide a set of operations that allow extending an ECS $\mathcal{D}$ while maintaining k-boundedness. Furthermore, we require these operations to be fully-persistent: a data structure is called *fully-persistent* if every version can be both accessed and modified [22]. In other words, the previous version of the data structure is always available after each operation. To satisfy the last requirement, the strategy will consist in extending $\mathcal{D}$ to $\mathcal{D}'$ for each operation, by always adding new nodes and maintaining the previous nodes untouched. Then $\mathcal{L}_{\mathcal{D}'}(v) = \mathcal{L}_{\mathcal{D}}(v)$ for each node $v \in V$, and so, the structure is fully-persistent.

More precisely, fix an ECS $\mathcal{D} = (\Sigma, V, I, \ell, r, \lambda)$. In the following, we say that $\mathcal{D}' = (\Sigma, V', I', \ell', r', \lambda')$ is an extension of $\mathcal{D}$ if, and only if, $\mathsf{obj} \subseteq \mathsf{obj}'$ for every $\mathsf{obj} \in \{V, I, \ell, r, \lambda\}$. Further, we write $\mathsf{op}(I) \to O$ to define the signature of an operation op where I is the input and O is the output. Then for any $a \in \Sigma$ and $v_1, \ldots, v_4 \in V$, we define the operations:

$$\mathsf{add}(\mathcal{D}, a) \to (\mathcal{D}', v') \qquad \mathsf{prod}(\mathcal{D}, v_1, v_2) \to (\mathcal{D}', v') \qquad \mathsf{union}(\mathcal{D}, v_3, v_4) \to (\mathcal{D}', v')$$

such that $\mathcal{D}'$ is an extension of $\mathcal{D}$ and $v' \in V' \setminus V$ is a fresh node such that $\mathcal{L}_{\mathcal{D}'}(v') = \{a\}$, $\mathcal{L}_{\mathcal{D}'}(v') = \mathcal{L}_{\mathcal{D}}(v_1) \cdot \mathcal{L}_{\mathcal{D}}(v_2)$, and $\mathcal{L}_{\mathcal{D}'}(v') = \mathcal{L}_{\mathcal{D}}(v_3) \cup \mathcal{L}_{\mathcal{D}}(v_4)$, respectively. We assume that the union and prod respect properties (1) and (2) of an unambiguous ECS, that is, $\mathcal{L}_{\mathcal{D}}(v_1)$ and $\mathcal{L}_{\mathcal{D}}(v_2)$ are disjoint and, for every $w \in \mathcal{L}_{\mathcal{D}}(v_3) \cdot \mathcal{L}_{\mathcal{D}}(v_4)$, there exists a unique way to decompose $w = w_1 \cdot w_2$ such that $w_1 \in \mathcal{L}_{\mathcal{D}}(v_3)$ and $w_2 \in \mathcal{L}_{\mathcal{D}}(v_4)$.

Next, we show how to implement each operation. In fact, the case of add and prod are straightforward. For $\mathsf{add}(\mathcal{D}, a) \to (\mathcal{D}', v')$ define $V' := V \cup \{v'\}$, $I' := I$, and $\lambda'(v') = a$. One can easily check that $\mathcal{L}_{\mathcal{D}'}(v') = \{a\}$ as expected. For $\mathsf{prod}(\mathcal{D}, v_1, v_2) \to (\mathcal{D}', v')$ we proceed in a similar way: define $V' := V \cup \{v'\}$, $I' := I \cup \{v\}$, $\ell'(v') := v_1$, $r'(v') = v_2$, and $\lambda'(v') = \odot$. Then $\mathcal{L}_{\mathcal{D}'}(v') = \mathcal{L}_{\mathcal{D}}(v_1) \cdot \mathcal{L}_{\mathcal{D}}(v_2)$. Furthermore, one can check that each operation takes constant time, $\mathcal{D}'$ is a valid ECS (i.e. unambiguous and k-bounded), and the operations are fully-persistent (i.e. the previous version $\mathcal{D}$ is available).

To define the union, we need to be a bit more careful to guarantee output-linear delay, specifically, the k-bounded property. For a node $v \in V$, we say that v is *safe* if (1) $\mathsf{odepth}_{\mathcal{D}}(v) \leq 1$, and (2) if $\mathsf{odepth}_{\mathcal{D}}(v) = 1$, then $\mathsf{odepth}_{\mathcal{D}}(r(v)) \leq 1$. In other words, v is safe if v is an output node, or its left child is an output node, and the right child is either an output node or has output depth 1. Note that a leaf or a product node are safe nodes by definition and, thus, the add and prod operations always produce safe nodes. The trick then is to show that, if v_3 and v_4 are safe nodes, then we can implement $\mathsf{union}(\mathcal{D}, v_3, v_4) \to (\mathcal{D}', v')$ and produce a safe node v'. For this define $(\mathcal{D}', v')$ as follows:

- If v_3 or v_4 are output nodes then $V' := V \cup \{v'\}$, $I' := I \cup \{v'\}$, and $\lambda(v') := \cup$. Moreover, if v_3 is the output node, then $\ell'(v') := v_3$ and $r'(v') := v_4$. Otherwise, we connect $\ell'(v') := v_4$ and $r'(v') := v_3$.
- If v_3 and v_4 are not output nodes (i.e. both are union nodes), then $V' := V \cup \{v', u_1, u_2\}$, $I' := I \cup \{v', u_1, u_2\}$, $\ell'(v') := \ell(v_3)$, $r'(v') := u_1$, and $\lambda'(v') := \cup$; $\ell'(u_1) := \ell(v_4)$, $r'(u_1) := u_2$, and $\lambda'(u_1) := \cup$; $\ell'(u_2) := r(v_3)$, $r'(u_2) := r(v_4)$, and $\lambda'(u_2) := \cup$.

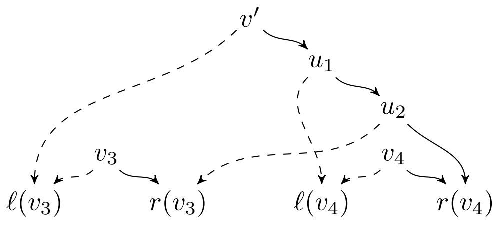

Figure 1 Gadget for $\mathsf{union}(\mathcal{D}, v_3, v_4)$. Nodes v', u_1, u_2, v_3 and v_4 are labeled as $\cup$. Dashed and solid lines denote the mappings in ℓ' and r' respectively.

This gadget is depicted in Figure 1 (note that a similar trick is used in [5] for computing an index over a circuit). This construction has several properties. First, one can easily check that $\mathcal{L}_{\mathcal{D}'}(v') = \mathcal{L}_{\mathcal{D}}(v_1) \cup \mathcal{L}_{\mathcal{D}}(v_2)$ and so the semantics is well-defined. Second, union can be computed in constant time in $|\mathcal{D}|$ given that we only need to add three fresh nodes, and the operation is fully-persistent given that we connect them without modifying $\mathcal{D}$. Furthermore, the produced node v' is safe in $\mathcal{D}'$, although nodes u_1 and u_2 are not necessarily safe. Finally, $\mathcal{D}'$ is 2-bounded whenever $\mathcal{D}$ is 2-bounded. This is straightforward to see for first case when v_3 or v_4 are output nodes. For the second case (i.e., Figure 1), we have to notice that v_3 and v_4 are safe, therefore $\ell(v_3)$ and $\ell(v_4)$ are output nodes, and then $\mathsf{odepth}_{\mathcal{D}'}(v') = \mathsf{odepth}_{\mathcal{D}'}(u_1) = 1$. Further, given that v_3 is safe, we know that $\mathsf{odepth}_{\mathcal{D}}(r(v_3)) \leq 1$, so $\mathsf{odepth}_{\mathcal{D}'}(u_2) \leq 2$. Given that the output depths of all fresh nodes in $\mathcal{D}'$ are bounded by 2 and $\mathcal{D}$ is 2-bounded, then we conclude that $\mathcal{D}'$ is 2-bounded as well.

By the previous discussion, if we start with an ECS $\mathcal{D}$ which is 2-bounded (or empty) and we apply the add, prod and union operators between safe nodes (which also produce safe nodes), then the result is 2-bounded as well. Finally, by Proposition 7, the result can be enumerated with output-linear delay.

▶ **Theorem 8.** *The operations* add*,* prod*, and* union *require constant time and are fully-persistent. Furthermore, if we start from an empty ECS $\mathcal{D}$ and apply* add*,* prod*, and* union *over safe nodes, the partial results $(\mathcal{D}', v')$ satisfy that v' is always a safe node and the set $\mathcal{L}_{\mathcal{D}'}(v)$ can be enumerated with output-linear delay for every node v.*

It is important to remark that restricting these operations only over safe nodes is a mild condition. Given that we will usually start from an empty ECS and apply these operations over previously returned nodes, the whole algorithm will always use safe nodes during its computation, satisfying the conditions of Theorem 8.

For technical reasons, our algorithm of the next section needs a slight extension of ECS by allowing leaves that produce the empty string ε. Let $\varepsilon \notin \Sigma$ be a symbol representing the empty string (i.e. $w \cdot \varepsilon = \varepsilon \cdot w = w$). We define an enumerable compact set with ε (called ε-ECS) as a tuple $\mathcal{D} = (\Sigma, V, I, \ell, r, \lambda)$ defined identically to an ECS except that $\lambda : V \to \Sigma \cup \{\cup, \odot, \varepsilon\}$ and $\lambda(v) \in \{\cup, \odot\}$ if, and only, if $v \in I$. Also, if $\lambda(v) = \varepsilon$, then $\mathcal{L}_{\mathcal{D}}(v) = \{\varepsilon\}$. The unambiguity restriction is the same for ε-ECS and one has to slightly extend k-boundedness to consider ε-nodes. However, to support the prod and union operations in constant time and to maintain the k-boundedness invariant, we need to extend the notion of safe nodes (called ε-safe) and the gadgets for prod and union.

▶ **Theorem 9.** *The operations* add*,* prod*, and* union *over ε-ECS take constant time and are fully-persistent. Furthermore, if we start from an empty ε-ECS $\mathcal{D}$ and apply* add*,* prod*, and* union *over ε-safe nodes, the partial results $(\mathcal{D}', v')$ satisfy that v' is always an ε-safe node and the set $\mathcal{L}_{\mathcal{D}'}(v)$ can be enumerated with output-linear delay for every node v.*

6 Evaluating visibly pushdown transducers with output-linear delay

The goal of this section is to describe an algorithm that takes an I/O-unambiguous VPT $\mathcal{T}$ plus a stream $\mathcal{S}$, and enumerates the set $[\![\mathcal{T}]\!](\mathcal{S}[1,n])$ for an arbitrary $n \geq 0$ with $\mathcal{O}(|Q|^2|\Delta|)$-update-time and output-linear delay. We divide the presentation of the algorithm into two parts. The first part explains the determinization of a VPA, which is instrumental in understanding our update phase. The second part gives the algorithm and proves its correctness. Given that a neutral symbol a can be represented as a pair $\texttt{<}a \cdot a\texttt{>}$, in this section we present the algorithm and definitions without neutral letters, that is, the structured alphabet is $\Sigma = (\Sigma^{\texttt{<}}, \Sigma^{\texttt{>}})$. Thus, from now on we use a for denoting any symbol in $\Sigma^{\texttt{<}} \cup \Sigma^{\texttt{>}}$.

Determinization of visibly pushdown automata. A significant result in Alur and Madhusudan's paper [4] that introduces VPA was that one can always determinize them. We provide here an alternative proof for this result that requires a somewhat more direct construction. This determinization process is behind our update algorithm and serves to give some crucial notions of how it works. We start by providing the determinization construction, introducing some useful notation, and then giving some intuition.

Given a VPA $\mathcal{A} = (Q, \Sigma, \Gamma, \Delta, I, F)$, we define the following deterministic VPA $\mathcal{A}^{\text{det}} = (Q^{\text{det}}, q_0^{\text{det}}, \Gamma^{\text{det}}, \delta^{\text{det}}, F^{\text{det}})$ with state set $Q^{\text{det}} = 2^{Q \times Q}$ and stack symbol set $\Gamma^{\text{det}} = 2^{Q \times \Gamma \times Q}$. The initial state is $q_0^{\text{det}} = \{(q,q) \mid q \in I\}$ and the set of final states is $F^{\text{det}} = \{S \in Q^{\text{det}} \mid S \cap (I \times F) \neq \emptyset\}$. Finally, we define the transition function δ^{det} such that if $\texttt{<}a \in \Sigma^{\texttt{<}}$, then $\delta^{\text{det}}(S, \texttt{<}a) = (S', T')$ where $S' = \{(q,q) \mid \exists p, p', \gamma.\ (p,p') \in S \wedge (p', \texttt{<}a, q, \gamma) \in \Delta\}$ and $T' = \{(p, \gamma, q) \mid \exists p'.\ (p,p') \in S \wedge (p', \texttt{<}a, q, \gamma) \in \Delta\}$; if $a\texttt{>} \in \Sigma^{\texttt{>}}$, then $\delta^{\text{det}}(S, T, a\texttt{>}) = S'$ where $S' = \{(p,q) \mid \exists p', q', \gamma.\ (p, \gamma, p') \in T \wedge (p', q') \in S \wedge (q', a\texttt{>}, \gamma, q) \in \Delta\}$.

To understand the purpose of this construction, first we need to introduce some notation. Fix a well-nested word $w = a_1 a_2 \cdots a_n$. A span s of w is a pair $[i,j\rangle$ of natural numbers i and j with $1 \leq i \leq j \leq n+1$. We denote by $w[i,j\rangle$ the subword $a_i \cdots a_{j-1}$ of w and, when $i = j$, we assume that $w[i,j\rangle = \varepsilon$. Intuitively, spans are indexing w with intermediate positions, like ${}_{\mathtt{1}}a_1{}_{\mathtt{2}}a_2{}_{\mathtt{3}} \ldots {}_{\mathtt{n}}a_n{}_{\mathtt{n+1}}$, where i is between symbols a_{i-1} and a_i. Then $[i,j\rangle$ represents an interval $\{i, \ldots, j\}$ that captures the subword $a_i \ldots a_{j-1}$.

Now, we say that a span $[i,j\rangle$ of w is well-nested if $w[i,j\rangle$ is well-nested. Note that ε is well-nested, so $[i,i\rangle$ is a well-nested span for every i. For a position $k \in [1, n+1]$, we define the *current-level span* of k, $\mathsf{currlevel}(k)$, as the well-nested span $[j,k\rangle$ such that $j = \min\{j' \mid [j',k\rangle \text{ is well-nested}\}$. Note that $[k,k\rangle$ is always well-nested and thus $\mathsf{currlevel}(k)$ is well defined. We also identify the *lower-level span* of k, $\mathsf{lowerlevel}(k)$, defined as $\mathsf{lowerlevel}(k) = \mathsf{currlevel}(j-1) = [i, j-1\rangle$ whenever $\mathsf{currlevel}(k) = [j,k\rangle$ and $j > 1$. In contrast to $\mathsf{currlevel}(k)$, $\mathsf{lowerlevel}(k)$ is not always well-defined given that it is "one level below" than $\mathsf{currlevel}(k)$ and this may not exist. More concretely, for $\mathsf{currlevel}(k) = [j,k\rangle$ and $\mathsf{lowerlevel}(k) = [i, j-1\rangle$, these spans will look as follows:

$$ {}_{\mathtt{1}}a_1{}_{\mathtt{2}}a_2{}_{\mathtt{3}} \cdots \texttt{<}a_{i-1}{}_{\mathtt{i}} \overbrace{a_i \ldots a_{j-2}}^{\mathsf{lowerlevel}(k)} {}_{\mathtt{j-1}} \texttt{<}a_{j-1}{}_{\mathtt{j}} \overbrace{a_j \ldots a_{k-1}}^{\mathsf{currlevel}(k)} \overset{\downarrow}{{}_{\mathtt{k}}} a_k \cdots {}_{\mathtt{n}}a_n{}_{\mathtt{n+1}} $$

As an example, consider the word ${}_{\mathtt{1}}({}_{\mathtt{2}}({}_{\mathtt{3}}){}_{\mathtt{4}}({}_{\mathtt{5}}({}_{\mathtt{6}}){}_{\mathtt{7}}){}_{\mathtt{8}}){}_{\mathtt{9}}$. The only well-nested spans besides the ones of the form $[i,i\rangle$ are $[1,9\rangle$, $[2,4\rangle$, $[2,8\rangle$, $[4,8\rangle$ and $[5,7\rangle$, therefore $\mathsf{currlevel}(8) = [2,8\rangle$, and $\mathsf{lowerlevel}(7) = [2,4\rangle$.

We are ready to explain the purpose of the determinization above. Let $w = a_1 a_2 \cdots a_n$ be a well-nested word and $\rho^{\text{det}} = (S_1, \tau_1) \xrightarrow{a_1} \ldots \xrightarrow{a_{k-1}} (S_k, \tau_k)$ be the (partial) run of $\mathcal{A}^{\text{det}}$ until some k. Furthermore, assume $\tau_k = T_k \cdot \tau$ for some $T_k \in \Gamma^{\text{det}}$ and $\tau \in (\Gamma^{\text{det}})^*$. The connection between ρ^{det} and the runs of $\mathcal{A}$ over $a_1 \ldots a_{k-1}$ is given by the following invariants:

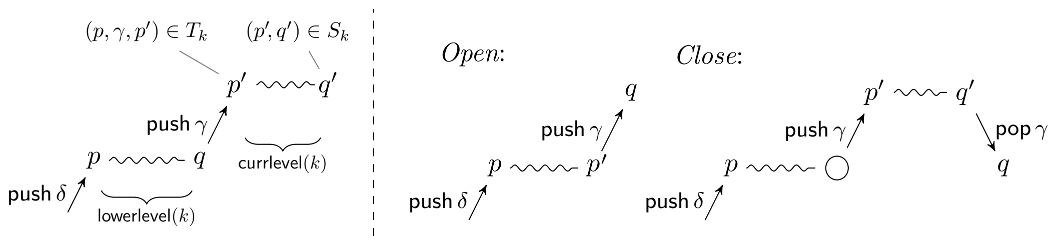

Figure 2 Left: An example run of some VPA $\mathcal{A}$ at step k. Right: Illustration of two nondeterministic runs for some VPA $\mathcal{A}$, as considered in the determinization process.

(a) $(p,q) \in S_k$ if, and only if, there exists a run $(q_1,\sigma_1) \xrightarrow{a_1} \dots \xrightarrow{a_{k-1}} (q_k,\sigma_k)$ of $\mathcal{A}$ over $a_1 \dots a_{k-1}$ such that $q_j = p$, $q_k = q$, and currlevel$(k) = [j,k\rangle$.

(b) $(p,\gamma,q) \in T_k$ if, and only if, there exists a run $(q_1,\sigma_1) \xrightarrow{a_1} \dots \xrightarrow{a_{k-1}} (q_k,\sigma_k)$ of $\mathcal{A}$ over $a_1 \dots a_{k-1}$ such that $q_i = p$, $q_j = q$, $\sigma_k = \gamma\sigma$ for some σ, and lowerlevel$(k) = [i,j-1\rangle$.

On one hand, (a) says that each pair $(p,q) \in S_k$ represents some non-deterministic run of $\mathcal{A}$ over w for which q is the k-th state, and p was visited on the step when the current symbol at the top of the stack was pushed. On the other hand, (b) says that $(p,\gamma,q) \in T_k$ represents some run of $\mathcal{A}$ over w for which γ is at the top of the stack, q was visited on the step when γ was pushed, and p was visited on the step when the symbol below γ was pushed (see Figure 2 (left)). More importantly, these conditions are exhaustive, that is, every run of $\mathcal{A}$ over $a_1 \dots a_{k-1}$ is represented by ρ^{det}.

By these two invariants, the correctness of $\mathcal{A}^{\text{det}}$ easily follows and the reader can get some intuition behind $\delta^{\text{det}}(S, {<}a)$ and $\delta^{\text{det}}(S,T,a{>})$ (see Figure 2 (right) for a graphical description). Indeed, the most important consequence of these two invariants is that a tuple $(q_j,q_k) \in S_k$ represents the interval of some run over $w[j,k\rangle$ with currlevel$(k) = [j,k\rangle$ and the tuple $(q_i,\gamma,q_j) \in T_k$ represents the interval of some run over $w[i,j-1\rangle$ with lowerlevel$(k) = [i,j-1\rangle$, i.e., the level below. In other words, the configuration (S_k,τ_k) of $\mathcal{A}^{\text{det}}$ forms a succinct representation of all the non-deterministic runs of $\mathcal{A}$. This is the starting point of our update algorithm, that we discuss next.

The streaming evaluation algorithm. In Algorithm 1 we present the update phase for solving the streaming version of EnumVPT. The main procedure is UpdatePhase, that receives an I/O-unambiguous VPT $\mathcal{T} = (Q,\Sigma,\Gamma,\Omega,\Delta,I,F)$ and a stream $\mathcal{S}$, reads the next k-th symbol and computes the set of outputs $[\![\mathcal{T}]\!](\mathcal{S}[1,k])$. More specifically, it constructs an ε-ECS $\mathcal{D}$ and a vertex v_{out} such that $\mathcal{L}_{\mathcal{D}}(v_{\text{out}}) = [\![\mathcal{T}]\!](\mathcal{S}[1,k])$ if $\mathcal{S}[1,k]$ is well-nested and $\emptyset$ otherwise. After the UpdatePhase procedure is done, we can enumerate $\mathcal{L}_{\mathcal{D}}(v_{\text{out}})$ with output-linear delay by calling the enumeration phase, that is, by applying Theorem 9.

Towards this goal, in Algorithm 1 we make use of the following data structures: First of all, we use an ε-ECS $\mathcal{D} = (\Sigma, V, I, \ell, r, \lambda)$, nodes $v \in V$, and the functions add, union, and prod over $\mathcal{D}$ and v (see Section 5). For the sake of simplification, we overload the notation of these operators slightly so that if $v = \emptyset$, then union$(\mathcal{D},v,v') = $ union$(\mathcal{D},v',v) = (\mathcal{D},v')$. We use a hash table S which indexes nodes v in $\mathcal{D}$ by pairs of states $(p,q) \in Q \times Q$. We denote the elements of S as "$(p,q) : v$" where (p,q) is the index and v is the content. Furthermore, we write $S_{p,q}$ to access the node v. We also use a stack T that stores hash tables: each element is a hash table which indexes vertices v in $\mathcal{D}$ by triples $(p,\gamma,q) \in Q \times \Gamma \times Q$. We assume that T has the standard stack methods push and pop where if $T = t_k \cdots t_1$, then push$(T,t) = t\, t_k \cdots t_1$ and pop$(T) = t_{k-1} \cdots t_1$. We write $\emptyset$ for denoting the empty stack or for checking if T is empty. Similarly to S, we use the notation $T_{p,\gamma,q}$ to access the nodes in the topmost hash-table in T (i.e. T is a stack of hash tables). We assume that accessing a non-assigned index in these hash tables returns the empty set. All variables (e.g., S, and T) are defined globally in Algorithm 1 and they can be accessed by any of the subprocedures. given that we use the RAM model (see Section 2), each operation over hash tables or stacks takes constant time.

Algorithm 1 builds the ε-ECS $\mathcal{D}$ incrementally, reading $\mathcal{S}$ one letter at a time by calling `yield`$[\mathcal{S}]$ and keeping a counter k for the position of the current letter. For every $k \in [1, n+1]$, UPDATEPHASE builds the k-th iteration of table S and stack T, which we note as S^k and T^k respectively. Before UPDATEPHASE is called for the first time, it runs INTIALIZE (lines 1-4) to set the initial values of k, $\mathcal{D}$, S, and T. We consider the initial S and T as the 1-st iteration, defined as $S^1 = \{(q,q) : v_\varepsilon \mid q \in I\}$ and $T^1 = \emptyset$ (i.e. the empty stack) where v_ε is a node in $\mathcal{D}$ such that $\mathcal{L}_\mathcal{D}(v_\varepsilon) = \{\varepsilon\}$ (lines 3-4). In the k-th iteration, depending on whether the current letter is an open symbol or a close symbol, the OPENSTEP or CLOSESTEP procedures are called, updating S^{k-1} and T^{k-1} to S^k and T^k, respectively. More specifically, UPDATEPHASE adds nodes to $\mathcal{D}$ such that the nodes in S^k represent the runs over $w[j,k\rangle$ where $\mathsf{currlevel}(k) = [j,k\rangle$, and the nodes in the topmost table in T^k represent the runs over $w[i,j-1\rangle$ where $\mathsf{lowerlevel}(k) = [i,j-1\rangle$. Moreover, for a given pair (p,q), the node $S^k_{p,q}$ represents all runs over $w[j,k\rangle$ with $\mathsf{currlevel}(k) = [j,k\rangle$ that start on p and end on q. For a given triple (p,γ,q) the node $T^k_{p,\gamma,q}$ represents all runs over $w[i,j-1\rangle$ with $\mathsf{lowerlevel}(k) = [i,j-1\rangle$ that start on p, and end on q right after pushing γ onto the stack. Here, the intuition gained in the determinization of VPA is crucial. Indeed, table S^k and stack T^k are the mirror of the configuration (S_k, τ_k) of $\mathcal{A}^{\text{det}}$ (recall invariants (a) and (b)).

Before formalizing these notions, we will describe in more detail what the procedures OPENSTEP and CLOSESTEP exactly do. Recall that the operation $\mathsf{add}(\mathcal{D}, a)$ simply creates a node in $\mathcal{D}$ labeled as a; the operation $\mathsf{prod}(\mathcal{D}, v_1, v_2)$ returns a pair $(\mathcal{D}', v')$ such that $\mathcal{L}_{\mathcal{D}'}(v') = \mathcal{L}_\mathcal{D}(v_1) \cdot \mathcal{L}_\mathcal{D}(v_2)$; and the operation $\mathsf{union}(\mathcal{D}, v_3, v_4)$ returns a pair $(\mathcal{D}', v')$ such that $\mathcal{L}_{\mathcal{D}'}(v') = \mathcal{L}_\mathcal{D}(v_3) \cup \mathcal{L}_\mathcal{D}(v_4)$. To improve the presentation of the algorithm, we include a simple procedure called IFPROD (lines 19-25). Basically, this procedure receives a node v, an output symbol $\mathcal{o}$, and a position k, and computes $(\mathcal{D}', v')$ such that $\mathcal{L}_{\mathcal{D}'}(v') = \mathcal{L}_\mathcal{D}(v) \cdot \{(\mathcal{o}, k)\}$ if $\mathcal{o} \neq \varepsilon$, and $\mathcal{L}_{\mathcal{D}'}(v') = \mathcal{L}_\mathcal{D}(v)$ otherwise.

In OPENSTEP, S^k is created (i.e. S'), and an empty table is pushed onto T^{k-1} to form T^k (line 27). Then, all nodes in S^{k-1} (i.e. S) are checked to see if the runs they represent can be extended with a transition in Δ (lines 28-29). If this is the case (lines 30 onwards), a node v_ε with the ε-output is added in S^k to start a new level (lines 30-32). Then, if the transition had a non-empty output, the node $S^k_{p,p'}$ is connected with a new label node to form the node v (lines 33-34). This node is stored in $T^k_{p,\gamma,q}$, or united with the node that was already present there (lines 35-36).

In CLOSESTEP, S^k is initialized as empty (line 41). Then, the procedure looks for all of the valid ways to join a node in T^{k-1}, a node in S^{k-1}, and a transition in Δ to form a new node in S^k. More precisely, it looks for quadruples (p,γ,p',q') for which $T^{k-1}_{p,\gamma,p'}$ and $S^{k-1}_{p',q'}$ are defined, and there is a close transition that starts on q' that reads γ (lines 42-43). These nodes are joined and connected with a new label node if it corresponds (lines 44-45), and stored in $S^k_{p,q}$ or united with the node that was already present there (lines 46-47). Finally, the top of the stack T is popped after all tuples (p,γ,p',q') are checked (line 48).

As it was already mentioned, in each step the construction of $\mathcal{D}$ follows the ideas of the determinization of a visibly pushdown automata. As such, Figure 2 also aids to illustrate how the table S^k and the top of the stack T^k are constructed.

Algorithm 1 The update phase of the streaming evaluation algorithm for EnumVPT given an I/O-unambiguous VPT $\mathcal{T} = (Q, \Sigma, \Gamma, \Omega, \Delta, I, F)$ and a stream $\mathcal{S}$.

```
 1: procedure Initialize($\mathcal{T}, \mathcal{S}$)
 2:     $k \leftarrow 1$, $\mathcal{D} \leftarrow \emptyset$
 3:     $(\mathcal{D}, v_\varepsilon) \leftarrow \mathsf{add}(\mathcal{D}, \varepsilon)$
 4:     $S \leftarrow \{(q,q) : v_\varepsilon \mid q \in I\}$, $T \leftarrow \emptyset$
 5:
 6: procedure UpdatePhase($\mathcal{T}, \mathcal{S}$)
 7:     $a \leftarrow \mathtt{yield}[\mathcal{S}]$
 8:     if $a \in \Sigma^{<}$ then
 9:         $\mathcal{D} \leftarrow$ OpenStep($\mathcal{D}, a, k$)
10:     else if $a \in \Sigma^{>}$ then
11:         $\mathcal{D} \leftarrow$ CloseStep($\mathcal{D}, a, k$)
12:     $k \leftarrow k + 1$
13:     $v_{\text{out}} \leftarrow \emptyset$
14:     if $T = \emptyset$ then
15:         for each $p \in I, q \in F$ s.t. $S_{p,q} \neq \emptyset$ do
16:             $(\mathcal{D}, v_{\text{out}}) \leftarrow \mathsf{union}(\mathcal{D}, v_{\text{out}}, S_{p,q})$
17:     EnumerationPhase($\mathcal{D}, v_{\text{out}}$)
18:
19: procedure IfProd($\mathcal{D}, v, ø, k$)
20:     if $ø \neq \varepsilon$ then
21:         $(\mathcal{D}', v') \leftarrow \mathsf{add}(\mathcal{D}, (ø, k))$
22:         $(\mathcal{D}', v') \leftarrow \mathsf{prod}(\mathcal{D}', v, v')$
23:     else
24:         $(\mathcal{D}', v') \leftarrow (\mathcal{D}, v)$
25:     return $(\mathcal{D}', v')$

26: procedure OpenStep($\mathcal{D}, {<}a, k$)
27:     $S' \leftarrow \emptyset$, $T \leftarrow \mathsf{push}(T, \emptyset)$
28:     for $p \in Q$ and $(p', {<}a, ø, q, \gamma) \in \Delta$ do
29:         if $S_{p,p'} \neq \emptyset$ then
30:             if $S'_{q,q} = \emptyset$ then
31:                 $(\mathcal{D}, v_\varepsilon) \leftarrow \mathsf{add}(\mathcal{D}, \varepsilon)$
32:                 $S'_{q,q} \leftarrow v_\varepsilon$
33:             $v \leftarrow S_{p,p'}$
34:             $(\mathcal{D}, v) \leftarrow$ IfProd($\mathcal{D}, v, ø, k$)
35:             $(\mathcal{D}, v) \leftarrow \mathsf{union}(\mathcal{D}, v, T_{p,\gamma,q})$
36:             $T_{p,\gamma,q} \leftarrow v$
37:     $S \leftarrow S'$
38:     return $\mathcal{D}$
39:
40: procedure CloseStep($\mathcal{D}, a{>}, k$)
41:     $S' \leftarrow \emptyset$
42:     for $p, p' \in Q$ and $(q', a{>}, ø, \gamma, q) \in \Delta$ do
43:         if $S_{p',q'} \neq \emptyset$ and $T_{p,\gamma,p'} \neq \emptyset$ then
44:             $(\mathcal{D}, v) \leftarrow \mathsf{prod}(\mathcal{D}, T_{p,\gamma,p'}, S_{p',q'})$
45:             $(\mathcal{D}, v) \leftarrow$IfProd($\mathcal{D}, v, ø, k$)
46:             $(\mathcal{D}, v) \leftarrow \mathsf{union}(\mathcal{D}, v, S'_{p,q})$
47:             $S'_{p,q} \leftarrow v$
48:     $T \leftarrow \mathsf{pop}(T)$
49:     $S \leftarrow S'$
50:     return $\mathcal{D}$
```

The way how the table S^k and the stack T^k are constructed is formalized in the following result. Recall that a run of $\mathcal{T}$ over a well-nested word $w = a_1 \cdots a_n$ is a sequence of the form $\rho = (q_1, \sigma_1) \xrightarrow{a_1/ø_1} \ldots \xrightarrow{a_n/ø_n} (q_{n+1}, \sigma_{n+1})$. Given a span $[i, j\rangle$, define a subrun of ρ as a subsequence $\rho[i, j\rangle = (q_i, \sigma_i) \xrightarrow{a_i/ø_i} \ldots \xrightarrow{a_{j-1}/ø_{j-1}} (q_j, \sigma_j)$. We also extend the function out to receive a subrun $\rho[i, j\rangle$ in the following way: $\mathsf{out}(\rho[i, j\rangle) = \mathsf{out}(ø_i, i) \cdot \ldots \cdot \mathsf{out}(ø_{j-1}, j-1)$. Finally, define $\text{Runs}(\mathcal{T}, w)$ as the set of all runs of $\mathcal{T}$ over w.

▶ **Lemma 10.** *Let $\mathcal{T}$ be a VPT and $w = a_1 \cdots a_n$ be a well-nested word. While running the procedure* UpdatePhase *of Algorithm 1, for every $k \in [1, n+1]$, every pair of states p, q and stack symbol γ the following hold:*

1. *$\mathcal{L}_{\mathcal{D}}(S^k_{p,q})$ has exactly all sequences $\mathsf{out}(\rho[j, k\rangle)$ such that $\rho \in \text{Runs}(\mathcal{T}, w[1, k\rangle)$, $\mathsf{currlevel}(k) = [j, k\rangle$, and $\rho[j, k\rangle$ starts on p and ends on q.*
2. *If $\mathsf{lowerlevel}(k)$ is defined, then $\mathcal{L}_{\mathcal{D}}(T^k_{p,\gamma,q})$ has exactly all sequences $\mathsf{out}(\rho[i, j\rangle)$ such that $\rho \in \text{Runs}(\mathcal{T}, w[1, j\rangle)$, $\mathsf{lowerlevel}(k) = [i, j-1\rangle$, and $\rho[i, j\rangle$ starts on p, ends on q, and the last symbol pushed onto the stack was γ.*

Since w is well nested, then $\mathsf{currlevel}(|w|+1) = [1, |w|+1\rangle$, and so, the lemma implies that the nodes in $S^{|w|+1}$ represent all runs of $\mathcal{T}$ over w. Then, whenever $\mathcal{S}[1, k]$ is well-nested, the stack T is empty (i.e., $T = \emptyset$) and there may be something to enumerate (line 14). By taking the union of all pairs in S^{k+1} that represent accepting runs (as is done in lines 15-16), we can conclude the following result:

▶ **Theorem 11.** *Given a VPT $\mathcal{T}$ and a stream $\mathcal{S}$,* UpdatePhase$(\mathcal{T}, \mathcal{S})$ *fulfils the conditions of a streaming evaluation algorithm and, after reading the k-th symbol, produces a pair $(\mathcal{D}, v_{\text{out}})$ such that $\mathcal{L}_{\mathcal{D}}(v_{\text{out}}) = [\![\mathcal{T}]\!](\mathcal{S}[1,k])$.*

At this point we address the fact that $\mathcal{D}$ needs to be unambiguous in order to enumerate all the outputs from $(\mathcal{D}, v_{\text{out}})$ without repetitions. This is guaranteed, essentially, by the fact that $\mathcal{T}$ is I/O-unambiguous as well. Indeed, the previous result holds even if $\mathcal{T}$ is not I/O-unambiguous. The next result guarantees that the output can be enumerated efficiently.

▶ **Lemma 12.** *Let $\mathcal{T}$ be an I/O-unambiguous VPT. While running* UpdatePhase *procedure of Algorithm 1, the ε-ECS $\mathcal{D}$ is unambiguous at every step.*

The complexity of this algorithm can be easily deduced from the fact that the ε-ECS operations we use take constant time (Theorem 9). For a VPT $\mathcal{T} = (Q, \Sigma, \Gamma, \Omega, \Delta, I, F)$, in each of the calls to OpenStep, lines 29-36 perform a constant number of instructions, and they are visited at most $|Q||\Delta|$ times. In each of the calls to CloseStep, lines 43-47 perform a constant number of instructions, and they are visited at most $|Q|^2|\Delta|$ times. Combined with Theorem 11, Lemma 12, and Theorem 9, this proves our main result (i.e. Theorem 3).

7 Future work

This paper offers several directions for future work. One direction is to find a streaming evaluation algorithm with polynomial update-time for non-deterministic VPT (i.e., in the size of the VPT). In [6], the authors provided a polynomial-time offline algorithm for non-deterministic word transducers (called vset automata). They extended this result to trees in [7]. One could use these techniques in Algorithm 1; however, it is unclear how to extend ECS to deal with ambiguity in a natural way. Regarding space resources, another direction is to find an "instance optimal" streaming evaluation algorithm for VPT. As we mentioned, this problem generalizes the weak evaluation problem stated in [45], given that it also considers the space to represent the output compactly. Finally, it would be interesting to explore practical implementations. Our view is that the data structure and algorithm presentation aid in reaching this goal, and it leaves space for suitable optimizations.

References

1 Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

2 Mehmet Altınel and Michael J Franklin. Efficient filtering of XML documents for selective dissemination of information. In *VLDB*, pages 53–64, 2000.

3 Rajeev Alur, Dana Fisman, Konstantinos Mamouras, Mukund Raghothaman, and Caleb Stanford. Streamable regular transductions. *Theor. Comput. Sci.*, 807:15–41, 2020.

4 Rajeev Alur and P. Madhusudan. Visibly pushdown languages. In *STOC*, pages 202–211, 2004.

5 Antoine Amarilli, Pierre Bourhis, Louis Jachiet, and Stefan Mengel. A circuit-based approach to efficient enumeration. In *ICALP*, pages 111:1–111:15, 2017.

6 Antoine Amarilli, Pierre Bourhis, Stefan Mengel, and Matthias Niewerth. Constant-delay enumeration for nondeterministic document spanners. In *ICDT*, pages 22:1–22:19, 2019.

7 Antoine Amarilli, Pierre Bourhis, Stefan Mengel, and Matthias Niewerth. Enumeration on trees with tractable combined complexity and efficient updates. In *PODS*, pages 89–103, 2019.

8 Marcelo Arenas, Luis Alberto Croquevielle, Rajesh Jayaram, and Cristian Riveros. Efficient logspace classes for enumeration, counting, and uniform generation. In *PODS*, pages 59–73, 2019.

9 Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *SIGMOD*, pages 1–16, 2002.

10 Guillaume Bagan. MSO queries on tree decomposable structures are computable with linear delay. In *CSL*, pages 167–181, 2006.

11 Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. On acyclic conjunctive queries and constant delay enumeration. In *CSL*, pages 208–222, 2007.

12 Ziv Bar-Yossef, Marcus Fontoura, and Vanja Josifovski. Buffering in query evaluation over XML streams. In *PODS*, pages 216–227, 2005.

13 Ziv Bar-Yossef, Marcus Fontoura, and Vanja Josifovski. On the memory requirements of XPath evaluation over XML streams. *J. Comput. Syst. Sci.*, 73(3):391–441, 2007.

14 Corentin Barloy, Filip Murlak, and Charles Paperman. Stackless processing of streamed trees. In *PODS*, 2021.

15 Christoph Berkholz, Fabian Gerhardt, and Nicole Schweikardt. Constant delay enumeration for conjunctive queries: a tutorial. *ACM SIGLOG News*, 7(1):4–33, 2020.

16 Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. Answering conjunctive queries under updates. In *PODS*, pages 303–318, 2017.

17 Pierre Bourhis, Juan L. Reutter, and Domagoj Vrgoc. JSON: Data model and query languages. *Inf. Syst.*, 89:101478, 2020.

18 Mathieu Caralp, Pierre-Alain Reynier, and Jean-Marc Talbot. Trimming visibly pushdown automata. *Theor. Comput. Sci.*, 578:13–29, 2015.

19 Yi Chen, Susan B. Davidson, and Yifeng Zheng. An efficient XPath query processor for XML streams. In *ICDE*, page 79, 2006.

20 Rada Chirkova and Jun Yang. Materialized views. *Found. Trends Databases*, 4(4):295–405, 2012.

21 Bruno Courcelle. Linear delay enumeration and monadic second-order logic. *Discret. Appl. Math.*, 157(12):2675–2700, 2009.

22 James R Driscoll, Neil Sarnak, Daniel Dominic Sleator, and Robert Endre Tarjan. Making data structures persistent. In *STOC*, pages 109–121, 1986.

23 Arnaud Durand and Etienne Grandjean. First-order queries on structures of bounded degree are computable with constant delay. *ACM Trans. Comput. Log.*, 8(4):21, 2007.

24 Emmanuel Filiot, Olivier Gauwin, Pierre-Alain Reynier, and Frédéric Servais. Streamability of nested word transductions. *LMCS*, 15(2), 2019.

25 Emmanuel Filiot, Jean-François Raskin, Pierre-Alain Reynier, Frédéric Servais, and Jean-Marc Talbot. Visibly pushdown transducers. *JCSS*, 97:147–181, 2018.

26 Fernando Florenzano, Cristian Riveros, Martín Ugarte, Stijn Vansummeren, and Domagoj Vrgoc. Efficient enumeration algorithms for regular document spanners. *TODS*, 45(1):3:1–3:42, 2020.

27 Olivier Gauwin, Joachim Niehren, and Yves Roos. Streaming tree automata. *Inf. Process. Lett.*, 109(1):13–17, 2008.

28 Olivier Gauwin, Joachim Niehren, and Sophie Tison. Bounded delay and concurrency for earliest query answering. In *LATA*, volume 5457, pages 350–361, 2009.

29 Olivier Gauwin, Joachim Niehren, and Sophie Tison. Earliest query answering for deterministic nested word automata. In *FCT*, volume 5699, pages 121–132, 2009.

30 Gang Gou and Rada Chirkova. Efficient algorithms for evaluating XPath over streams. In *SIGMOD*, pages 269–280. ACM, 2007.

31 Todd J. Green, Ashish Gupta, Gerome Miklau, Makoto Onizuka, and Dan Suciu. Processing XML streams with deterministic automata and stream indexes. *ACM Trans. Database Syst.*, 29(4):752–788, 2004.

32 Alejandro Grez and Cristian Riveros. Towards streaming evaluation of queries with correlation in complex event processing. In *ICDT*, pages 14:1–14:17, 2020.

33 Alejandro Grez, Cristian Riveros, and Martín Ugarte. A formal framework for complex event processing. In *ICDT*, pages 5:1–5:18, 2019.

34 Muhammad Idris, Martín Ugarte, and Stijn Vansummeren. The dynamic Yannakakis algorithm: Compact and efficient query processing under updates. In *SIGMOD*, pages 1259–1274, 2017.

35 Mark Jerrum, Leslie G. Valiant, and Vijay V. Vazirani. Random generation of combinatorial structures from a uniform distribution. *Theor. Comput. Sci.*, 43:169–188, 1986.

36 Vanja Josifovski, Marcus Fontoura, and Attila Barta. Querying XML streams. *VLDB J.*, 14(2):197–210, 2005.

37 Ahmet Kara, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. Trade-offs in static and dynamic evaluation of hierarchical queries. In *PODS*, pages 375–392, 2020.

38 Viraj Kumar, P. Madhusudan, and Mahesh Viswanathan. Visibly pushdown automata for streaming XML. In *WWW*, pages 1053–1062, 2007.

39 Leonid Libkin. *Elements of finite model theory*, volume 41. Springer, 2004.

40 Milos Nikolic and Dan Olteanu. Incremental view maintenance with triple lock factorization benefits. In *SIGMOD*, pages 365–380, 2018.

41 Dan Olteanu. SPEX: Streamed and progressive evaluation of XPath. *IEEE Trans. Knowl. Data Eng.*, 19(7):934–949, 2007.

42 Dan Olteanu, Tim Furche, and François Bry. An efficient single-pass query evaluator for XML data streams. In *SAC*, pages 627–631, 2004.

43 Dan Olteanu and Jakub Závodný. Size bounds for factorised representations of query results. *ACM TODS*, 40(1):2:1–2:44, 2015.

44 Luc Segoufin. Enumerating with constant delay the answers to a query. In *ICDT*, pages 10–20, 2013.

45 Luc Segoufin and Victor Vianu. Validating streaming XML documents. In *PODS*, pages 53–64, 2002.

46 Mirit Shalem and Ziv Bar-Yossef. The space complexity of processing XML twig queries over indexed documents. In *ICDE*, pages 824–832, 2008.

47 Balder ten Cate and Maarten Marx. Navigational XPath: calculus and algebra. *SIGMOD Record*, 36(2):19–26, 2007.

48 Szymon Torunczyk. Aggregate queries on sparse databases. In *PODS*, pages 427–443, 2020.