Streaming query evaluation with constant delay enumeration over nested documents

Martin Muñoz and Cristian Riveros
PUC & IMFD 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. With this in mind, 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 describes 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.

1 Introduction

Streaming query evaluation [3, 10] 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:

\[ \text{<doc> <a> <b/> <c/> <b/> </a> <c> <b/> <b/> </c> <doc> ...} \]

and for each document it has to evaluate the query \( Q = //a/b \) (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. [14]). 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 [24, 50]. 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
Streaming query evaluation with constant delay enumeration over nested documents

attracted researchers’ attention, finding sophisticated solutions to several query evaluation problems [12, 16, 11, 6, 28, 7].

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 $Q$ 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.

Our main result applies to the streaming evaluation of XML and JSON query languages. In the appendix, we also show an application in the context of document spanners [25].

Related work. The problem of streaming query evaluation has been extensively studied in the last decades. Some work considered streaming verification, like schema validation [51] or type-checking [12], where the output is true or false. Other proposals [20, 46, 10, 35, 45] provided streaming algorithms for XPath or XQuery’s fragments; however, extending them for reaching constant-delay enumeration seems unlikely. Furthermore, most of these works [42, 34, 33] assumed outputs of fixed size (i.e., tuples). People have also considered other aspects of streaming evaluation with outputs like earliest query answering [33] or bounded delay [32] (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 [13, 14], 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 [5] are a model usually used for streaming evaluation of boolean queries [42]. In [26, 1], authors studied the evaluation of VPT in a streaming fashion, but none of them saw enumeration problems. Other extensions [31] 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 [50], as we already discussed. In [11, 6], 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 [8], 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 [8] to get constant update-time. Streaming evaluation with constant-delay enumeration was included in the context of dynamic query evaluation [35, 17, 23, 31] or complex event processing [37, 36]. 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 ε ∈ Σ⁺ represents the empty word of length 0.

We will work over a structured alphabet Σ = (Σ⁺, Σ°, Σ¹) comprised of three disjoint sets Σ⁺, Σ°, and Σ¹ that contain open, close, and neutral symbols respectively (in [3] [27] these sets are named call, return, and local, respectively). Furthermore, we will denote symbols in Σ⁺, Σ° or Σ¹ by a, ø, and a, respectively. Instead, we will use s to denote any symbol in Σ⁺, Σ°, or Σ¹. The set of well-nested words over Σ, denoted as Σ⁺⁺⁺, is defined as the closure of the following rules: Σ⁺⁺⁺ = Σ⁺⁺⁺, if w₁, w₂ ∈ Σ⁺⁺⁺ and if w ∈ Σ⁺⁺⁺ and a ∈ Σ⁺ and b ∈ Σ° then a· w · b ∈ Σ⁺⁺⁺. In addition, we will work with prefixes of well-nested words, that we call prefix-nested words. We denote the set of prefixes of Σ⁺⁺⁺ as prefix(Σ⁺⁺⁺).

A stream $S = s₁s₂...$ is an infinite sequence where $s_i ∈ Σ⁺ ∪ Σ° ∪ Σ¹$. Given a stream $S = s₁s₂...$ and positions $i, j ∈ N$ such that $i ≤ j$, the word $S[i,j]$ is the sequence $s_is_{i+1}...sj$. We also use this notation to refer to subsequences of infinite sequences that are not composed of symbols in Σ. For a stream $S$, we will always assume that for each $i ∈ N$, the word $S[1,i]$ is a prefix of some nested word (i.e., it can be completed to form a nested word). We also consider a method yield($S$) which can be called to access each element of $S$ sequentially.

Visibly pushdown automata. A visibly pushdown automaton [5] (VPA) is a tuple $A = (Q, Σ, Γ, Δ, I, F)$ where $Q$ is a finite set of states, $Σ = (Σ⁺, Σ°, Σ¹)$ is the input alphabet, Γ is the stack alphabet, $Δ ⊆ (Q × Σ⁺ × Σ° × Γ) ∪ (Q × Σ° × Γ × Q) ∪ (Q × Σ¹ × Q)$ is the transition relation, $I ⊆ Q$ is a set of initial states, and $F ⊆ Q$ is a set of final states. A transition $(q, a, q′, γ)$ is a push-transition where on reading $a ∈ Σ⁺$, $γ$ is pushed onto the stack and the current state switches from $q$ to $q′$. Conversely, $(q, a, γ, q′)$ is a pop-transition where on reading $a ∈ Σ°$ 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 $a ∈ Σ¹$, 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 $w = s₁...s_n$ in Σ⁺⁺⁺, a run of $A$ on $w$ is a sequence $ρ = (q₁, σ₁) ↦ (q_{i+1}, σ_{i+1})$, where each $q_i ∈ Q$, $σ_i ∈ Γ⁺$, $q_1 = e$, and for each $i ∈ [1,n]$ the following holds: (1) if $s_i ∈ Σ⁺$, then there is $γ ∈ Γ$ such that $(q_i, s_i, q_{i+1}, γ) ∈ Δ$ and $σ_{i+1} = γσ_i$; (2) if $s_i ∈ Σ°$, then there is $γ ∈ Γ$ such that $(q_i, s_i, q_{i+1}, γ) ∈ Δ$ and $σ_{i+1} = γσ_i$; (3) if $s_i ∈ Σ¹$, then $(q_i, s_i, q_{i+1}) ∈ Δ$ and $σ_{i+1} = σ_i$. A run $ρ$ is accepting if $q_{n+1} ∈ F$. A well-nested word $w ∈ Σ⁺⁺⁺$ is accepted by a VPA $A$ if there is an accepting run of $A$ on $w$. The language $L(A)$ is the set of well-nested words accepted by $A$. Note that if $ρ$ is an accepting run of $A$ on a well-nested word $w$, then $σ_{n+1} = e$. A set of well-nested words $L ⊆ Σ⁺⁺⁺$ is called a visibly pushdown language if there exists a VPA $A$ such that $L = L(A)$.

A VPA $A = (Q, Σ, Γ, δ, I, F)$ is said to be deterministic if $|I| = 1$ and $δ$ is a function subset of $(Q × Σ⁺ ∪ Q × Σ° ∪ Q × Σ¹)$. We also say that $A$ is unambiguous if, for every $w ∈ L(A)$, there exists exactly one accepting run of $A$ on $w$. In [5], it is shown that for every VPA $A$ there exists an equivalent deterministic VPA of at most exponential size.

Model of computation. As it is common in the enumeration algorithms literature [11] [22] [50], for our algorithms we assume the computational model of Random Access Machines (RAM) with uniform cost measure, and addition and subtraction as basic operations. 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 [28]. This notion is a refinement of the definition of constant-delay [53] or linear-delay [22] 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 [39, 9].

First, we need to define what is an enumeration problem outside the stream setting. Let \( \Omega \) 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 \( [q]_R(x) = \{ y | ((q, x), y) \in R \} \) 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 \( (q, S) \) such that \( S \) is an infinite sequence of elements in \( \Omega \). 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 \( [q]_R(S[1,n]) \) at each position \( n \geq 1 \).

2. **A streaming \( \Delta \)-enumeration problem** for \( R \) is one where the objective is to enumerate the set \( [q]_R^\Delta(S[1,n]) = [q]_R(S[1,n]) \setminus \bigcup_{i < n} [q]_R(S[1,i]) \) 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 [21] and more recently, for dynamic query evaluation [35, 17]. For the sake of simplification, in the following we provide all definitions for the full-enumeration scenario. All definitions can be extended to \( \Delta \)-enumeration by changing \([q]_R\) to \([q]_R^\Delta\).

We turn now to our notion of efficiency for solving a streaming enumeration problem. Let \( f : \mathbb{N} \to \mathbb{N} \). We say that \( E \) is a streaming evaluation algorithm for \( R \) with \( f \)-update-time if \( E \) operates in the following way: it receives a query \( q \) and reads the stream \( S \) by calling the \texttt{yield}(\( S \)) method sequentially. After the \( n \)-th call to \texttt{yield}(\( S \)), 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 \([q]_R(S[1,n])\) 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 \( \Omega \), and \( y_1, y_2, \ldots, y_m \) is an enumeration (without repetitions) of the set \([q]_R(S[1,n])\), (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 \( E \)'s operation into an update and enumeration phase is to make an output-sensitive analysis of \( E \)'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 \( \text{time}_i(x) \) be the time in the enumeration phase when the algorithm writes the \( i \)-th \( \# \) when running on \( x \) for \( i \leq m + 1 \). Define \( \text{delay}_i(x) = \text{time}_{i+1}(x) - \text{time}_i(x) \) 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 \( \text{delay}_i(x) \leq k \cdot |y_i| \).

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 \( O(n \cdot f(|y|)) \) 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 \( O(g(|q|, |\mathcal{S}[1, n]|)) \).

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 \( \Delta \)-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 \( \Delta \)-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 [33], dynamic query evaluation [17, 38], and complex event processing [37, 50].

4 Visibly pushdown transducers and main result

In this section, we present the definition of visibly pushdown transducers [27] (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 \( \mathcal{T} = (Q, \Sigma, \Gamma, \Omega, \Delta, \ell, F) \) where \( Q, \Sigma, \Gamma, \ell, I, \) and \( F \) are the same as for VPA, \( \Omega \) is the output alphabet with \( \varepsilon \notin \Omega \), and \( \Delta \subseteq (Q \times \Sigma^* \times (\Omega \cup \{\varepsilon\}) \times Q \times \Gamma) \cup (Q \times \Sigma^* \times (\Omega \cup \{\varepsilon\}) \times \Gamma \times Q) \cup (Q \times \Sigma^1 \times (\Omega \cup \{\varepsilon\}) \times Q) \) is the transition relation. As usual for transducers, a symbol \( s \in \Sigma^* \cup \Sigma^1 \) 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, \( \varepsilon \) represents that no symbol is printed for that transition.

A run \( \rho \) of \( \mathcal{T} \) over a well-nested word \( w = s_1 s_2 \cdots s_n \in \Sigma^{***} \) is a sequence of the form \( \rho = (q_1, \sigma_1) \overset{s_1/o_1}{\longrightarrow} \cdots \overset{s_n/o_n}{\longrightarrow} (q_{n+1}, \sigma_{n+1}) \) 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 \( (q_i, s_i, \sigma_i, q_{i+1}, o) \in \Delta \) for some \( \gamma \in \Gamma \) and \( \sigma_{i+1} = \gamma \sigma_i \), (2) if \( s_i \in \Sigma^\ell \), then \( (q_i, s_i, \sigma_i, q_{i+1}) \in \Delta \) for some \( \gamma \in \Gamma \) and \( \sigma_i = \gamma \sigma_{i+1} \), and (3) if \( s_i \in \Sigma^1 \), then \( (p_i, s_i, \sigma_i, q_{i+1}) \in \Delta \) and \( \sigma_i = \sigma_{i+1} \). We say that the run is accepting if \( q_{n+1} \in F \). We call a pair \((q_i, \sigma_i)\) a configuration of \( \rho \). Finally, the output of an accepting run \( \rho \) is defined as: \( \text{out}(\rho) = \text{out}(q_1, 1) \cdot \cdots \cdot \text{out}(q_n, n) \) where \( \text{out}(\sigma, i) = \varepsilon \) when \( \sigma = \varepsilon \) and \( (\sigma, i) \) otherwise. Note that in \( \sigma_1 \cdots \sigma_n \) we use \( \varepsilon \) as a symbol, and in \( \text{out}(\rho) \) we use \( \varepsilon \) 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: \( \mathcal{T}(w) = \{ \text{out}(\rho) \mid \rho \) is an accepting run of \( \mathcal{T} \) over \( w \)\}.

Strictly speaking, our definition of VPT is richer than the one studied in [27]. In our definition of VPT each output element is a tuple \((\alpha, i)\) where \( \sigma \) is the symbol and \( i \) is the
Streaming query evaluation with constant delay enumeration over nested documents

output position, where for a standard VPT \([27]\) an output element is just the symbol \(\sigma\). The extension presented here is indeed important for practical applications like in document spanners [28, 7] or in XML query evaluation [13, 52].

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 documents. For the boolean case, it was shown [13] that VPA describe the same class of queries as MSO over nested words, called MSO\(_{\text{match}}\) (see the appendix). 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 [5] can be easily extended to link VPT with formulas expressible in MSO\(_{\text{match}}\). Given space restrictions, we give here an informal statement of this result (see the appendix).

\[\begin{align*}
\text{Proposition 1.} & \quad \text{Let } \varphi(X_1, \ldots, X_m) \text{ be a MSO}_{\text{match}} \text{ formula with } m \text{ free second-order variables } X_1, \ldots, X_m. \text{ There is a VPT } \mathcal{T} \text{ for which there is a one-to-one correspondence between the set } [\mathcal{T}] (w) \text{ and the set } \{(A_1, \ldots, A_m) \mid w \models \varphi(A_1, \ldots, A_m)\} \text{ for any word } w \in \Sigma^{\infty}. \\
& \quad \text{Moreover, for each VPT } \mathcal{T} \text{ there is an MSO}_{\text{match}} \text{ formula for which the same relation holds.}
\end{align*}\]

In other words, VPT has the same expressive power than MSO over nested words. Given that fragments of query languages over nested documents (e.g., navigational XPath [23], JSON Navigational Logic [13]) are usually included in MSO, this shows that VPT is an expressive formalism for query evaluation over nested documents. As an example, in the appendix we show how to translate some XPath queries into VPT, including \(Q\).

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 \(\Delta\) is a partial function of the form \(\Delta: (Q \times \Sigma \times \Omega \to Q \times \Gamma) \cup (Q \times \Sigma^* \times \Omega \times \Gamma \to Q) \cup (Q \times \Sigma^1 \times \Omega \to Q)\). On the other hand, we say that \(\mathcal{T}\) is input/output unambiguous (I/O-unambiguous for short) if for every \(w \in \Sigma^{\infty}\) and every \(\mu \in \llbracket \mathcal{T} \rrbracket (w)\) there is exactly one accepting run \(\rho\) of \(\mathcal{T}\) over \(w\) such that \(\mu = \text{out}(\rho)\). 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 [28] 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.

\[\begin{align*}
\text{Lemma 2.} & \quad \text{For every VPT } \mathcal{T} \text{ there exists an I/O-deterministic VPT } \mathcal{T}' \text{ of size } \mathcal{O}(2^{Q \llbracket \mathcal{T} \rrbracket}) \text{ such that } \llbracket \mathcal{T}' \rrbracket (w) = \llbracket \mathcal{T} \rrbracket (w) \text{ for every } w \in \Sigma^{\infty}. \\
& \quad \text{In this paper, we are interested on the following streaming enumeration problem for VPT.}
\end{align*}\]

Let \(\mathcal{C}\) be a class of VPT (e.g. I/O-deterministic VPT).

\begin{center}
\begin{tabular}{|c|}
\hline
\textbf{Problem:} & \textbf{ENUMVPT}\(\mathcal{C}\) \\
\textbf{Input:} & a VPT \(\mathcal{T} \in \mathcal{C}\) and \(w \in \Sigma^{\infty}\) \\
\textbf{Output:} & Enumerate \(\llbracket \mathcal{T} \rrbracket (w)\) \\
\hline
\end{tabular}
\end{center}

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.

\[\begin{align*}
\text{Theorem 3.} & \quad \text{The streaming full-enumeration problem of ENUMVPT for the class of I/O-unambiguous VPT can be solved with update-time } \mathcal{O}(|Q|^2 \Delta) \text{ and output-linear delay. For the general class of VPT, it can be solved with update-time } \mathcal{O}(2^{|Q| \Delta}) \text{ and output-linear delay.}
\end{align*}\]

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 [9]). 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 $\Delta$-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 $|\mathcal{T}'|(w) = |\mathcal{T}(w)| \cup \{ |\mathcal{T}'(w[1,i]) | \mid i < |w|, w[1,i] \in \Sigma^{\prec} \}$ for each $w \in \Sigma^{\prec}$.

**Theorem 4.** The streaming $\Delta$-enumeration problem of EnumVPT for the class of I/O-unambiguous VPT can be solved with update-time $O(|Q|^{2}|\Delta|)$ and output-linear delay. For the general class of VPT, it can be solved with update-time $O(|Q|^{2}|\Delta|)$ 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 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 $w \in \text{prefix}(\Sigma^{\prec})$. Let $\text{outputweight}(\mathcal{T}, w)$ be the number of positions less than $|w|$ that appear in some output of $[\mathcal{T}](w \cdot w')$ for some $w \cdot w' \in \Sigma^{\prec}$. Furthermore, for a well-nested word $u$ let $\text{depth}(u)$ be the maximum number of nesting pairs inside $u$, formally, $\text{depth}(a) = 0$ for $a \in \Sigma \cup \{ \varepsilon \}$, $\text{depth}(u_1 \cdot u_2) = \max\{ \text{depth}(u_1), \text{depth}(u_2) \}$, and $\text{depth}(\text{a} \cdot u \cdot \text{b}) = \text{depth}(u) + 1$. For $w \in \text{prefix}(\Sigma^{\prec})$, we define $\text{depth}(w) = \min\{ \text{depth}(w') \mid w \text{ is a prefix of } w' \}$. 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 $S$ requires at least $\Omega(\text{depth}(S[1,n]))$ bits of space.

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

In [13, 14], the authors provide lower bounds on the amount of space needed for evaluating XPath in terms of the nesting and the concurrency (see [13] 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 [19] if for every $w \in \text{prefix}(\Sigma^{\prec})$ and every (partial) run $\rho$ of $\mathcal{T}$ over $w$, there exists $w'$ and an accepting run $\rho'$ of $\mathcal{T}$ over $w \cdot w'$ such that $\rho$ is a prefix of $\rho'$. 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 [19] 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 $O(|Q|^{2}|\Delta|)$, output-linear delay and $O(\max\{ \text{depth}(S[1,n]), \text{outputweight}(\mathcal{T}, S[1,n]) \} \times |Q|^{2}|\Delta|)$ space for every stream $S$.

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 (see the appendix). Note
Streaming query evaluation with constant delay enumeration over nested documents

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 [51]. This is an open problem in the area (see [15] 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 [6, 7]. Indeed, ECS can be considered a refinement of the d-DNNF circuits used in [6] or of the set circuits used in [7]. Several papers [47, 8, 54] 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 [38, 17], 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 $\Sigma$ be a (possibly infinite) alphabet. We define an Enumerable Compact Set (ECS) as a tuple $D = (\Sigma, V, I, \ell, r, \lambda)$ such that $V$ and $I \subseteq V$ are finite sets of nodes, $\ell : I \rightarrow V$ and $r : I \rightarrow V$ are the left and right functions, and $\lambda : V \rightarrow \Sigma \cup \{\cup, \circ\}$ is a label function such that $\lambda(v) \in \{\cup, \circ\}$ if, and only if, $v \in I$. Further, we assume that the directed graph $(V, \{(v, \ell(v)), (v, r(v)) | v \in V\})$ 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 $\lambda(v) = \circ$, and a union node if $\lambda(v) = \cup$. We define the size of $D$ as $|D| = |V|$. For each node $v$ in $D$, we associate a set of words $L_D(v)$ recursively as follows: (1) $L_D(v) = \{a\}$ whenever $\lambda(v) = a \in \Sigma$, (2) $L_D(v) = L_D(\ell(v)) \cup L_D(r(v))$ whenever $\lambda(v) = \cup$, and (3) $L_D(v) = L_D(\ell(v)) \cdot L_D(r(v))$ whenever $\lambda(v) = \circ$, where $L_1 \cdot L_2 = \{w_1 \cdot w_2 | w_1 \in L_1$ and $w_2 \in L_2\}$.

The size $|L_D(v)|$ can be exponential with respect to $|D|$. For this reason, we say that $D$ is a compact representation of $L_D(v)$ for any $v \in V$. Although $L_D(v)$ 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 $D = (\Sigma, V, I, \ell, r, \lambda)$ and $v \in V$. |
| Output: | Enumerate the set $L_D(v)$ 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 $D$. The first restriction is to guarantee that $D$ is not ambiguous, namely, for each $w \in L_D(v)$ there is at most one way to retrieve $w$ from $D$. Formally, we say that $D$ is unambiguous if $D$ satisfies the following two properties: (1) for every union node $v$ it holds that $L_D(\ell(v))$ and $L_D(r(v))$ are disjoint, and (2) for every product node $v$ and for every $w \in L_D(v)$, there exists a unique way to decompose $w = w_1 \cdot w_2$.
such that \( w_1 \in \mathcal{L}_D(\ell(v)) \) and \( w_2 \in \mathcal{L}_D(r(v)) \). Thus, if \( D \) is unambiguous, there will be no duplicates if we enumerate \( \mathcal{L}_D(v) \) 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 \( D \), define the (left) output-depth of a node \( v \in V \), denoted by \( \text{odepth}_D(v) \), recursively as follows: \( \text{odepth}_D(v) = 0 \) whenever \( \lambda(v) \in \Sigma \) or \( \lambda(v) = \emptyset \), and \( \text{odepth}_D(v) = \text{odepth}_D(\ell(v)) + 1 \) whenever \( \lambda(v) = \cup \). Then, for a fixed \( k \in \mathbb{N} \) we say that \( D \) is \( k \)-bounded if \( \text{odepth}_D(v) \leq k \) for all \( v \in V \).

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

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

The enumeration algorithm above does not require any preprocessing over \( D \) 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 \( D \) while maintaining \( k \)-boundedness. Furthermore, we require these operations to be fully-persistent: a data structure is called \textit{fully-persistent} if every version can be both accessed and modified. 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 \( D \) to \( D' \) for each operation, by always adding new nodes and maintaining the previous nodes untouched. Then \( \mathcal{L}_D(v) = \mathcal{L}_D(v) \) for each node \( v \in V \), and so, the structure is fully-persistent.

More precisely, fix an ECS \( D = (\Sigma, V, I, \ell, r, \lambda) \) and \( D' = (\Sigma, V', I', \ell', r', \lambda') \) is an extension of \( D \) if, and only if, \( \text{obj} \subseteq \text{obj}' \) for every \( \text{obj} \in \{V, I, \ell, r, \lambda\} \). Further, we write \( \text{op}(I) \to O \) to define the signature of an operation \( \text{op} \) 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:

\[ \begin{align*}
\text{add}(D, a) & \to (D', v') \\
\text{prod}(D, v_1, v_2) & \to (D', v') \\
\text{union}(D, v_3, v_4) & \to (D', v')
\end{align*} \]

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

Next, we show how to implement each operation. In fact, the case of \text{add} and \text{prod} are straightforward. For \( \text{add}(D, a) \to (D', v') \) define \( V' := V \cup \{v'\}, \quad I' := I \cup \{v\}, \quad \lambda'(v') = a \). One can easily check that \( \mathcal{L}_{D'}(v') = \{a\} \) as expected. For \( \text{prod}(D, v_1, v_2) \to (D', v') \) we proceed in a similar way: define \( V' := V \cup \{v'\}, \quad I' := I \cup \{v\}, \quad \ell'(v') := v_1, \quad r'(v') := v_2, \) and \( \lambda'(v') = \emptyset \). Then \( \mathcal{L}_{D'}(v') = \mathcal{L}_D(v_1) \cdot \mathcal{L}_D(v_2) \). Furthermore, one can check that each operation takes constant time, \( D' \) is a valid ECS (i.e. unambiguous and \( k \)-bounded), and the operations are fully-persistent (i.e. the previous version \( D \) 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) \( \text{odepth}_D(v) \leq 1 \), and (2) if \( \text{odepth}_D(v) = 1 \), then \( \text{odepth}_D(r(v)) \leq 1 \). 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 \text{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 \( \text{union}(D, v_3, v_4) \to (D', v') \) and produce a safe node \( v' \). For this define \( (D', v') \) as follows:

- If \( v_3 \) or \( v_4 \) are output nodes then \( V' := V \cup \{v', u_1, u_2\} \), \( I' := I \cup \{v'\} \), and \( \lambda(v') := \emptyset \). 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 \( \lambda'(u_1) := \cup \).

This gadget is depicted in Figure 1 (note that a similar trick is used in 10 for computing an index over a circuit). This construction has several properties. First, one can easily check that \( L_{D'}(v') = L_D(v_1) \cup L_D(v_2) \) and so the semantics is well-defined. Second, union can be computed in constant time in \(|D|\) given that we only need to add three fresh nodes, and the operation is fully-persistent given that we connect them without modifying \( D \). Furthermore, the produced node \( v' \) is safe in \( D' \), although nodes \( u_1 \) and \( u_2 \) are not necessarily safe. Finally, \( D' \) is \( 2 \)-bounded whenever \( D \) 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 \( \text{odepth}_{D'}(v') = \text{odepth}_{D}(u_1) = 1 \). Further, given that \( v_3 \) is safe, we know that \( \text{odepth}_{D}(r(v_3)) \leq 1 \), so \( \text{odepth}_{D}(u_2) \leq 2 \). Given that the output depths of all fresh nodes in \( D' \) are bounded by 2 and \( D \) is \( 2 \)-bounded, then we conclude that \( D' \) is \( 2 \)-bounded as well.

By the previous discussion, if we start with an ECS \( D \) which is \( 2 \)-bounded (or empty) and we apply the add, \text{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, \text{prod}, and union require constant time and are fully-persistent. Furthermore, if we start from an empty ECS \( D \) and apply add, \text{prod}, and union over safe nodes, the partial results \( (D', v') \) satisfy that \( v' \) is always a safe node and the set \( L_{D'}(v) \) 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 \( \varepsilon \). 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 \( \varepsilon \) (called \( \varepsilon \)-ECS) as a tuple \( \mathcal{D} = (\Sigma, V, I, \ell, r, \lambda, \delta) \) defined identically to an ECS except that \( \varepsilon \) and \( \lambda(v) \in \{\emptyset, v\} \) 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 \( \varepsilon \)-ECS and one has to slightly extend \( k \)-boundedness to consider \( \varepsilon \)-nodes. However, to support the \text{prod} and \text{union} operations in constant time and to maintain the \( k \)-boundedness invariant, we need to extend the notion of safe nodes (called \( \varepsilon \)-safe) and the gadgets for \text{prod} and \text{union}. Given space restrictions, we show these extensions in the appendix and state here the main result, that will be used in the next section.

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

\section{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 plus a stream \( \mathcal{S} \), and enumerates the set \( \mathcal{T}(\mathcal{S}[1, n]) \) for an arbitrary \( n \geq 1 \) with \( O(\mid\mathcal{Q}\mid\mid\Delta\mid) \)-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 \( \langle a, a \rangle \), in this section we present the algorithm and definitions without neutral letters, that is, the structured alphabet is \( \Sigma = (\Sigma^\prime, \Sigma^\prime) \). Thus, from now on we use \( a \) for denoting any symbol in \( \Sigma^\prime \cup \Sigma^\prime \).

\textbf{Determinization of visibly pushdown automata.} A significant result in Alur and Madhusudan’s paper \cite{AM04} 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 \( \mathcal{A}^\text{det} = (Q^\text{det}, q^0, \Gamma, \delta^\text{det}, F^\text{det}) \) with state set \( Q^\text{det} = 2^Q \times Q \) and stack symbol set \( \Gamma^\text{det} = 2^\Gamma \times \Gamma \). The initial state is \( q^0 = \{(q, q) \mid q \in I\} \) 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 \( \delta^\text{det} \) such that if \( \alpha \in \Sigma^\prime \), then \( \delta^\text{det}(S, \alpha) = (S', T') \) where \( S' = \{(q, q) \mid \exists p, p', \gamma. (p, p') \in S \land (p', \alpha, q, \gamma) \in \Delta\} \) and \( T' = \{(p, \gamma, q) \mid \exists p'. (p, p') \in S \land (p', \alpha, q, \gamma) \in \Delta\} \). If \( \alpha \in \Sigma^\prime \), then \( \delta^\text{det}(S, \alpha) = S' \) where \( S' = \{(p, q) \mid \exists p', q', \gamma. (p, q, p') \in T \land (p', q') \in S \land (q', \alpha, \gamma, q) \in \Delta\} \).

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

Now, we say that a span \( [i, j] \) of \( w \) is well-nested if \( w[i, j] \) is well-nested. Note that \( \varepsilon \) is well-nested, so \( [i, i] \) is a well-nested span for every \( i \). For a position \( k \in [1, n + 1] \), we define the current-level span of \( k \), \( \text{curlevel}(k) \), as the well-nested span \( [j, k] \) such that \( j = \min\mid\{j' \mid \text{\{j', k\} is well-nested}\} \). Note that \( [k, k] \) is always well-nested and thus \( \text{curlevel}(k) \) is well...
12 Streaming query evaluation with constant delay enumeration over nested documents

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

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

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

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 \( (p, q, \gamma) \in T_k \) represents some run of \( \mathcal{A} \) over \( w \) for which \( \gamma \) is at the top of the stack, \( q \) was visited on the step when \( \gamma \) was pushed, and \( p \) was visited on the step when the symbol below \( \gamma \) was pushed (see Figure 2 (left)). More importantly, these conditions are exhaustive, that is, every run of \( \mathcal{A} \) over \( a_1 \ldots a_{k-1} \) is represented by \( \rho^{\det} \).

By these two invariants, the correctness of \( \mathcal{A}^{\det} \) easily follows and the reader can get some intuition behind \( \delta^{\det}(S, \sigma a) \) and \( \delta^{\det}(S, T, a) \) (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 \( w[j, k] \) with \( \text{curvlevel}(k) = [j, k] \) and the tuple \( (q_i, \gamma, q_j) \in T_k \) represents the interval of some run over \( w[i, j-1] \) with \( \text{lowlevel}(k) = [i, j-1] \), i.e., the level below. In other words, the configuration \( (S_k, \tau_k) \) of \( \mathcal{A}^{\det} \) 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 \( T = (Q, \Sigma, \Gamma, \Omega, \Delta, I, F) \) and a stream \( S \), reads the next
k-th symbol and computes the set of outputs \([T](S[1,k])\). More specifically, it constructs an \(\varepsilon\)-ECS \(D\) and a vertex \(v_{out}\) such that \(L_D(v_{out}) = [T](S[1,k])\) if \(S[1,k]\) is well-nested and \(\emptyset\) otherwise. After the UpdatePhase procedure is done, we can enumerate \(L_D(v_{out})\) 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 \(\varepsilon\)-ECS \(D = (\Sigma, V, I, \ell, r, \lambda)\), nodes \(v \in V\), and the functions add, union, and \(\text{prod}\) over \(D\) and \(v\) (see Section 2). For the sake of simplification, we overload the notation of these operators slightly so that if \(v = \emptyset\) then union \((D, v, v') = \text{union}(D, v', v) = (D, v')\). We use a hash table \(S\) which indexes nodes \(v\) in \(D\) 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 \(D\) by triples \((p, \gamma, q) \in Q \times \Gamma \times Q\). We assume that \(T\) has the standard stack methods \(\text{push}\) and \(\text{pop}\) where if \(T = t_k \cdots t_1\), then \(\text{push}(T, t) = t t_k \cdots t_1\) and \(\text{pop}(T) = t_k \cdots t_1\). 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 sub procedures, given that we use the RAM model (see Section 2), each operation over hash tables or stacks takes constant time.

Algorithm 1 builds the \(\varepsilon\)-ECS \(D\) incrementally, reading \(S\) one letter at a time by calling \(\text{yield}(S)\) 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 Initialze (lines 1-4) to set the initial values of \(k\), \(D\), \(S\), and \(T\). We consider the initial \(S\) and \(T\) as the 1-st iteration, defined as \(S^1 = \{(q, q) : v_c \mid q \in I\}\) and \(T^1 = \emptyset\) (i.e. the empty stack) where \(v_c\) is a node in \(D\) such that \(L_D(v_c) = \{\varepsilon\}\) (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 \(D\) such that the nodes in \(S^k\) represent the runs over \(w[j,k]\) where \(\text{curlevel}(k) = [j,k]\), and the nodes in the topmost table in \(T^k\) represent the runs over \(w[i,j-1]\) where \(\text{lowerlevel}(k) = [i,j-1]\). Moreover, for a given pair \((p,q)\), the node \(S^k_{p,q}\) represents all runs over \(w[j,k]\) with \(\text{curlevel}(k) = [j,k]\) that start on \(p\) and end on \(q\). For a given triple \((p, \gamma, q)\) the node \(T^k_{p, \gamma, q}\) represents all runs over \(w[i,j-1]\) with \(\text{lowerlevel}(k) = [i,j-1]\) that start on \(p\), and end on \(q\) right after pushing \(\gamma\) 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 \((S_k, \tau_k)\) of \(A^{\text{det}}\) (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 add\((D, a)\) simply creates a node in \(D\) labeled as \(a\); the operation \(\text{prod}(D, v_1, v_2)\) returns a pair \((D', v')\) such that \(L_{D'}(v') = L_{D'}(v_1) \cup L_{D'}(v_2)\); and the operation union \((D, v_3, v_1)\) returns a pair \((D', v')\) such that \(L_{D'}(v') = L_{D'}(v_3) \cup L_{D'}(v_1)\). To improve the presentation of the algorithm, we include a simple procedure called IFFProd (lines 19-25). Basically, this procedure receives a node \(v\), an output symbol \(e\), and a position \(k\), and computes \((D', v')\) such that \(L_{D'}(v') = L_{D'}(v) \cdot \{(e, k)\}\) if \(e \neq \varepsilon\), and \(L_{D'}(v') = L_{D'}(v)\) 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 \(\Delta\) (lines 28-29). If this is the case (lines 30 onwards), a node \(v_c\) with the \(\varepsilon\)-output is added in \(S^k\) to start a new level (lines 30-32). Then,
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 $S$.

1: procedure Initialize($\mathcal{T}, S$)
2: $k \leftarrow 1$, $D \leftarrow \emptyset$
3: $(D, v_s) \leftarrow \text{add}(D, \varepsilon)$
4: $S \leftarrow \{(q, q) : v \in I\}$, $T \leftarrow \emptyset$
5: 6: procedure UpdatePhase($\mathcal{T}, S$)
7: $a \leftarrow \text{yield}[S]$
8: if $a \in \Sigma^*$ then
9: $D \leftarrow \text{OpenStep}(D, a, k)$
10: else if $a \in \Sigma^+$ then
11: $D \leftarrow \text{CloseStep}(D, a, k)$
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: $(D', v_{\text{out}}) \leftarrow \text{union}(D, D_{p,q}, S_{p,q})$
17: EndProcedure $\text{EnumerationPhase}(D, v_{\text{out}})$
18: 19: procedure IfProd($D, v, a, k$)
20: if $a \neq \varepsilon$ then
21: $(D', v') \leftarrow \text{add}(D, (a, k))$
22: $(D', v') \leftarrow \text{prod}(D, v, v')$
23: else
24: $(D', v') \leftarrow (D, v)$
25: return $(D', v')$
26: procedure OpenStep($\mathcal{D}, a, k$)
27: $S' \leftarrow \emptyset$, $T \leftarrow \text{push}(T, \emptyset)$
28: for $p \in Q$ and $(p', a, q, q', \gamma) \in \Delta$ do
29: if $S_{p,p'} \neq \emptyset$ then
30: if $S_{q,q'} = \emptyset$ then
31: $(D, v) \leftarrow \text{IfProd}(D, v, a, k)$
32: $(D, v) \leftarrow \text{union}(D, v, T_{p,q,q'})$
33: $T_{p,q,q'} \leftarrow v$
34: return $\mathcal{D}$
35: 36: procedure CloseStep($\mathcal{D}, a, k$)
37: $S' \leftarrow S$
38: return $\mathcal{D}$
39: 40: for $p, p' \in Q$ and $(q', a, q, q', \gamma) \in \Delta$ do
41: if $S_{p,p'} \neq \emptyset$ and $T_{p,q,q'} \neq \emptyset$ then
42: $(D, v) \leftarrow \text{IfProd}(D, v, a, k)$
43: $(D, v) \leftarrow \text{union}(D, v, S_{p,q'})$
44: $S_{p,q'} \leftarrow v$
45: return $\mathcal{D}$
46: 47: $T \leftarrow \text{pop}(T)$
48: 49: return $\mathcal{D}$
50: 51: 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 $T^k_{p,q,q'}$, 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 $\Delta$ to form a new node in $S^k$. More precisely, it looks for quadruples $(p, q', q')$ for which $T^k_{p,q,q'}$ and $S^k_{p,q'}$ are defined, and there is a close transition that starts on $q'$ that reads $\gamma$ (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 $(p, q, p', q')$ are checked (line 48).

As it was already mentioned, in each step the construction of $\mathcal{D}$ follows the ideas of the deterministic 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.

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 $w = a_1 \ldots a_n$ is a sequence of the form $\rho = (q_1, \sigma_1) \xrightarrow{a_1/\sigma_1} \ldots \xrightarrow{a_n/\sigma_n} (q_{n+1}, \sigma_{n+1})$. Given a span $[i, j]$, define a subrun of $\rho$ as a subsequence $\rho[i,j] = (q_i, \sigma_i) \xrightarrow{a_{i+1}/\sigma_i} \ldots \xrightarrow{a_{j}/\sigma_{j-1}} (q_j, \sigma_j)$. We also extend the function out to receive a subrun $\rho[i,j]$ in the following way: $\text{out}(\rho[i,j]) = \text{out}(a_i, i) \ldots \text{out}(a_{j-1}, j - 1)$. 
Finally, define $\text{Runs}(T, w)$ as the set of all runs of $T$ over $w$.

\begin{lemma}
Let $T$ be a VPT and $w = a_1 \cdots a_n$ be a well-nested word. While running the procedure $\text{UpdatePhase}$ of Algorithm \[7\] for every $k \in [1, n + 1]$, every pair of states $p, q$ and stack symbol $\gamma$ the following hold:
\begin{enumerate}
\item $\mathcal{L}_D(S_k) \setminus [S_{k+1}]$ has exactly all sequences $\text{out}(\rho[i, j])$ such that $\rho \in \text{Runs}(T, w[1, k])$, $\text{currlevel}(k) = [j, k]$, and $\rho[i, j]$ starts on $p$ and ends on $q$.
\item If $\text{lowerlevel}(k)$ is defined, then $\mathcal{L}_D(T_k^{[\rho[i, j]]})$ has exactly all sequences $\text{out}(\rho[i, j])$ such that $\rho \in \text{Runs}(T, w[1, j])$, $\text{lowerlevel}(k) = [i, j - 1]$, and $\rho[i, j]$ starts on $p$, ends on $q$, and the last symbol pushed onto the stack was $\gamma$.
\end{enumerate}
\end{lemma}

Since $w$ is well-nested, then $\text{currlevel}([w] + 1) = [1, |w| + 1]$, and so, the lemma implies that the nodes in $S^{[w+1]}$ represent all runs of $T$ over $w$. Then, whenever $S[1, k]$ is well-nested, the stack $T$ is empty (i.e., $T = \epsilon$) 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:

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

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

\begin{lemma}
Let $T$ be an I/O-unambiguous VPT. While running $\text{UpdatePhase}$ procedure of Algorithm \[7\], the $\varepsilon$-ECS $D$ is unambiguous at every step.
\end{lemma}

The complexity of this algorithm can be easily deduced from the fact that the $\varepsilon$-ECS operations we use take constant time (Theorem \[9\]). For a VPT $T = (Q, \Sigma, \Gamma, \Omega, \Delta, I, F)$, in each of the calls to $\text{OpenStep}$, lines 29-36 perform a constant number of instructions, and they are visited at most $|Q| |\Delta|$ times. In each of the calls to $\text{CloseStep}$, 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 \[3\]).

\section{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 \[7\], the authors provided a polynomial-time offline algorithm for non-deterministic word transducers (called vsf automata). They extended this result to trees in \[8\]. One could use these techniques in Algorithm \[11\] however, it is unclear how to extend ECS to deal with ambiguity naturally. 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 \[51\], 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 Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. *Compilers, principles, techniques*. Addison wesley, 7(8):9, 1986.
3 Mehmet Altinel and Michael J Franklin. Efficient filtering of xml documents for selective dissemination of information. In *VLDB*, pages 53–64, 2000.
4 Rajeev Alur, Dana Fisman, Konstantinos Mamouras, Mukund Raghothaman, and Caleb Stanford. Streamable regular transductions. *Theor. Comput. Sci.*, 807:15–41, 2020.
5 Rajeev Alur and P. Madhusudan. Visibly pushdown languages. In *STOC*, pages 202–211, 2004.
6 Antoine Amarilli, Pierre Bourhis, Louis Jachiet, and Stefan Mengel. A circuit-based approach to efficient enumeration. In *ICALP*, pages 111:1–111:15, 2017.
7 Antoine Amarilli, Pierre Bourhis, Stefan Mengel, and Matthias Niewerth. Constant-delay enumeration for nondeterministic document spanners. In *ICDT*, pages 22:1–22:19, 2019.
8 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.
9 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.
10 Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *SIGMOD*, pages 1–16, 2002.
11 Guillaume Bagan. MSO queries on tree decomposable structures are computable with linear delay. In *CSL*, pages 167–181, 2006.
12 Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. On acyclic conjunctive queries and constant delay enumeration. In *CSL*, pages 208–222, 2007.
13 Ziv Bar-Yossef, Marcus Fontoura, and Vanja Josifovski. Buffering in query evaluation over XML streams. In *PODS*, pages 216–227, 2005.
14 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.
15 Corentin Barloy, Filip Murlak, and Charles Paperman. Stackless processing of streamed trees. In *PODS*, 2021.
16 Christoph Berkholz, Fabian Gerhardt, and Nicole Schweikardt. Constant delay enumeration for conjunctive queries: a tutorial. *ACM SIGLOG News*, 7(1):4–33, 2020.
17 Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. Answering conjunctive queries under updates. In *PODS*, pages 303–318, 2017.
18 Pierre Bourhis, Juan L. Reutter, and Domagoj Vrgoc. JSON: data model and query languages. *Inf. Syst.*, 89:101478, 2020.
19 Mathieu Caralp, Pierre-Alain Reynier, and Jean-Marc Talbot. Trimming visibly pushdown automata. *Theoretical Computer Science*, 578:13–29, 2015.
20 Yi Chen, Susan B. Davidson, and Yifeng Zheng. An efficient xpath query processor for XML streams. In *ICDE*, page 79.
21 Rada Chirkova and Jun Yang. Materialized views. *Found. Trends Databases*, 4(4):295–405, 2012.
22 Bruno Courcelle. Linear delay enumeration and monadic second-order logic. *Discrete Applied Mathematics*, 157(12):2675–2700, 2009.
23 James R Driscoll, Neil Sarnak, Daniel Dominic Sleator, and Robert Endre Tarjan. Making data structures persistent. In *STOC*, pages 109–121, 1986.
24 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.
25 Ronald Fagin, Benny Kimelfeld, Frederick Reiss, and Stijn Vansummeren. Document spanners: A formal approach to information extraction. *J. ACM*, 62(2):12:1–12:51, 2015.
Emmanuel Filiot, Olivier Gauwin, Pierre-Alain Reynier, and Frédéric Servais. Streamability of nested word transductions. *LMCS*, 15(2), 2019.

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

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.

Dominik D. Freydenberger. A logic for document spanners. *Theory Comput. Syst.*, 63(7):1679–1754, 2019.

Dominik D. Freydenberger, Benny Kimelfeld, and Liat Peterfreund. Joining extractions of regular expressions. In *PODS*, pages 137–149, 2018.

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

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

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

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

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.

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

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

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

Mark R Jerrum, Leslie G Valiant, and Vijay V Vazirani. Random generation of combinatorial structures from a uniform distribution. *Theoretical computer science*, 43:169–188, 1986.

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

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.

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

Francisco Maturana, Cristian Riveros, and Domagoj Vrgoc. Document spanners for extracting incomplete information: Expressiveness and complexity. In *PODS*, pages 125–136, 2018.

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

Dan Olteanu. SPLEX: streamed and progressive evaluation of xpath. *IEEE Trans. Knowl. Data Eng.*, 19(7):934–949, 2007.

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

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

Liat Peterfreund. Grammars for document spanners. In *ICDT*, pages 7:1–7:18, 2021.

Liat Peterfreund, Balder ten Cate, Ronald Fagin, and Benny Kimelfeld. Recursive programs for document spanners. In *ICDT*, pages 13:1–13:18, 2019.

Luc Segoufin. Enumerating with constant delay the answers to a query. In *ICDT*, pages 10–20, 2013.
18 Streaming query evaluation with constant delay enumeration over nested documents

51 Luc Segoufin and Victor Vianu. Validating streaming XML documents. In PODS, pages 53–64, 2002.
52 Mirit Shalem and Ziv Bar-Yossef. The space complexity of processing XML twig queries over indexed documents. In ICDE, pages 824–832, 2008.
53 Balder ten Cate and Maarten Marx. Navigational xpath: calculus and algebra. SIGMOD Record, 36(2):19–26, 2007.
54 Szymon Torunczyk. Aggregate queries on sparse databases. In PODS, pages 427–443, 2020.
A Proofs from Section 4

A.1 Proof of Proposition 1

To precisely state the proposition, we need to show how we encode words as logical structures. This is done, as usual, with an order predicate and unary predicates to represent the order and the letters of each positions of the word, respectively. More formally, fix a structured alphabet $\Sigma$ and let $w \in \Sigma^\infty$ be a word of length $n$. We encode $w$ as a structure $([1, n], \leq, (P_a)_{a \in \Sigma})$ where $[1, n]$ is the domain, $\leq$ is the total order over $[1, n]$, and $P_a = \{ i \mid w[i] = a \}$. We also have a binary relation match over $[1, n]$ that corresponds to the matching relation of open and close symbols: $\text{match}(i, j)$ is true iff $w[i]$ is an open symbol and $w[j]$ is its matching close symbol. By some abuse of notation, we also use $w$ to denote its corresponding logical structure. A MSO$_{\text{match}}$ formula $\varphi$ over $\Sigma$ is given by:

$$\varphi := P_a(x) \mid x \in X \mid x \leq y \mid \text{match}(x, y) \mid \neg \varphi \mid \varphi \lor \psi \mid \exists x. \varphi \mid \exists X. \varphi$$

where $a \in \Sigma$, $x$ and $y$ are first-order variables and $X$ is a monadic second order variable (i.e. a set variable). We write $\varphi(\bar{x}, \bar{X})$ where $\bar{x}$ and $\bar{X}$ are the sets of free first-order and monadic second-order variables of $\varphi$, respectively. An assignment $\sigma$ for $w$ is a function $\sigma : \bar{x} \cup \bar{X} \rightarrow 2^{[1, n]}$ such that $|\sigma(x)| = 1$ for every $x \in \bar{x}$ (note that we treat first-order variables as a special case of monadic second-order variables). As usual, we denote by $\text{dom}(\sigma) = \bar{x} \cup \bar{X}$ the domain of the function $\sigma$. Then we write $(w, \sigma) \models \varphi$ when $\sigma$ is an assignment over $w$, $\text{dom}(\sigma) = \bar{x} \cup \bar{X}$, and $w$ satisfies $\varphi(\bar{x}, \bar{X})$ when each variable in $\bar{x} \cup \bar{X}$ is instantiated by $\sigma$. Given a formula $\varphi(\bar{x}, \bar{X})$, we define $[\varphi](w) = \{ \sigma \mid (w, \sigma) \models \varphi(\bar{x}, \bar{X}) \}$. For the sake of simplification, from now on we will only use $\bar{X}$ to denote the free variables of $\varphi(\bar{X})$ and use $X \in \bar{X}$ for an first-order or monadic second-order variable.

For any assignment $\sigma$ over $w$, we define the support of $\sigma$, denoted by $\text{supp}(\sigma)$, as the set of positions mentioned in $\sigma$; formally, $\text{supp}(\sigma) = \{ i \mid \exists v \in \text{dom}(\sigma) \text{ s.t. } i \in \sigma(v) \}$. Furthermore, we encode assignments as sequences over the support as follows: Let $\text{supp}(\sigma) = \{ i_1, \ldots, i_m \}$ such that $i_j < i_{j+1}$ for every $j < m$. Then, we define the (word) encoding of $\sigma$ as:

$$\text{enc}(\sigma) = (\bar{X}_1, i_1)(\bar{X}_2, i_2)\ldots(\bar{X}_m, i_m)$$

such that $\bar{X}_j = \{ X \in \text{dom}(\sigma) \mid i_j \in \sigma(X) \}$ for every $j \leq m$. That is, we represent $\sigma$ as an increasing sequence of positions, where each position is labeled with the variables of $\sigma$ where it belongs.

The statement of the proposition can be formulated as follows:

**Proposition 13 (Proposition 1).** Fix a structured alphabet $\Sigma$. Let $\bar{X}$ be a set of MSO variables and $\mathcal{X} = 2^\bar{X}$.

1. For any MSO$_{\text{match}}$ formula $\varphi(\bar{X})$ there exists a VPT $T$ with output alphabet $\mathcal{X}$ such that:

$$[T](w) = \{ \text{enc}(\sigma) \mid \sigma \in [\varphi](w) \}.$$

2. For any VPT $T$ with output alphabet $\mathcal{X}$ there exists a MSO$_{\text{match}}$ formula $\varphi(\bar{X})$ such that:

$$\{ \text{enc}(\sigma) \mid \sigma \in [\varphi](w) \} = [T](w).$$

The proof of this proposition is largely based on the proof of Theorem 4 in [5]. To prove (1) we can follow the exact same argument as the if direction of the proof and be left with a VPA $\mathcal{A}$ over the input alphabet $\Sigma^X = \Sigma \times \mathcal{X}$ whose language is the set of words which encode a valuation $\sigma$ of $\bar{X}$ along with a word $w$ for which $(w, \sigma) \models \varphi(\bar{X})$. We define a straightforward
A VPT that translates this query is shown in Figure 4.

The VPT in this section are shown graphically using the following notation: An open transition \((p, s, o, \gamma, q)\) is represented by an edge from \(p\) to \(q\) with the label \(s\) if \(o = \varepsilon\) and with the label \(s/\gamma\) if \(o \neq \varepsilon\). A close transition \((p, s, \varepsilon, \gamma, q)\) is represented with the label \(s\gamma\). As is customary, we extend this notation by representing multiple transitions that differ only by their input symbol as a single transition over the set of these symbols. We also use the symbol \(\mid\) to group transitions that start and end in the same states.

As a first example, consider the XPath query \(Q_1 = //a/b\). This query can be translated into the VPT shown in Figure 3.

As a more involved example, consider the following XPath query over the tree alphabet \(\{a, b, c\}\):

\[ Q_2 = \text{child}:a/\text{descendant}:b[\text{following-sibling}:c] \]

A VPT that translates this query is shown in Figure 4.
A.3 Proof of Lemma 2

Let $\mathcal{T} = (Q, \Sigma, \Gamma, \Omega, \Delta, I, F)$. We will construct an input-output deterministic VPT $\mathcal{T}' = (Q', \Sigma', \Gamma', \Omega, \delta_{\text{det}}, S_I, F')$ as follows: Let $Q' = 2^{Q \times Q}$ and $\Gamma' = 2^{Q \times \Sigma}$. Let $S_I = \{(q, q) \mid q \in I\}$ and let $F' = \{S \mid (p, q) \in S \text{ for some } p \in I \text{ and } q \in F\}$. Let $\delta$ be defined as follows:

- For $a \in \Sigma^*$ and $\sigma \in \Omega$, $\delta(S, a, \sigma) = (S', T)$, where:
  
  $$ T = \{(p, \gamma, q) \mid (p, p') \in S \text{ and } (p', a, \sigma, \gamma, q) \in \Delta \text{ for some } q \in Q\}, $$
  
  $$ S' = \{(q, q) \mid (p, p') \in S \text{ and } (p', a, \sigma, \gamma, q) \in \Delta \text{ for some } p, p' \in Q \text{ and } q \in \Gamma\} $$

- For $a \in \Sigma^*$ and $\sigma \in \Omega$, $\delta(S, a, \sigma, T) = S'$ where, if $T \subseteq Q \times \Gamma \times Q$, then:
  
  $$ S' = \{(p, q) \mid (p, \gamma, p') \in T \text{ and } (p', q') \in S \text{ and } (q', a, \sigma, q) \in \Delta \text{ for some } p', q' \in Q, \gamma \in \Gamma\}, $$

- For $a \in \Sigma^1$ and $\sigma \in \Omega$, $\delta(S, a) = S'$ where:
  
  $$ S' = \{(q, q') \mid (q, q') \in S \text{ and } (q', a, \sigma, q'') \in \Delta \text{ for some } q' \in Q\}. $$

One can immediately check that this automaton is input-output deterministic since the transition relation is modelled as a partial function.

We will prove that $\mathcal{T}$ and $\mathcal{T}'$ are equivalent by induction on well-nested words. To aid our proof, we will introduce a couple of ideas. First, we extend the definition of a run to include sequences that start on an arbitrary configuration. Also, given a run
Streaming query evaluation with constant delay enumeration over nested documents

\[ \rho = (q_1, \sigma_1) \xrightarrow{s_1/\mu_1} (q_2, \sigma_2) \xrightarrow{s_2/\mu_2} \ldots \xrightarrow{s_n/\mu_n} (q_{n+1}, \sigma_{n+1}), \]
and a span \([i,j]\), define a subrun of \(\rho\) as the subsequence

\[ \rho[i,j] = (q_i, \sigma_i) \xrightarrow{s_i/\mu_i} (q_{i+1}, \sigma_{i+1}) \xrightarrow{s_{i+1}/\mu_{i+1}} \ldots \xrightarrow{s_{j-1}/\mu_{j-1}} (q_j, \sigma_j). \]

In this proof, we only consider subruns such that \(w[i,j] = s_is_{i+1} \ldots s_{j-1}\) is a well-nested word. A second definition we will use is that of a VPT with arbitrary initial states. Formally, let \(S \subseteq Q\). We define \(T_q\) as the VPT that simulates \(T\) by starting on the configuration \((q, \varepsilon)\). Note that for a run \(\rho = (q_1, \sigma_1) \xrightarrow{s_1/\mu_1} \ldots \xrightarrow{s_n/\mu_n} (q_{n+1}, \sigma_{n+1})\) of \(T\) over \(w = s_1 \ldots s_n\) and a well-nested span \([i,j]\), the subrun \(\rho[i,j]\) is one of the runs of \(T_q\) over \(w[i,j]\) modulo \(\sigma_i\), which is present in all of the stacks in \(\rho\) as a common suffix.

We shall prove first that \([T](w) \subseteq [T'](w)\) for every well-nested word \(w\). This is done with the aid of the following result:

\[ \triangleright \text{Claim 14.} \] For a well-nested word \(w\), output sequence \(\mu\), states \(p, q \subseteq Q\), and a set \(S\) that contains \((p, q)\), if there is a run of \(T_q\) over \(w\) and \(\mu\) such that its last state is \(q'\), the (only) run of \(T_S\) over \(w\) and \(\mu\) ends in a state \(S'\) which contains \((p, q')\).

**Proof.** We will prove the claim by induction on \(w\).

If \(w = \varepsilon\), the proof is trivial since \(q = q'\). If \(w = a \in \Sigma^1\) the proof follows straightforwardly from the construction of \(\delta\).

If \(w, \upsilon \in \Sigma^{\ast\ast}\), and \(\mu, \kappa \in \Omega^\ast\), let \(p, q \in Q\), let \(S\) be a set that contains \((p, q)\), and let \(\rho\) be a run of \(T_q\) over \(w\upsilon\) and \(\mu\kappa\), which ends in a state \(q'\). Our goal is to prove that the run \(\rho'\) of \(T_S\) over \(w\upsilon\) and \(\mu\kappa\) ends in a state that contains \((p, q')\). Let \(n = |w|, m = |\upsilon|,\) and let \(w'\) be the last state of the subrun \(\rho[1, n + 1]\). Consider as well \(\rho'(n + 1, n + m + 1)\), which is a run of \(T_{\upsilon'}\) over \(\upsilon\) and \(\kappa\) that ends in \(q'\). From the hypothesis two conditions follow: (1) In the run of \(T_S\) over \(w\) and \(\mu\) the last state \(S'\) contains \((p, q'\upsilon)\), and (2) in the run of \(T_{\upsilon'}\) over \(\upsilon\) and \(\kappa\) the last state contains \((p, q')\). It can be seen that \(\rho'\) is the concatenation of these two runs, so this proves the claim.

If \(w \in \Sigma^{\ast\ast}, a \in \Sigma^\ast, b \in \Sigma^\ast, \mu \in \Omega^\ast,\) and \(\alpha_1, \alpha_2 \in \Omega,\) let \(p, q \in Q\), let \(S\) be a set that contains \((p, q)\), and let \(\rho\) be a run of \(T_q\) over \(awa\upsilon b\) and \(\alpha_1\mu\alpha_2\), let \(n = |w|,\) and let \(q_1, q_2, \ldots, q_{n+2}, q_{n+3}\) be the states of \(\rho\) in order. Our goal is to prove that the run \(\rho'\) of \(T_S\) over \(awa\upsilon b\) and \(\alpha_1\mu\alpha_2\) ends in a state that contains \((p, q_{n+3})\). Let \((q_2, x)\) be the second configuration of \(\rho\). This implies that \((q, a, a, q_1, q_2, \gamma) \in \Delta\) and \((q_{n+2}, b, a_2, \gamma, q_{n+3}) \in \Delta\). Let \(S'\) and \(T\) be such that \(\delta(S, a, a_1) = (S', T)\). Therefore, \((q_2, q_2) \in S'\) and \((p, \gamma, q_2) \in T\). Consider the subrun \(\rho[2, n + 2]\), which is a run of \(T_{q_2}\) over \(w\) and \(\mu\) that ends in \(q_{n+2}\) modulo the stack suffix \(\gamma\). Since \((q_2, q_2) \in S'\), from the hypothesis it follows that the run of \(T_{\upsilon'}\) over \(w\) and \(\mu\) ends in a state \(S''\) that contains \((q_2, q_{n+2})\). This run starts on the configuration \((S', \varepsilon)\) and ends in \((S'', \varepsilon)\), so a run on the same automaton that starts on \((S', T)\) and reads the same symbols will end in \((S'', T)\), which is the case for the subrun \(\rho'[2, n + 2]\). Therefore, the construction of \(\delta\) implies that \((p, q_{n+3})\) is contained in the last state of \(\rho'\), which proves the claim.

Let now \(w\) be a well-nested word and \(\mu\) be an output sequence such that \(T\) accepts \((w, \mu)\). Let \(\rho\) be an accepting run of \(T\) over \((w, \mu)\) which starts on a state \(p \in I\) and ends in a state \(q \in F\). Note that \(T_p\) also accepts \((w, \mu)\). Note that \(T = T_{S_I}\), and since \((p, p) \in S_I\) the claim implies that the run of \(T\) over \((w, \mu)\) ends in a state which contains \((p, q)\), and so this run is accepting. This proves that \([T](w) \subseteq [T'](w)\).

To prove that \([T](w) \subseteq [T'](w)\) we use a similar result:
Claim 15. For a well-nested word \( w \), output sequence \( \mu \), states \( q, p, q' \subseteq Q \), and a set \( S \) that contains \( (p, q) \), if the run of \( T_S \) over \( w \) and \( \mu \) ends on a state \( S' \) that contains \( (p, q') \), then there is a run of \( T_q \) over \( w \) and \( \mu \) such that its last state is \( q' \).

Proof. We will prove the claim by induction on \( w \).

If \( w = \varepsilon \), the proof is trivial since \( q = q' \). If \( w = a \in \Sigma^1 \) the proof follows straightforwardly from the construction of \( \delta \).

If \( w, v \in \Sigma^{<\omega} \), and \( \rho, \kappa \in \Omega^* \), let \( p, q, q' \in Q \), let \( S \) be a set that contains \( (p, q) \), and let \( \rho \) be the run of \( T_S \) over \( wv \) and \( \mu \kappa \), which ends in a state \( S' \) that contains \( (p, q') \). Our goal is to prove that there is a run \( \rho' \) of \( T_q \) over \( wv \) and \( \mu \kappa \) such that its last state is \( q' \). Let \( S' = \text{last state of the subrun } \rho \) and \( \kappa \) that ends in \( S' \). From the construction of \( \delta \), it is clear that if a non-empty state \( S' \) follows from \( S \) in a run of \( T \), then \( S \) is not empty. Let \( (p, q') \in S_{wv} \). From the hypothesis two conditions follow: (1) There is a run \( \rho_1 \) of \( T_q \) over \( w \) and \( \mu \) such that its last state is \( q' \). We then construct \( \rho' \) by concatenating \( \rho_1 \) and \( \rho_2 \) which ends in \( q' \), and this proves the claim.

If \( w \in \Sigma^{<\omega} \), \( q, a \in \Sigma^* \), \( b \in \Sigma^* \), \( \rho \in \Omega^* \), and \( q_1, q_2 \in Q \), let \( \rho \) be the run of \( T_S \) over \( awb \) and \( q_1 \mu \rho_2 \). Let \( n = |w| \), let \( S, S_2, \ldots, S_{n+2}, S_{n+3} \) be the states of \( \rho \) in order, and suppose there is a pair \( (p, q') \in S_{n+3} \). Our goal is to prove that there is a run \( \rho' \) of \( T_q \) over \( awb \) and \( \rho_1 \mu \rho_2 \) that ends in \( q' \). We will prove the claim by induction on \( w \).

Let \( S_2 \), \( T \) be the second configuration of \( \rho \). From the construction of \( \delta \), there exist \( q_2, q_{n+2} \in Q \) and \( z \in \Gamma \) such that \( (q_{n+2}, b, q_2, \gamma, q_{n+3}) \in \Delta \) and \( (p, q_2) \in T \). Since \( w \) is well-nested, \( T \) could only have been pushed after reading \( a/o_1 \), which implies that \( (q_2, a, q_1, q_2, \gamma) \in \Delta \). This, in turn, means that \( (q_2, q_2) \in S_2 \). Let \( \rho' \) be the subrun \( \rho'[2, n+2] \), which is a run of \( T_{S_{n+2}} \) over \( w \) and \( \mu \) that ends in \( S_{n+1} \) and \( S_{n+2} \) modulo the common stack suffix \( T \). We then have that \( (q_2, q_2) \in S_2 \) and \( (q_2, q_{n+2}) \in S_{n+2} \), and, from the hypothesis, it follows that there is a run \( \rho'' \) of \( T_{S_{n+2}} \) over \( w \) and \( \mu \) such that its last state is \( q_{n+2} \). In a similar fashion as in the previous claim, we modify the run slightly to obtain one that starts and ends on the stack \( \gamma \). This new run can be easily extended with the transitions \( (q_2, a, q_1, q_2, \gamma), (q_{n+2}, b, q_2, \gamma, q_{n+3}) \in \Delta \), and as a result, we obtain a run \( \rho' \) of \( T_q \) that fulfills the conditions of the claim.

Let now \( w \) be a well nested word and let \( \mu \) be an output sequence such that \( T \) accepts \( (w, \mu) \). Since \( T' = T_{S_1} \), and the run of \( T' \) over \( (w, \mu) \) ends in a state \( S \in F' \), we have that \( S \) contains an element \( (p, q) \) such that \( p \in I \) and \( q \in F' \). Moreover, \( (p, p) \in S_1 \). From the previous claim, it follows that there is an accepting run of \( T_q \) over \( (w, \mu) \) such that its last state is \( q \). Therefore, \( T \) accepts \( (w, \mu) \). This proves that \( T'(w) \subseteq T(w) \).

We conclude that \( T(w) \subseteq T'(w) \) for every well-nested word \( w \).

A.4 Proof of Theorem 4

The proof of the theorem is a consequence of the following lemma.

Lemma 16. For every I/O-unambiguous VPT \( T \) there exists an I/O-unambiguous VPT \( T' \) such that \( T'(w) = T(w) \cup \bigcup_{i<w} T(w[i, i]) \) for every \( w \in \Sigma^{<\omega} \). Furthermore, the size of \( T' \) is linear on the size of \( T \).
24 Streaming query evaluation with constant delay enumeration over nested documents

Let $\mathcal{T} = (Q, \Sigma, \Gamma, \Omega, \Delta, I, F)$ be an I/O-unambiguous VPT. We construct a VPT $\mathcal{T}' = (Q', \Sigma, \Gamma, \Omega, \Delta', I, F')$ such that $Q' = Q \times \{1, 2\}$, $I' = I \times \{1\}$, $F' = F \times \{1\}$ and $\Delta'$ is as follows:

$$\Delta' = \{((p, 1), \alpha, \omega, (q, 1), \gamma) \mid \alpha \in \Sigma^* \text{ and } (p, \alpha, \omega, q, \gamma) \in \Delta \text{ where either } \omega \in \Omega \text{ or } p \notin F\} \cup \{((p, 1), \alpha, \varepsilon, (q, 1), \gamma) \mid \alpha \in \Sigma^* \text{ and } (p, \alpha, \varepsilon, q, \gamma) \in \Delta \text{ where } p \in F\} \cup \{((p, 2), \alpha, \varepsilon, (q, 2), \gamma) \mid \alpha \in \Sigma^* \text{ and } (p, \alpha, \varepsilon, q, \gamma) \in \Delta \text{ where } \alpha \in \Omega\} \cup \{((p, 2), \alpha, \varepsilon, (q, 2), \gamma) \mid \alpha \in \Sigma^* \text{ and } (p, \alpha, \varepsilon, q, \gamma) \in \Delta\}.\]$$

This construction was only shown for symbols in $\Sigma^*$, but it should include analogous constructions for symbols in $\Sigma^+$ and $\Sigma^0$, which are omitted for convenience. The idea behind this construction is to separate the VPT in two halves. Each run starts on the first half (marked 1) and once it reaches a final state, it changes into the second half (marked 2). The run then stays on the second half until it sees an output symbol, with which it returns to the first half.

To show that $[\mathcal{T}'](w) = [\mathcal{T}]([w[1,i]])$ consider a $w \in \Sigma^{\ast\ast}$. Let $\mu$ be an output in $[\mathcal{T}]([w])$ and consider an accepting run $\rho'$ such that $\text{out}(\rho') = \mu$. We can construct an accepting run $\rho$ of $\mathcal{T}$ over $w$ by starting from $\rho'$ and replacing any appearance of a state $(q,k)$ by $q$. From this it follows that $\mu \in [\mathcal{T}]([w[1,i]])$ for some $i < |w|$. From the construction of $\Delta'$ it can be seen that the $i$-th and following states in $\rho'$ are of the form $(q,2)$, as all of the following transitions in $\rho'$ have $\varepsilon$ as their output symbols. Therefore, $\rho'$ cannot be an accepting run, and we reach a contradiction, from which we conclude that $\mu \notin [\mathcal{T}]([w[1,i]])$. Let $\mu$ now be an output in $\mu \in [\mathcal{T}]([w]) \setminus \bigcup_{i \in [w]}[\mathcal{T}]([w[1,i]])$ and let $\rho$ be the accepting run of $\mathcal{T}$ over $w$ such that $\text{out}(\rho) = \mu$. It can be seen from the construction of $\Delta'$ that the run of $\mathcal{T}$ over $w$ is identical to $\rho$ except each state $q$ in $\rho$ appears as $(q,k)$ in $\rho'$. We will show that the last state in $\rho'$ is of the form $(q,2)$. Towards a contradiction, assume that it is not. Therefore, in $\rho'$ there is a transition where the first state is of the form $(p,1)$ and the second is of the form $(q,2)$, and furthermore, every transition following this one has $\varepsilon$ as its output symbol. Let $i$ be the step where this happens. From the construction of $\Delta$ we see that the $i$-th state is in $F$, from which it follows that the run $\rho'_i$ built from the first $i$ steps in $\rho'$ is an accepting run of $\mathcal{T}'$ over $w[1,i]$ and that $\text{out}(\rho'_i) = \mu$. We can do a similar process as a above and construct an accepting run of $\mathcal{T}$ over $w[1,i]$ that renders the same output $\mu$, which contradicts our assumption that $\mu \notin \bigcup_{i \in [w]}[\mathcal{T}]([w[1,i]])$. We conclude that $[\mathcal{T}']([w]) = [\mathcal{T}]([w]) \setminus \bigcup_{i \in [w]}[\mathcal{T}]([w[1,i]])$.

To show that $\mathcal{T}'$ is unambiguous, consider a $w \in \Sigma^{\ast\ast}$. Let $\mu \in [\mathcal{T}']([w])$ and consider two accepting runs $\rho_1$ and $\rho_2$ such that $\text{out}(\rho_1) = \text{out}(\rho_2) = \mu$. Let us build a run $\rho$ of $\mathcal{T}$ over $w$ as in the previous part of the proof, which is the same for $\rho_1$ and $\rho_2$ since $\mathcal{T}$ is unambiguous. This implies that both $\rho_1$ and $\rho_2$ contain the same sequence of states in $Q$. Suppose now that the runs are different, which is only possible if at some step $i$, the $i$-th state in $\rho_1$ and $\rho_2$ are the same, and in the $(i+1)$-th state in $\rho_1$ and $\rho_2$ are different. This cannot be the case since from the construction of $\mathcal{T}'$, for a given transition $t \in \Delta$ that starts in a state $p$, and some $k \in \{1,2\}$, there exists exactly one transition $t' \in \Delta'$ that starts in $(p,k)$. This is a contradiction, so we prove that $\mathcal{T}'$ is unambiguous.

### A.5 Proof of Proposition 5

**Part 1.** This proof is a corollary of Theorem 4.5 in [14]. The proof of this result implies that for the XQuery query $Q = \text{//a[b and c]}$, any streaming algorithm that verifies if an XML document matches $Q$ (the problem $\text{BOOLEVAL}_Q$) and any integer $r \geq 1$, there exists
a document of depth at most \( r + C \), where \( C \) is a constant value, on which the algorithm requires \( \Omega(r) \) bits of space.

Our proof will show a VPA \( A \) which can simulate the query \( Q \) for a direct mapping \( \nu \) of the documents that are constructed in [14], where \( \nu((a)) = a, \nu((/a)) = a, \nu((/b)) = b, \) and \( \nu((/c)) = c \). The VPA is shown in Figure 5. We convert this VPA into a VPT \( T \) by adding an \( \varepsilon \) output symbol on each transition, so the problem of deciding if \( A \) accepts \( w \) is equivalent to deciding if \( \mathcal{T}(w) \) is empty, or the set \( \{ \varepsilon \} \). The theorem follows by taking this \( T \) as the one in the statement, considering an arbitrary streaming evaluation algorithm \( E \) that solves \( \text{EnumVPT} \) with input \( T \), and using this algorithm along with the mapping \( \nu \) to solve \( \text{Booleval}Q \).

**Part 2.** This proof uses the main ideas of the proof of Theorem 1 in [13]. Here, the authors describe a set-computing communication complexity problem. In the problem \( \mathcal{P} \), Alice and Bob compute a two-argument function \( p(\cdot, \cdot) \), defined as follows. Alice’s input is a subset \( A \subseteq \{1, \ldots, k\} \), Bob’s input is a bit \( b \in \{0, 1\} \), and \( p(A, b) \) is defined to be \( a \), if \( b = 1 \), and \( /a \) otherwise. Proposition 1 in [13] proves that the one-way communication complexity of \( \mathcal{P} \) is at least \( k \).

Let \( T = (Q, \Sigma, \Gamma, \Omega, \Delta, I, F) \) is defined over the alphabets \( \Sigma = \{a, b, x\} \), and \( \Omega = \{x\} \) and have its sets \( Q, \Delta, I, F \) be as presented in Figure 6. It can be seen that it satisfies

\[
[T](w) \begin{cases} 
\{(x, i) \mid w[i] = b\} & \text{if } w \text{ ends in } \$ \\
\emptyset & \text{otherwise.}
\end{cases}
\]

Consider an arbitrary algorithm \( E \) that solves \( \text{EnumVPT} \) with input \( T \). We will now present a reduction that creates a protocol for \( \mathcal{P} \) which makes use of the algorithm \( E \). Here, Alice receives the set \( A \) and generates a word \( w \) of size \( k \) such that \( w[i] = b \) if \( i \in A \) and \( w[i] = a \) otherwise. Alice then executes \( E \) on input \( T \) and \( w \) as the first \( k \) characters of a stream. She sends the state of the algorithm to Bob, who receives the bit \( b \), and does the
following: If \( b = 1 \) he continues running \( E \) as if the last character of the input was \$. If \( b = 0 \), he stops executing \( E \) immediately. In either case, the output given by \( E \) contains all the information necessary to compute the set \( p(A, b) \), so the reduction is correct. This proves that \( E \) requires at least \( k \) bits for an input of size less than \( k \), and so \( E \) for any \( n \geq 1 \), requires at least \( n \) bits of space in a worst-case stream \( S \), which is in \( \Omega(outputweight(T, S[1, n])) \).

### A.6 Proof of Proposition 6

The time bounds are implied by Theorem 3 so we will prove the space bounds. The algorithm has a update phase and an enumeration phase, and the enumeration phase only processes the data structure that was built on the update phase, using at most linear extra space, as is explained in Section 5. As such, we will prove that Algorithm 1 on input \((T, w)\) uses \( \mathcal{O}(\text{depth}(w) + \text{outputweight}(T, w)) \times |Q|^2 \Delta | \) space at every point in its execution, which implies the statement of the proposition, where \( w = S[1, n] \) for some stream \( S \) and \( n \).

As it is explained in Section 6, Algorithm 1 uses a hash table \( S \), and a stack \( T \) that stores hash tables. The size of the stack at each point is bounded \( \text{depth}(w) \), and the size of each hash table is bounded by \( |Q|^2 |I| \), so the size of \( S \) and \( T \) combined is in \( \mathcal{O}(\text{depth}(w) |Q|^2 \Delta) \). The rest of the space used is related to the \( \varepsilon\)-ECS \( D \), which we will now bound by \( \mathcal{O}(\text{outputweight}(T, w[1, k]) |Q|^2 \Delta) \) at each step \( k \).

For every step \( k \) of the algorithm, consider an \( \varepsilon\)-ECS \( \mathcal{D}_{\varepsilon}^k \) which is composed solely of the nodes that are reachable from of the ones stored in \( S^k \), or the ones stored in some hash table in \( T^k \) (borrowing the notation from Section 6). A simple induction argument on \( k \) shows that the rest of the nodes in \( D \) can be discarded with no effect over the correctness of the algorithm, so they are not considered in the memory used by it. Therefore, proving that at each step \( |\mathcal{D}_{\varepsilon}^k| \in \mathcal{O}(\text{outputweight}(T, w[1, k]) |Q|^2 \Delta) \) is enough to complete the proof.

Let \( I \) be the set of positions less than \( k \) that appear in some output of \( T(w[1, k] \cdot w') \) for some \( w \cdot w' \in \text{prefix}(\Sigma^*\varepsilon) \). We now refer to Lemma 10 since it implies that for each node \( v \) stored in \( S^k \) or the topmost hash table in \( T^k \), each sequence in \( \mathcal{L}_D(v) \) corresponds to at least one valid run of \( T \) over \( w[1, k] \), and since \( T \) is trimmed, each one of these runs is part of an accepting run of \( T \) over \( w[1, k] \cdot w' \), for some word \( w' \). Therefore, each of the positions that appear in some of these sets is in \( I \). Furthermore, we can use this lemma to characterize the positions in the rest of the hash tables in \( T^k \), since appending any close symbol \( a_\varepsilon \) to \( w[1, k] \) will make the algorithm pop an element from \( T \), which will make the next hash table the topmost. This argument can be extended to any of the hash tables in \( T^k \), so in all, Lemma 10 implies that all of the positions that appear in some non-empty leaf in \( \mathcal{D}_{\varepsilon}^k \) are in \( I \). Theorem 3 implies that the set of these positions corresponds exactly to \( I \), since if there was any position in \( I \) missing from the leaves in \( D \), the algorithm would not be correct.

Lastly, we will show that \( |\mathcal{D}_{\varepsilon}^k| \leq |I| \times |Q|^2 \Delta | \times d \), where \( d \) is a constant. Towards this goal, we will bound the number of \( \varepsilon \)-leaves, non-empty leaves, and product nodes by \( \mathcal{O}(|I| \times |Q|^2 \Delta) \) independently. Union nodes can be bounded by counting the other types of nodes: The only cases where a union node is created are (1) in line 35, only after a product node had been created, (2) during the creation of a product node (as described in Theorem 9), (3) in line 46, but only whenever one of the previous lines had created either a product node or a non-empty leaf node, and (4) in line 15, which only happens once at the end of the update phase, and iterates by nodes in \( S \), so the number of union nodes created at this for loop at most \( |\mathcal{D}_{\varepsilon}^k| \). The number of \( \varepsilon \)-nodes is at most one, owing to Theorem 6 since its proof shows that at the end of step \( k \), each of the nodes in \( \mathcal{D}_{\varepsilon}^k \) is \( \varepsilon \)-safe. The number of non-empty leaves can be straightforwardly shown to be \( \mathcal{O}(|I| \times |Q|^2 \Delta) \) since each of these leaves was introduced in some step in \( I \), and in each one of these steps, the number
of operations that the algorithm does is in $O(|Q|^2\Delta)$.

To show a bound over the number of product nodes, consider a slight modification of Algorithm 1: product nodes that are created in line 44 are labeled with the step $k$ in which the algorithm is at the moment. Now, for a set of nodes $A$ let $D_A$ be the $\varepsilon$-ECS that is obtained by removing all of the nodes that are not reachable from some node in $A$ from $D$. Let $I_A$ be the set of positions that appear in some non-empty leaf node in $D_A$, and let $P_A$ be the set of step labels that appear in some product node in $D_A$ excluding the steps in $I_A$. Also, let $V_k$ be the set of nodes in $D_A$. We will show by induction on $k$ that $|P_A| \leq |I_A| - 1$ for any $A \subseteq V_k$ which contains at least one node that is not an $\varepsilon$-node. Consider any set $A \subseteq V_k$. The first observation we make here is that we can partition the nodes in $A$ to a collection $\{A_H\}$ of sets of nodes depending on the hash table $H$ they are reachable from, given that they are in $D_A$. Let $Q_A = P_A \cup I_A$. From Lemma 10 we get that for two different sets $A_{H_1}$ and $A_{H_2}$ in the collection, the sets $Q_{H_1}$ and $Q_{H_2}$ are disjoint. Therefore, in step $k$, if the algorithm enters OPENSTEP, we only need to focus on the set $A_S$, and if the algorithm enters CLOSESTEP (note that in this case, $S^k$ is composed only of $\varepsilon$-nodes). The rest of the hash tables were reachable on a previous step, so the inequality can be reached by adding up the inequalities that held in those steps. First, note that if none of the product nodes in $A$ were created in step $k$, then we can consider the set $B$ of nodes reachable from $A$ that were created in a previous step and notice that $P_A = P_B \cup I_B \subseteq I_A$, so the statement follows since $B \subseteq V_{k-1}$. Also, note that if the algorithm in step $k$ enters OPENSTEP, all of the product nodes created in this step are directly connected to a non-$\varepsilon$ leaf created in this same step, so the statement also follows. From this point on, we can assume that the algorithm enters CLOSESTEP on step $k$, and all of the nodes in $A$ are reachable from some node in $S^k$, and there is at least one product node in $A$ that was created in step $k$. Let $P$ be the set of product nodes in $A$ that were created on step $k$. Consider the span curlevel($k$) = [$j$, $k$). The prod operation in line 44 either creates a new product node, or makes $v$ reference a node that already existed in $S^{k-1}$ or the topmost table in $T^j$. Furthermore, if a product node is created in line 44, then Theorem 5 tells us that it must be connected to a node in $S^{k-1}$ that is not an $\varepsilon$-node, and to a node in the topmost table in $T^j$ that is also not an $\varepsilon$-node. Consider now the set of nodes $B$ that is made up of (1) nodes in $A$ that are reachable from $S^{k-1}$ and (2) nodes in $S^{k-1}$ that are connected to a product node in $P$. Consider also the set of nodes $C$ that is made up of (1) nodes in $A$ that are reachable from the topmost table in $T^j$, and nodes in the topmost table in $T^j$ that are connected to a node in $P$. Note that both sets $B$ and $C$ contain a non-$\varepsilon$ node, and are composed of nodes created in a previous step, so assume that $|P_B| \leq |I_B| - 1$ and that $|P_C| \leq |I_C| - 1$. It can be seen that every node in $D_A$ is either in $B$, $C$, or was created on step $k$, so we get that $P_A = P_B \cup P_C \cup \{k\}$ and $I_A \geq I_B \cup I_C$. From Lemma 10 we get that $Q_B$ and $Q_C$ are disjoint, and putting these facts to together gives us that $|P_A| = |P_B| + |P_C| + 1 \leq |I_B| + |I_C| - 1 \leq |I_A| - 1$.

Having proven this statement, we can deduce that the number of product nodes in $D_A$ is in $O(|I| \times |Q|^2\Delta)$ since the number of steps where they are created is bounded by $|I|$. Therefore, $|D_A| \leq |I| \times |Q|^2\Delta \times d$, for some constant $d$. This concludes the proof.

### A.7 Counterexample that the algorithm is not instance optimal

In this section, we show a VPT for which only logarithmic space in outputweight($T$, $w$) is enough for any stream $S$. Let $o$ be any output symbol and consider a VPT $T$ for which the output set is $[T](w) = \{(o, i) \mid 1 \leq i \leq |w|\}$ if the last symbol in $w$ is $o$ and the empty set otherwise. Clearly, the output weight of any $w$ with respect to $T$ is linear in $|w|$. However,
one could design a streaming evaluation algorithm that has only a counter that stores the length of the input so far, and produces the correct output set after reading the last symbol in \( w \). The enumeration phase can easily be done with output-linear delay (i.e., by counting from 1 to \(|w|\)). This completes the example.

## B Proofs from Section 5

### B.1 Proof of Proposition 7

Let \( \mathcal{D} = (\Sigma, V, I, \ell, r, \lambda) \) be a \( k \)-bounded ECS and \( v \in V \). We will show that the set \( L_{\mathcal{D}}(v) \) can be enumerated with output-linear delay. To show that this is possible we use a data structure we call an output tree. This is a dynamic binary tree \( T \) that append itself to an ECS \( \mathcal{D} \). We define it as follows: If \( v \) is a leaf node in \( \mathcal{D} \), then \( v \) is an output tree of \( \mathcal{D} \). If \( T \) is an output tree and \( v \) is a union node, then \( T' = v(T) \) is an output tree of \( \mathcal{D} \). If \( T_1 \) and \( T_2 \) are output trees and \( v \) is a product node, then \( v(T_1, T_2) \) is an output tree of \( \mathcal{D} \). In either case, we say that \( T \) is rooted in \( v \), and we denote it by \( \text{root}(T) = v \). For an output tree \( T \) we define the functions \( \text{child}_T, \ell_{\text{child}}_T \) and \( \text{rchild}_T \) as follows: If \( v(T') \) is a subtree of \( T \), then \( \text{child}_T(v) = T' \). If \( v(T_1, T_2) \) is a subtree of \( T \), then \( \text{lchild}_T(v) = T_1 \) and \( \text{rchild}_T(v) = T_2 \). These functions are not defined in any other case.

> **Definition 17.** Let \( \mathcal{D} = (\Sigma, V, I, \ell, r, \lambda) \) be an ECS. An output tree \( T \) of \( \mathcal{D} \) is full if for each node \( v \) in \( T \) the following hold: If \( v \) is an union node in \( \mathcal{D} \), then \( \text{child}_T(v) \) is either rooted in \( \ell(v) \) or in \( r(v) \). If \( v \) is a product node in \( \mathcal{D} \), then \( \text{child}_T(v) \) is rooted in \( \ell(v) \) and \( \text{rchild}_T(v) \) is rooted in \( r(v) \).

We define the function \( \text{print}(T) \) as follows: If \( v \) is a leaf node \( v \), then \( \text{print}(T) = \lambda(v) \). If \( T = v(T') \) then \( \text{print}(T) = \text{print}(T') \). If \( T = v(T_1, T_2) \) then \( \text{print}(T) = \text{print}(T_1) \cdot \text{print}(T_2) \).

> **Lemma 18.** Let \( \mathcal{D} \) be an ECS and let \( v \) be a node in \( \mathcal{D} \). For a full output tree \( T \) of \( \mathcal{D} \) rooted on \( v \) it holds that \( \text{print}(T) \in L_{\mathcal{D}}(v) \).

**Proof.** We prove this by induction on the size of \( T \). The case \( T = v \) where \( v \) is a leaf node is trivial. If \( T = v(T') \), \( v \) is an union node, so the proof follows since \( \text{print}(T) \) is equal to \( \text{print}(T') \) which is either in \( L(\ell(v)) \) or \( L(r(v)) \), and therefore in \( L(v) \). If \( T = v(T_1, T_2) \) then \( v \) is a product node. We have that \( \text{print}(T_1) \in L(\ell(v)) \) and \( \text{print}(T_2) \in L(r(v)) \), from which it follows that \( \text{print}(T) \in L(v) \).

> **Lemma 19.** Let \( \mathcal{D} \) be an unambiguous ECS and let \( v \) be a node in \( \mathcal{D} \). For each \( \mu \in L_{\mathcal{D}}(v) \) there exists exactly one full output tree \( T_\mu \) of \( \mathcal{D} \) rooted in \( v \) such that \( \text{print}(T) = \mu \).

**Proof.** Let \( \text{reach}_{\mathcal{D}}(v) \) be the number of nodes reachable from \( v \) in \( \mathcal{D} \), including itself. We will prove this lemma by induction in \( \text{reach}_{\mathcal{D}}(v) \). If \( \text{reach}_{\mathcal{D}}(v) = 1 \), then \( v \) is a leaf node and the proof follows directly since the only output tree rooted in \( v \) is itself. Assume that it holds for every node \( v \) such that \( \text{reach}_{\mathcal{D}}(v) < s \). Let \( v \) be a node such that \( \text{reach}_{\mathcal{D}}(v) = s \) and let \( \mu \in L(v) \). If \( v \) is a union node suppose without loss of generality that \( \mu \notin L(\ell(v)) \). Note that since \( \mathcal{D} \) is unambiguous we have that \( \mu \notin L(r(v)) \). If \( T_\mu = v(T') \) and \( T' \) was rooted in \( r(v) \), Lemma 18 would imply that \( \text{print}(T_\mu) = \text{print}(T') \in L(r(v)) \) which leads to a contradiction. Therefore, \( T_\mu \) could only be of the form \( v(T') \) where \( T' \) is rooted in \( \ell(v) \). From our hypothesis, there exists only one full output tree \( T_\mu' \) such that \( \text{print}(T_\mu') = \mu \), so the proof follows from taking \( T_\mu = v(T_\mu') \). If \( v \) is a product node note that any full output tree \( T \) rooted in \( v \) is of the form \( v(T_1, T_2) \), where \( T_1 \) and \( T_2 \) are rooted in \( \ell(v) \) and \( r(v) \) respectively.
Since \( D \) is unambiguous, there exists only two strings \( \mu_1 \) and \( \mu_2 \) such that \( \mu = \mu_1 \cdot \mu_2 \) and \( \mu_1 \in \mathcal{L}(\ell(v)) \) and \( \mu_2 \in \mathcal{L}(r(v)) \). Let \( T_{\mu_1} \) and \( T_{\mu_2} \) be the only full output trees that are rooted in \( \ell(v) \) and \( r(v) \) respectively for which the hypothesis hold. The proof follows by taking \( T_\mu = v(T_{\mu_1}, T_{\mu_2}) \).

For an ECS \( D \) and node \( v \) we define a total order over the full output trees rooted in \( v \) recursively: If \( v \) is a leaf node there exists only one tree rooted in \( v \) so the order is trivial. If \( v \) is a union node then let \( T_1 = v(T'_1) \) and \( T_2 = v(T'_2) \) be full output trees. We have that \( T_1 < T_2 \) if and only if \( \text{root}(T'_1) = \ell(v) \) and \( \text{root}(T'_2) = r(v) \), or \( T'_1 < T'_2 \). If \( v \) is a product node then let \( T = v(T_1, T_2) \) and \( T' = v(T'_1, T'_2) \). We have that \( T < T' \) if and only if \( T_1 < T'_1 \), or \( T_1 = T'_1 \) and \( T_2 < T'_2 \).

For an ECS \( D \) and an output tree \( T \) on \( D \) we define the operation \( \text{tilt}(T) \) as follows: If \( T = v \), then \( \text{tilt}(T) = v \). If \( T = v(T') \) where \( \text{root}(T') = \ell(v) \), then \( \text{tilt}(T) = v(\text{tilt}(T)) \). If \( T = v(T') \) where \( \text{root}(T') = r(v) \), then \( \text{tilt}(T) = \text{tilt}(T') \). If \( T = v(T_1, T_2) \), then \( \text{tilt}(T) = v(\text{tilt}(T_1), \text{tilt}(T_2)) \). Intuitively, what this operation does is to bypass any union node in \( T \) whose child is a right child in \( D \).

\[\textbf{Definition 20.} \quad \text{For an ECS } D, \text{ an output tree } T \text{ of } D \text{ is left-tilted if it can be obtained as } T = \text{tilt}(T') \text{ where } T' \text{ is a full output tree.}\]

Two left-tilted output trees can be seen in Figure 7. The first tree in the figure is also full. Note that since the root could be a union node whose child is a right child, the root of \( \text{tilt}(T) \) could be a different node than the root of \( T \). We also notice the following result.

\[\textbf{Lemma 21.} \quad \text{Let } D \text{ ECS with a node } v. \text{ The first tree } T \text{ in the ordered sequence of full output trees rooted in } v \text{ is also left-tilted. In other words, } \text{tilt}(T) = T.\]

\[\text{Proof.} \quad \text{We define the operation } \text{build}(v) \text{ as follows. If } v \text{ is a leaf node, then } \text{build}(v) = v. \text{ If } v \text{ is a union node then } \text{build}(v) = v(\text{build}(\ell(v)), \text{build}(r(v))). \text{ Let } T' \text{ be a different full output tree rooted in } v. \text{ A straightforward induction shows that } T < T'.\]

\[\textbf{Lemma 22.} \quad \text{Let } D \text{ be an ECS with an output tree } T. \text{ We have that } \text{print}(\text{tilt}(T)) = \text{print}(T).\]

\[\text{Proof.} \quad \text{The proof follows by a straightforward induction on the tree.}\]
Streaming query evaluation with constant delay enumeration over nested documents

**Algorithm 2** Enumeration of the set $L_D(v)$ for a CE $D$ and a node $v$.

1: procedure Enumerate$(D, v)$
2: $T \leftarrow \text{BuildTree}(D, v)$
3: Output #
4: while $T \neq \emptyset$ do
5: Output print$(T)$
6: Output #
7: $T \leftarrow \text{NextTree}(D, T)$
8: procedure BuildTree$(D, v)$
9: if $\lambda(v) \in \Sigma$ then
10: return $v$
11: else if $\lambda(v) = \odot$ then
12: $T_1 \leftarrow \text{BuildTree}(D, \ell(v))$
13: $T_2 \leftarrow \text{BuildTree}(D, r(v))$
14: Return $v(T_1, T_2)$
15: else if $\lambda(v) = \uplus$ then
16: $T \leftarrow \text{BuildTree}(D, \ell(v))$
17: Return $v(T)$
18: procedure NextTree$(D, T)$
19: if $T = v$ then
20: return $\emptyset$
21: else if $T = v(T_1, T_2)$ then
22: $T_2 \leftarrow \text{NextTree}(D, T_2)$
23: if $T_2$ is empty then
24: return $\emptyset$
25: if $T_1$ is empty then
26: $T_2 \leftarrow \text{BuildTree}(D, r(v))$
27: return $T$
28: else if $T = v(T')$ then
29: $T' \leftarrow \text{NextTree}(D, T')$
30: if $T' = \emptyset$ then
31: $T \leftarrow \text{BuildTree}(D, r(v))$
32: return $T$
33: end

$\text{tilt}(T_1), \ldots, \text{tilt}(T_m)$ for which $T_1 < \ldots < T_m$ is the complete sequence of full output trees rooted in $v$. After generating each tree $T$, the procedure outputs the string print$(T)$ which can be easily done with a depth-first traversal on the tree. The procedure is detailed in Algorithm 2.

The procedure BuildTree builds a completely embedded output tree rooted in $u$. The procedure NextTree receives a tree rooted in $u$ and recursively builds the next tree in the sequence $\text{tilt}(T_1), \ldots, \text{tilt}(T_m)$ for which $T_1 < \ldots < T_m$ is the sequence of full output trees rooted in $u$.

We can deduce the following from Lemma 21.

**Corollary 23.** Let $D$ be an ECS and let $v$ be one of its nodes. BuildTree$(D, v)$ builds a full output tree $T$ that is the first in the ordered sequence of full output trees rooted in $v$.

We prove the correctness of the algorithm in the following results.

**Lemma 24.** Let $D$ be an ECS and let $v$ be one of its nodes. Let $T_1 < \ldots < T_m$ be the sequence of full output trees rooted in $v$. If the procedure NextTree receives $(D, \text{tilt}(T_i))$ it returns $\text{tilt}(T_{i+1})$, or $\emptyset$ if $i = m$.

**Proof.** We prove this by induction in $\text{reach}_D(v)$. If $v$ is a leaf node, the sequence consists only of the tree $v$, so the proof follows directly. Assume it holds for nodes $v'$ such that $\text{reach}_D(v') < s$ and let $v$ be such that $\text{reach}_D(v) = s$. If $v$ is a union node notice that there exists an $e$ such that the sequence of full output trees rooted in $v$ is $T_1 < \ldots < T_e < T_{e+1} < \ldots < T_m$ where $T_e = v(T'_e)$ and $T_{e+1} = v(T'_{e+1})$, and $\text{root}(T'_e) = \ell(v)$ and $\text{root}(T'_{e+1}) = r(v)$. If $i < e$ or $i > e$, then the proof follows by induction. Otherwise, if $i = e$, note that the procedure BuildTree$(D, r(v))$ builds the first full output tree rooted in $r(v)$, which is $T'_{e+1}$, and is equal to $\text{tilt}(T_{e+1})$. If $v$ is a product node the proof follows straightforwardly by induction over the algorithm.

From the previous results, correctness of the algorithm follows.
Claim 25. Enumerate receives an ECS $\mathcal{D}$ and one of its nodes $v$ and outputs all of the elements in $\mathcal{L}_\mathcal{D}(v)$ one by one without repetition.

Proof. Let $T_1 < \cdots < T_m$ be the sequence of full output trees rooted in $v$. The algorithm starts by generating $T_1 = \text{tilt}(T_1)$ as proven by Corollary 21. Then on each step $i$, the algorithm iterates $T$ as $\text{tilt}(T_i)$ to transform it into $\text{tilt}(T_i)$, as proven by Lemma 24. In each step, an element in $\mathcal{L}_\mathcal{D}(v)$ is given as output as proven by Lemma 18. Moreover, the sequence $T_1 < \cdots < T_m$ allows the set $\mathcal{L}_\mathcal{D}(v)$ to be produced exhaustively without repetitions, as proven by Lemma 19. ◀

The following results ensure that each tree in the sequence can be generated efficiently.

Lemma 26. Let $\mathcal{D}$ be an ECS, let $v$ one of its nodes, and let $T_1 < \cdots < T_m$ be the sequence of full output trees rooted in $v$. If the procedure NextTree receives $(\mathcal{D}, T)$ it returns the tree $T'$ in at most $c(|T| + |T'|)$ time, for some constant $c$.

Proof. We choose $c$ as a factor of the number of steps that are taken in NextTree without taking into account recursion. That is, the time that it takes to run steps 19-33 without calls. A first observation that we make is that BuildTree builds a tree $T$ in time at most $c|T|$, since each call to BuildTree takes less than $c$ steps, and exactly one call to BuildTree is done per node in $T$. We prove the lemma by induction on the tree. If $v$ is a leaf node, then the proof is trivial. If $v$ is a product node, let $T = v(T_1, T_2)$, and let $T'$ be the output of NextTree such that $T' = v(T'_1, T'_2)$ or $T' = \varnothing$. If the call in line 22 returns an nonempty tree, then the procedure takes time $c + c(|T_2| + |T'_2|)$. Otherwise, line 22 takes time $c|T_2|$. Then, if the call in line 24 returns a nonempty tree, it takes time $c(|T_1| + |T'_1|)$, and then the call in line 27 takes time $c|T'_2|$; otherwise, it takes time $c|T_1|$. In each of the routes where $T'$ is not empty, the execution time is bounded by $c(|T_1| + |T_2| + |T'_1| + |T'_2| + 1) \leq c(|T| + |T'|)$, and if $T' = \varnothing$, it is bounded by $c(|T_1| + |T_2| + 1) = c|T|$ which proves the statement. If $v$ is a union node, let $T = v(T')$ and let $T'_{\text{out}}$ be the output of NextTree. If the call in line 30 returns a nonempty tree, it takes time $c(|T'| + |T'_{\text{out}}|)$, where $T'_{\text{out}} = v(T'_{\text{out}})$, and the procedure takes total time $c + c(|T'| + |T'_{\text{out}}|) \leq c(|T| + |T'_{\text{out}}|)$, which proves the statement. Otherwise, the call in line 30 takes time $c|T'|$, and then line 32 takes time $c|T'_{\text{out}}|$, which adds to a total time $c + c(|T'| + |T'_{\text{out}}|) = c(|T| + |T'_{\text{out}}|)$, which also proves the statement. ◀

Lemma 27. Let $\mathcal{D}$ be a $k$-bounded ECS and $T$ be a left-tilted output tree in $\mathcal{D}$. The size of $T$ is at most $2k|\text{print}(T)|$.

Proof. Note that $|\text{print}(T)|$ is equal to the number of leaves in $T$. Since $T$ is left-tilted, then for each union node $v$ in $T$ we have that child$_T(v)$ is rooted in $\ell(v)$. We also have that $\mathcal{D}$ is $k$-bounded, so there are at most $k$ nodes between each pair of product nodes in $T$. We know that a binary tree with $e$ leaves has $2e - 1$ nodes and $2e - 2$ edges. Therefore, if we replace each edge by $k - 1$ nodes we obtain a tree whose size is an upper bound for the size of $T$, and the proof follows. ◀

From these lemmas we obtain a result that ensures nearly output-linear delay.

Claim 28. Let $\mathcal{D}$ be a $k$-bounded ECS and let $v$ be a node in $\mathcal{D}$. For some sequence $\mu_1, \ldots, \mu_m$ that contains exactly the elements in $\mathcal{L}_\mathcal{D}(v)$ without repetition, Enumerate can produce each element $\mu_i$ for $i \in [2, m]$ with delay $c(|\mu_{i-1}| + |\mu_i|)$, and $\mu_1$ with delay $c|\mu_1|$, where $c$ is a constant.
Proof. The sequence in question is the one given by the total order $T_1 < \cdots < T_m$ of total output trees rooted in $v$, for which $\mu_i = \text{print}(T_i)$. Let $c'$ be the constant in Lemma 26 and let $d$ be a constant such that $\text{print}(T)$ can be produced in time $d|T|$. We have that BuildTree can build a tree $T$ in size. In Lemma 26 it is shown that the first tree $T_1$ in the sequence can be generated in time $c'|T_1|$, and in Lemma 27 we show that $|T_1| \leq 2k|\mu_1|$. From this, it follows that $\mu_1$ can be produced in time $2k(c' + d)|\mu_1|$. For each $i \in [2, m]$ Lemma 26 shows that $T_i$ can be generated in time $c'(|T_{i-1}| + |T_i|)$. We can bound this number by $2kc'(|\mu_{i-1}| + |\mu_i|)$ using Lemma 27. Printing the output takes time $d|T_1|$, so the total time is $2kc'(|\mu_{i-1}| + |\mu_i|) + 2kd|\mu_i|$, which is bounded by $2k(c' + d)(|\mu_{i-1}| + |\mu_i|)$. We conclude the proof by taking $c = 2k(c' + d)$.

We optimize this result to obtain the desired statement.

**Proposition 29** (Proposition 7). Fix $k \in \mathbb{N}$. Let $\mathcal{D}$ be an unambiguous and $k$-bounded ECS. Then the set $\mathcal{L}_\mathcal{D}(v)$ can be enumerated with output-linear delay for any node $v$ in $\mathcal{D}$.

**Proof.** Let $c'$ be the constant from Claim 28. We have that the elements in $\mathcal{L}_\mathcal{D}(v)$ can be enumerated with delay proportional to the size of each output, plus the size of the previous output. Moreover, the first output can be produced in constant time. Let $\mu_1, \ldots, \mu_m$ be the elements in $\mathcal{L}_\mathcal{D}(v)$ in the order that the algorithm from Claim 28 produces them. We utilize this algorithm as an oracle to retrieve each output $\mu_i$ in order. After each output is obtained, it is printed in an auxiliary tape instead of the output tape. Then the oracle is called to obtain the following output $\mu_{i+1}$. Once $c'\mu_i$ steps have been taken in the oracle, $\mu_i$ is printed to the output tape. After this, the call to obtain $\mu_{i+1}$ is resumed. This output is then obtained $c(|\mu_i| + |\mu_{i+1}|)$ steps after the oracle was called, but only $c|\mu_{i+1}|$ steps after $\mu_i$ was printed. From this, we have that for each $i \in [1, m]$, printing $\mu_i$ takes time $(2c' + d)|\mu_i|$, where $d$ arises from the modifications to the algorithm. By taking the constant $c = (2c' + d)$ we obtain output-linear delay.

### B.2 Proof of Theorem 8

The construction of the operators and the reasoning why each partial result $(\mathcal{D}', v')$ is 2-bounded is stated in the paper. By adding the condition that $\mathcal{D}'$ is unambiguous we can deduce that $\mathcal{L}_{\mathcal{D}'}(v')$ can be enumerated with output-linear delay using Proposition 7.

### B.3 Proof of Theorem 9

In $\varepsilon$-ECS, $\varepsilon$-nodes are treated quite particularly. For a given $\varepsilon$-ECS $\mathcal{D}$, we require that any node $v \in \mathcal{D}$ satisfies exactly one of the following: (1) $\lambda(v) = \varepsilon$ and for any node $u$ which is reachable from $v$ it holds that $\lambda(u) \neq \varepsilon$, (2) $\lambda(v) = \varepsilon$ or (3) $\lambda(v) = \cup, \lambda(\ell(v)) = \varepsilon$, and $r(v)$ satisfies (1). In other words, $\varepsilon$ can only be child of a union node with in-degree 0. For the rest of the proof, we will refer to a node $v$ that satisfies each case as a node such that (1) $\varepsilon \notin \mathcal{L}_{\mathcal{D}}(v)$, (2) $\lambda(v) = \varepsilon$ or (3) $v$ is in Case 3, respectively. Note that this construction ensures that if $\varepsilon \in \mathcal{L}_{\mathcal{D}}(v)$, it can be retrieved in constant time.

With these conditions in mind, we can address output-depth, $k$-bounded and safeness. The definition of output-depth is unchanged for nodes $v$ for which $\varepsilon \notin \mathcal{L}_{\mathcal{D}}(v)$, if $\lambda(v) = \varepsilon$, then $\text{odepth}(v) = 0$, and if $v$ is in Case 3, $\text{odepth}(v) = 1$. The definition of $k$-bounded is unchanged. The definition of safe nodes is unchanged except for the additional restriction that a node $v$ can only be safe if $\varepsilon \notin \mathcal{L}_{\mathcal{D}}(v)$. 
\[ \begin{array}{cc}
\text{(a)} & \text{(b)} \\
\text{union} & \text{union} \\
\end{array} \]

**Figure 8** Gadgets for \( \text{prod} \) as defined for an \( \varepsilon \)-ECS. Nodes \( v'_a, v'_b \) and \( v'_c \) correspond to \( v' \) as is defined for cases (a), (b) and (c) respectively.

\[ \begin{array}{cc}
\text{(c)} & \text{(d)} \\
\text{union} & \text{union} \\
\end{array} \]

**Claim 30.** For a \( k \)-bounded unambiguous \( \varepsilon \)-ECS \( D \), the set \( L_D(v) \) can be enumerated with output-linear delay for every node \( v \) in \( D \).

**Proof.** To prove this, we formalize the idea behind the construction of an \( \varepsilon \)-ECS. Let \( D_v \) be the \( \varepsilon \)-ECS induced by the nodes that are reachable from \( v \). Formally, let \( V_v \) be this set of nodes. Then \( D_v = (\Sigma, V_v, \ell, r, \lambda) \) where \( I_v = I \cap V_v \), and also \( \ell, r, \lambda \) are the functions \( \ell, r \) and \( \lambda \) induced by \( V_v \). It is straightforward to check that \( L_D(v) = L_D(v) \). Note that if \( v \notin L_D(v) \), then \( D_v \) is a regular ECS, and if \( v \) is in Case 3, then \( D_{r(v)} \) is a regular ECS as well. Furthermore, if \( D \) is unambiguous and \( k \)-bounded, then the ECS in each of these cases is also unambiguous and \( k \)-bounded. From here, the proof follows straightforwardly by using Proposition 7 over these ECS.

One last notion we make use of is \( \varepsilon \)-safe nodes. For a given \( \varepsilon \)-ECS \( D \) and \( v \in D \) we say that \( v \) is \( \varepsilon \)-safe if either (1) \( v \) is safe, (2) \( \lambda(v) = \varepsilon \), or (3) \( v \) is in Case 3 and \( r(v) \) is safe.

We define the operations \( \text{add, prod} \) and \( \text{union over } D \) to return a pair \( (D', v') \) such that \( D' = (\Sigma, V', \ell', r', \lambda') \) as follows:

- For \( \text{add}(D, a) \rightarrow (D', v') \) we define \( V' = V \cup \{v' \} \), \( I' = I \), and \( \lambda'(v') = a \).

Assume \( v_1 \) and \( v_2 \) are \( \varepsilon \)-safe. Further, assume that for every word in \( w \in L_D(v_1) \cdot L_D(v_2) \) there exist only two non-empty words \( w_1 \) and \( w_2 \) such that \( w_1 \in L_D(v_1), w_2 \in L_D(v_2) \) and \( w = w_1 \cdot w_2 \). Since both \( v_1 \) and \( v_2 \) \( \varepsilon \)-safe, we define \( \text{prod}(D, v_1, v_2) \rightarrow (D', v') \) by separating into nine cases, of which the first six are straightforward:

- If \( \varepsilon \notin L_D(v_1) \) and \( \varepsilon \notin L_D(v_2) \), we use the construction given for a regular ECS.
- If \( \varepsilon \notin L_D(v_1) \) and \( \lambda(v_2) = \varepsilon \), we define \( v' = v_1 \), and \( D' = D \).
- If \( \lambda(v_1) = \varepsilon \) and \( \varepsilon \notin L_D(v_2) \), we define \( v' = v_2 \), and \( D' = D \).
- If \( \lambda(v_1) = \varepsilon \) and \( \lambda(v_2) = \varepsilon \), we define \( v' = v_1 \), and \( D' = D \).
- If \( v_1 \) is in Case 3 and \( \lambda(v_2) = \varepsilon \), we define \( v' = v_1 \), and \( D' = D \).

The other three cases are more involved and they are presented graphically in Figure 8. Formally, they are defined as follows:

\( \begin{array}{ll}
\text{(a)} & \text{If } \varepsilon \notin L_D(v_1) \text{ and } v_2 \text{ is in Case 3, then } V' = V \cup \{v', v''\}, I' = I \cup \{v', v''\}, \ell'(v') = v_1, r'(v') = v'' = \varepsilon, \ell(v'') = v_1, r(v'') = v_2, \lambda(v') = \varepsilon \text{ and } \lambda(v'') = \varepsilon. \\
\text{(b)} & \text{If } v_1 \text{ is in Case 3 and } \varepsilon \notin L_D(v_1), \text{ then } V' = V \cup \{v', v''\}, I' = I \cup \{v', v''\}, \ell'(v') = v_1, r'(v') = v_2 = v_3, \lambda'(v') = \varepsilon, \lambda(v'') = \varepsilon = \varepsilon. \\
\text{(c)} & \text{If both } v_1 \text{ and } v_2 \text{ are in Case 3, we do a slightly more delicate construction. First, we define } a \text{ ECS } V'' \text{ with } V'' = V \cup \{v', v''\}, I'' = I \cup \{v'', v''\}, \ell''(v') = v_1, r''(v') = v_2, \lambda''(v') = \varepsilon. \text{ Lastly, let } V' = V' \cup \{v', v''\}, I' = I' \cup \{v', v''\}, \ell'(v') = v_1, r'(v') = v_2, \lambda(v') = \varepsilon. \end{array} \)
Streaming query evaluation with constant delay enumeration over nested documents

Note that the union operation in case (c) does not recurse since \( r(v_1) \) is safe. In particular, it does not reach any \( \varepsilon \)-leaf.

Assume \( v_1 \) and \( v_2 \) are \( \varepsilon \)-safe nodes. Further, assume that \( \mathcal{L}_D(v_1) \setminus \{ \varepsilon \} \) and \( \mathcal{L}_D(v_2) \setminus \{ \varepsilon \} \) are disjoint. We define union \( (D, v_1, v_2) \rightarrow (D', v') \) as follows:

- If \( \varepsilon \not\in \mathcal{L}_D(v_1) \) and \( \varepsilon \not\in \mathcal{L}_D(v_2) \), we use the construction given for a regular ECS.
- If \( \varepsilon \not\in \mathcal{L}_D(v_1) \) and \( \lambda(v_2) = \varepsilon \), we define \( V' = V \cup \{ v' \} \), \( I' = I \cup \{ v' \} \) and \( \lambda(v') = \emptyset \). We connect \( t'(v') = v_2 \) and \( r'(v') = v_1 \).
- If \( \varepsilon \not\in \mathcal{L}_D(v_1) \) and \( v_2 \) is in Case 3, let \( (D'', v'') = \text{union}(D, v_1, r(v_2)) \) as defined for a regular ECS. We define \( V' = V'' \cup \{ v' \} \), \( I' = I'' \cup \{ v' \} \) and \( \lambda(v') = \emptyset \) where \( \lambda' \) is an extension of \( \lambda'' \). We connect \( t'(v') = \delta(v_2) \) and \( r'(v') = v'' \).
- If \( \lambda(v_1) = \varepsilon \) and \( \varepsilon \not\in \mathcal{L}_D(v_2) \), we define \( V' = V \cup \{ v' \} \), \( I' = I \cup \{ v' \} \) and \( \lambda(v') = \emptyset \). We connect \( t'(v') = v_1 \) and \( r'(v') = v_2 \).
- If \( \lambda(v_1) = \varepsilon \) and \( \lambda(v_2) = \varepsilon \), we define \( D' = D \) and \( v' = v_1 \).
- If \( \lambda(v_1) = \varepsilon \) and \( v_2 \) is in Case 3, we define \( D' = D \) and \( v' = v_2 \).
- If \( v_1 \) is in Case 3 and \( \varepsilon \not\in \mathcal{L}_D(v_2) \), let \( (D'', v'') = \text{union}(D, r(v_1), v_2) \) as defined for a regular ECS. We define \( V' = V'' \cup \{ v' \} \), \( I' = I'' \cup \{ v' \} \) and \( \lambda(v') = \emptyset \) where \( \lambda' \) is an extension of \( \lambda'' \). We connect \( t'(v') = \delta(v_2) \) and \( r'(v') = v'' \). (*)
- If \( v_1 \) is in Case 3 and \( \lambda(v_2) = \varepsilon \), we define \( D' = D \) and \( v' = v_1 \).
- If both \( v_1 \) and \( v_2 \) are in Case 3, let \( (D', v') = \text{union}(D, r(v_1), v_2) \) by using the construction of case (*).

Whenever \( D'' \) is mentioned it is assumed to be equal to \((\Sigma, V'', I'', \ell'', r'', \lambda'')\).

It is straightforward to check that each operation behaves as expected. That is, if \( \text{add}(D, a) \rightarrow (D', v') \), then \( \mathcal{L}_D(v') = \{ a \} \), if \( \text{prod}(D, v_1, v_2) \rightarrow (D', v') \), then \( \mathcal{L}_D(v') = \mathcal{L}_D(v_1) \cup \mathcal{L}_D(v_2) \), and if union \( (D, v_1, v_2) \rightarrow (D', v') \), then \( \mathcal{L}_D(v_1) \cup \mathcal{L}_D(v_2) \). Moreover, if both \( v_1 \) and \( v_2 \) are \( \varepsilon \)-safe, then the resulting node \( v' \) is \( \varepsilon \)-safe as well for each operation.

Note that each operation falls into a finite number of cases which can be checked exhaustively, and each construction has a fixed size, so they take constant time. Furthermore, each operation is fully persistent.

Finally, let \((D', v')\) be a partial result obtained from applying the operations \text{add}, \text{prod} and union such that \( D' \) is unambiguous. The proof follows from Claim 30.

## C  Proofs from Section 6

### C.1 Proof of Lemma 10

We will prove the lemma by induction on \( k \). The case \( k = 0 \) is trivial since \( \text{currlevel}(0) = [0,0] \), \( S^0_{\rho,q} \) is empty and \( \text{lowerlevel}(0) \) is not defined. We assume that statements 1 and 2 of the lemma are true for \( k - 1 \) and below.

If \( a_k \in \Sigma^* \), the algorithm proceeds into \text{OpenStep} to build \( S^k \) and \( T^k \). Statement 1 can be proved trivially since \( \text{currlevel}(k) = [k,k] \), similarly as for the base case. For statement 2 let \( \text{lowerlevel}(k) = [i,k-1] \), and consider a run \( \rho \in \text{Rums}(T, w[1,k]) \) such that \( \rho[i,k] \) starts on \( p \) and ends on \( q \) for some \( p,q \) and \( \gamma \), and let \( p' \) be its second-to-last state. Since \( a_k \) is an open symbol, then the string \( a_{i+1}\cdots a_{k-1} \) is well-nested, so it holds that \( \text{currlevel}(k-1) = [i,k-1] \). Therefore, from our hypothesis it holds that \( \mathcal{L}_D(S^{k-1}_{\rho,p}) \) contains \( \rho[i,k-1] \), and so, \( \text{out}(\rho[i,k]) \) is included in \( \mathcal{L}_D(T^k_{\rho,\gamma,q}) \) at some iteration of \( T^k_{\rho,\gamma,q} \) at line 36. To show that every element in \( \mathcal{L}_D(T^k_{\rho,\gamma,q}) \) corresponds to some run \( \rho \in \text{Rums}(T, w[1,k]) \), we note that the only step that modifies \( T^k_{\rho,\gamma,q} \) is line 36, which is reached only when a valid subrun from \( i \) to \( k \) can be constructed.
If \( a_k \in \Sigma^* \), the algorithm proceeds into CloseStep to build \( S^k \) and \( T^k \). Let \( \text{currlevel}(k) = [j, k] \). In this case, statement 2 can be deduced directly from the hypothesis since \( j < k \) and the table on the top of \( T^k \) is the same as \( T^j \). To prove statement 1 notice that since \( a_k \) is a close symbol it holds that \( \text{currlevel}(k - 1) = [j', k - 1] \) and \( \text{lowerlevel}(k - 1) = [j, j' - 1] \) for some \( j' \). Consider a run \( \rho \in \text{Runs}(T, w) \) such that \( \rho(j, k) \) starts on \( p \), ends on \( q \), and the last symbol pushed onto the stack is \( \gamma \). This run can be subdivided in three subruns from \( p \) to \( p' \), from \( p' \) to \( q' \), and a transition from \( q' \) to \( q \) as it is illustrated in Figure 2 (Right). The first two subruns correspond to \( \rho(j, j' + 1) \) and \( \rho(j', k - 1) \), for which \( \text{out}(\rho[j, j'+1]) \in \mathcal{L}_D(T_p^{\gamma-1}_{\rho'}) \) and \( \text{out}(\rho[j', k-1]) \in \mathcal{L}_D(S^k_{\rho')}. \) Therefore, \( \text{out}(\rho[j, k]) \in \mathcal{L}_D(S^k_{\rho')} \) at some iteration of line 47. To show that every element in \( S^k_{\rho'} \) corresponds to some run \( \rho \in \text{Runs}(T, w(1, k)) \), note that the only line at which \( S^k_{\rho'} \) is modified are is line 47, which is reached only when a valid run from \( j \) to \( k \) has been constructed.

### C.2 Proof of Theorem 11

This theorem is a straightforward consequence of Lemma 10.

### C.3 Proof of Lemma 12

**Proof.** For the sake of simplification, assume that \( T \) is I/O-unambiguous on subruns as well. Formally, we extend the condition so that for every well-nested word \( w \), span \( [i, j] \) and \( \mu \in \Omega^* \) there exists only one run \( \rho \in \text{Runs}(T, w) \) such that \( \mu = \text{out}(\rho[i, j]) \). Towards a contradiction, we assume that \( D \) is not I/O-unambiguous. Therefore, at least one of these conditions must hold: (1) There is some union node \( v \) in \( D \) for which \( \mathcal{L}_D(\ell(v)) \) and \( \mathcal{L}_D(r(v)) \) are not disjoint, or (2) there is some product node \( v \) for which there are at least two ways to decompose some \( \mu \in \mathcal{L}_D(v) \) in non-empty strings \( \mu_1 \) and \( \mu_2 \) such that \( \mu = \mu_1 \cdot \mu_2 \) and \( \mu_1 \in \mathcal{L}(\ell(v)) \) and \( \mu_2 \in \mathcal{L}(r(v)) \).

Assume the first condition is true and let \( v \) be an union node that satisfies it, and let \( k \) be the step in which it was added to \( D \). If this node was added on OpenStep, then the node \( v \) represents a subset of the subruns defined in condition 1 of Lemma 10. Consider two different iterations of lines 35-36 on step \( k \) where two nodes \( v \) and \( v' \) were united for which there is an element \( \mu \in \mathcal{L}_D(v) \cap \mathcal{L}_D(v') \). Since these nodes were assigned to \( T_{p,\gamma,q} \) on different iterations, the states \( p' \) that were being considered must have been different. Therefore, if \( \text{lowerlevel}(k) = [i, j], \mu = \text{out}(\rho[i, k]) = \text{out}(\rho'[i, k]) \) for two runs \( \rho \) and \( \rho' \) where the \( (k - 1) \)-th state is different. This violates the condition that \( T \) is unambiguous. If this node was added on CloseStep, we can follow an analogous argument. Note that union nodes created on a prod operation are unambiguous by construction (see Theorem 9).

Assume now that the second condition is true and let \( v \) be a node for which the condition holds and let \( k \) be the step where it was created. We note that this node could not have been created in OpenStep since the only step that creates product nodes is line 36, where \( v_3 \) has the label \( (v, k) \), and \( S_{p,v'} \) is connected to nodes that were created in a previous step, so all of the elements \( \mu \in \mathcal{L}(S_{p,v'}) \) only contain pairs \( (v, j) \) where \( j < k \). We can follow a similar argument to prove that this node could not have been created in line 45 of CloseStep. We now have that \( v \) was created in line 44 of OpenStep, and therefore \( \ell(v) = T_{p,\gamma,q}^{k-1} \) and \( r(v) = S_{p,v'}^{k-1} \) unless either of these indices were empty. However, that is not possible since we assumed that the step where \( v \) was created was \( k \), and if either were empty, no node would have been created. Now let \( \mu \in \mathcal{L}(v) \) be such that there exist strings \( \mu_1, \mu_2 \in \mathcal{L}(T_{p,\gamma,q}) \) and \( \mu_2, \mu_2' \in \mathcal{L}(S_{p,v')^{k-1}) \) such that \( \mu = \mu_1 \cdot \mu_2 \) and \( \mu_2 \neq \mu_2' \). Without loss of generality, let \( \mu'' \) be the non-empty suffix in \( \mu_1 \) such that \( \mu_1 \cdot \mu'' = \mu_1 \). Here we reach a contradiction since \( \mu'' \)
is a prefix of $\mu_2$ and thus it must contain a pair $(\alpha, j)$ such that and $j \in \text{lowerlevel}(k)$ and $j \in \text{curlevel}(k)$, which is not possible.

The fact that all nodes in $D$ are $\varepsilon$-safe carries easily from Theorem 9.

D Applications in document spanners

This section presents an application of our enumeration algorithm to the evaluation of recursive spanners \cite{25}. Practical formalisms to define document spanner for information extraction with recursion was only proposed very recently. In \cite{18}, the author suggests using extraction grammars to specify document spanners, which is the natural extension of regular spanners to a controlled form of recursion. Furthermore, the author gives an enumeration algorithm for unambiguous functional extraction grammars that outputs the results with constant-delay after quintic time preprocessing (i.e., in the document). We can show a streaming enumeration algorithm with update-time that is independent of the document, and output-linear delay by restricting to the class of visibly pushdown extraction grammars. We proceed by recalling the framework of document spanners and extraction grammars to define the class of visibly pushdown extraction grammars and state the main algorithmic result.

We start by recalling the basics of document spanners \cite{25}. Fix an alphabet $\Sigma$ and a set of variables $\mathcal{V}$ such that $\Sigma \cap \mathcal{V} = \emptyset$. A document $d$ over $\Sigma$ is basically a word in $\Sigma^*$. A span $s$ of a document $d$ is a pair $[i, j)$ of natural numbers $i$ and $j$ with $1 \leq i \leq j \leq |d| + 1$. Intuitively, a span represents a substring of $d$ by identifying the starting and ending position. We denote by $\text{Spans}(d)$ the set of all possible spans of $d$. Let $X \subseteq \mathcal{V}$ be a finite set of variables. An $(X, d)$-mapping $\mu : X \rightarrow \text{Spans}(d)$ assigns variables in $X$ to spans of $d$. An $(X, d)$-relation is a finite set of $(X, d)$-mappings. Then a document spanner $P$ (or just spanner) is a function associated with a finite set $X$ of variables that maps documents $d$ into $(X, d)$-relations.

We use the framework of extraction grammars, recently proposed in \cite{18}, to specify document spanners. For $X \subseteq \mathcal{V}$, let $C_X = \{(x, x) \mid x \in X\}$ be the set of captures of $X$ where, intuitively, $\{x\}$ denotes the opening of $x$, and $\{x\}$ its closing. An extraction context-free grammar, or extraction grammar for short, is a tuple $G = (X, V, \Sigma, S, P)$ such that $X \subseteq \mathcal{V}$, $\Sigma$ is a finite set of non-terminal symbols with $\Sigma \cap \mathcal{V} = \emptyset$, $V \subseteq \Sigma$ is the alphabet of terminal symbols, and $P \subseteq V \times (V \cup \Sigma \cup C_X)^*$ is a finite relation. In the literature, the elements of $V$ are also referred as “variables”, but we call them non-terminals to distinguish $V$ from $\mathcal{V}$. Each pair $(A, \alpha) \in P$ is called a production and we write it as $A \rightarrow \alpha$. The set of productions $P$ defines the (left) derivation relation $\Rightarrow_G \subseteq (V \cup \Sigma \cup C_X)^* \times (V \cup \Sigma \cup C_X)^*$ such that $wA\beta \Rightarrow_G w\alpha\beta$ iff $w \in (\Sigma \cup \mathcal{C}_X)^*$, $A \in V$, $\alpha, \beta \in (V \cup \Sigma \cup C_X)^*$, and $A \rightarrow \alpha \in P$. We denote by $\Rightarrow^*_G$ the reflexive and transitive closure of $\Rightarrow_G$. Then the language defined by $G$ is $L(G) = \{w \in (\Sigma \cup \mathcal{C}_X)^* \mid S \Rightarrow^*_G w\}$. A word $w \in L(G)$ is called a $\text{ref-word}$ produced by $G$.

In order to define a spanner from $G$, we need to interpret ref-words as mappings. \cite{25}. Formally, a ref-word $r = a_1 \ldots a_n \in (\Sigma \cup \mathcal{C}_X)^*$ is called valid for $X$ if, for every $x \in X$, there exists exactly one position $i$ with $a_i = \{x\}$ and exactly one position $j$ with $a_j = \{x\}$, such that $i < j$. In other words, a valid ref-word defines a correct match of open and close captures. Moreover, each $x \in X$ induces a unique factorization of $r$ of the form $r = r^p_x \cdot \{x\} \cdot r^c_x$. This factorization defines the $(X, d)$-mapping as follows. Let $\text{plain} : (\Sigma \cup \mathcal{C}_X)^* \rightarrow \Sigma^*$ be the morphism that removes the captures from ref-words, namely, $\text{plain}(a) = a$ when $a \in \Sigma$ and $\text{plain}(c) = \varepsilon$ when $c \in \mathcal{C}_X$. We extend plain to operate over strings in the obvious way. Furthermore, let $r$ be a valid ref-word for $X$, $d$ be a document, and assume that $\text{plain}(r) = d$.
Then we define the \((X,d)\)-mapping \(\mu^r\) such that \(\mu^r(x) = [i,j] \) if \(r = r^p_x \cdot \{x \cdot r_x \cdot \} x \cdot r^p_x, i = \text{plain}(r_x) + 1, \) and \(j = i + \text{plain}(r_x)\). Finally, the spanner \([G]\) associated to an extraction grammar \(G\) is defined over any document \(d \in \Sigma^*\) as follows:

\[
[G](d) = \{ \mu^r \mid r \in L(G), r \text{ is valid for } X, \text{ and } \text{plain}(r) = d \}.
\]

There are two classes of extraction grammars that are relevant for our discussion. The first class of grammars are called functional extraction grammars. An extraction grammar \(G\) is \emph{functional} if every \(r \in L(G)\) is valid for \(X\). In \cite{48} it was shown that for any extraction grammar \(G\) there exists an equivalent functional grammar \(G'\) (i.e. \([G] = [G']\)). Non-functional grammars are problematic given that, even for regular spanners, their decision problems easily become intractable \cite{43,30}. For this reason, we restrict to functional extraction grammars without loss of expressive power. The second class of grammars are called unambiguous extraction grammars. An extraction grammar \(G\) is \emph{unambiguous} if for every \(r \in L(G)\) there exists exactly one path from \(S\) to \(r\) in the graph \(((V \cup \Sigma \cup C_X)^*, \Rightarrow G)\). In other words, there exists exactly one left-most derivation.

We consider now a sub-class of extraction grammars for nested words. Let \(\Sigma = (\Sigma^c, \Sigma^s, \Sigma^l)\) be a structured alphabet. Then a \emph{visibly pushdown extraction grammar} (VPEG) is a functional extraction grammar \(G = (X, V, \Sigma, S, P)\) in which \(\Sigma = (\Sigma^c, \Sigma^s, \Sigma^l)\) is a structured alphabet, and all the productions in \(P\) are of one of the following forms: (1) \(A \rightarrow \epsilon\); (2) \(A \rightarrow aB\) such that \(a \in \Sigma^c \cup C_X\) and \(B \in V\); (3) \(A \rightarrow aBbC\) such that \(a \in \Sigma^c, b \in \Sigma^s,\) and \(B, C \in V\). Intuitively, rules \(A \rightarrow aBbC\) allow to produce arbitrary sequences of neutral symbols, where rules \(A \rightarrow aBbC\) forces the word to be well-nested.

Visibly pushdown extraction grammars are a subclass of extraction grammars that works for well-nested documents. In fact, the reader can notice that the visibly pushdown restriction for extraction grammars is the analog counterpart of visibly pushdown grammars\footnote{The definition of visibly pushdown grammars in \cite{33} is slightly more complicated given that they consider nested words that are not necessary well-nested (see the discussion in Section \ref{sec:visibly_pushdown_grammar}).} introduced in \cite{33}. Therefore, one could expect that VPEGs are less expressive than extraction grammars. Interestingly, we can use Theorem \ref{thm:streaming_algorithm} to give an efficient streaming enumeration algorithm for evaluating VPEG.

\begin{itemize}
  \item \textbf{Theorem 31.} Fix a set of variables \(X\). The problem of, given a visibly pushdown extraction grammar \(G = (X, V, \Sigma, S, P)\) and a stream \(S\), enumerating all \((X, S[1, n])\)-mappings of \([G](w)\) can be solved with update-time \(O(2^{|G|^3})\), and output-linear delay. Furthermore, if \(G\) is restricted to be unambiguous, then the problem can be solved with update-time \(O(|G|^3)\).
\end{itemize}

This result goes by constructing an extraction pushdown automata \cite{48} from \(G\), and reduce it to a visibly pushdown transducers. Note that, although the update-time of the algorithm is exponential in the size of the grammar, in terms of data-complexity the update-time is constant. Furthermore, for the special case of unambiguous grammars the update-time is even polynomial. Unambiguous grammars are very common in parsing tasks \cite{2} and, thus, this restriction could be useful in practice.

\subsection{D.1 Proof of Theorem 31}

To link the model of visibly pushdown extraction grammars and visibly pushdown automata we define another class of automata based on the ideas in \cite{48}. Let \(A\) be an \emph{extraction visibly pushdown automaton} (EVPA) if \(A = (X, Q, \Sigma, \Gamma, \Delta, I, F)\) where \(X\) is a set of variables, \(Q\)
is a set of states, \( \Sigma = (\Sigma^*, \Sigma^\prime, \Sigma^\dagger) \) is a visibly pushdown alphabet, \( \Gamma \) is a stack alphabet, \( \Delta \subseteq (Q \times \Sigma^* \times Q \times \Gamma) \cup (Q \times \Sigma^* \times \Gamma \times Q) \cup (Q \times (\Sigma^\dagger \cup \mathcal{C}_X) \times Q) \), \( I \) is a set of initial states, and \( F \) is a set of final states. Note that this is a simple extension of VPA where neutral transitions are allowed to read neutral symbols or captures in \( X \). We define the runs as in VPA except the input in a EVPA is a ref-word \( w \in (\Sigma \cup \mathcal{C}_X) \), and we say that \( w \in \mathcal{L}(\mathcal{A}) \) if and only if there is an accepting run of \( \mathcal{A} \) on \( w \). Furthermore, \( \mathcal{A} \) is unambiguous if for every ref-word \( w \) there exists at most one accepting run of \( \mathcal{A} \) over \( w \). It is straightforward to see that this is a direct counterpart to visibly pushdown extraction grammars. Therefore, we can use the ideas in [5] to obtain a one-to-one conversion from one to another.

\[
\begin{align*}
\triangleright \text{ Claim 32.} & \quad \text{For a given VPEG } G \text{ there exists an EVPA } \mathcal{A}_G \text{ such that } \mathcal{L}(G) = \mathcal{L}(\mathcal{A}_G). \\
& \quad \text{Moreover, } \mathcal{A}_G \text{ is unambiguous if and only if } G \text{ is unambiguous, and } \mathcal{A}_G \text{ can be constructed in time } \mathcal{O}(|G|). \\
\end{align*}
\]

\textbf{Proof.} Let \( G = (X, V, \Sigma, S, P) \) be a VPEG. We construct a EVPA \( \mathcal{A}_G = (X, Q, \Sigma, \Gamma, \Delta, I, F) \) such that \( \mathcal{L}(G) = \mathcal{L}(\mathcal{A}_G) \) using an almost identical construction to the one in Theorem 6 of [5]. The only differences arise in that our structure is defined for well-nested words, so it can be slightly simplified, and in the case where a production is of the form \( X \rightarrow aY \), for which we add the possibility that \( a \in \mathcal{C}_X \). This construction provides one transition in \( \Delta \) per production in \( P \), and in some cases it needs to check if a variable is nullable. Checking if a single variable is nullable is costly, but by a constant number of traversals in \( P \) it is possible to check which variables in \( X \) are nullable or not, which can be done before building \( \Delta \). Therefore, this construction can be done in time \( \mathcal{O}(|P|) \). Finally, \( \mathcal{A}_G \) is unambiguous if and only if \( G \) is unambiguous, which is another consequence of Theorem 6 of [5].

\textcolor{red}{\triangleright}

Here we define the spanner \([\mathcal{A}]\) for a given EVPA \( \mathcal{A} \) identically to the definition for an extraction grammar. Note that from the proof it also follows that if \( G \) is functional, then \( \mathcal{A}_G \) is functional as well.

For the next part of the proof assume that \( \mathcal{A}_G \) is unambiguous. We will show that for an EVPA \( \mathcal{A} \) and stream \( \mathcal{S} \), the set \([\mathcal{A}](d)\), can be enumerated with output-linear delay and update-time \( \mathcal{O}(|[\mathcal{A}](d)|^3) \), for \( d = S[1, n] \). Towards this goal, we will start with an unambiguous \( \mathcal{A}_G = (X, Q, \Sigma, \Gamma, \Delta, I, F) \) and convert it into a VPT \( \mathcal{T}_G \) with output symbol set \( 2^X \) and use our algorithm to enumerate the set \( [\mathcal{T}_G](w) \) where \( d' = d \# \), using a dummy symbol \#. Each element \( w \in [\mathcal{T}_G](d') \) can then be converted into a mapping \( \mu \in [G](d) \) after it is given as output in time \( \mathcal{O}(|\mu|) \).

Let \( \mathcal{T}_G = (Q', \Sigma', \Gamma, \Omega, \Delta', I, F') \) where \( Q' = Q \cup \{q_f\}, \Sigma' = (\Sigma^*, \Sigma^\prime, \Sigma^\dagger \cup \#) \) such that \( \Sigma^\dagger \# = \Sigma^\dagger \cup \{\#\} \), \( \Omega = 2^X \cup \{\varepsilon\} \) and \( F' = \{q_f\} \). To define \( \Delta' \) we introduce a merge operation on a path over \( \mathcal{A}_G \). This is defined for any non-empty sequence of transitions \( t = (p_1, v_1, q_1)(p_2, v_2, p_2)\cdots(p_m, v_m, q_m) \in \Delta^* \) such that \( v_i \in \mathcal{C}_X \) for \( i \in [1, m] \), and \( q_i = p_{i+1} \in [i, m - 1] \). If these conditions hold, we say that \( t \) is a v-path ending in \( p_m \). Let \( t \) be such a v-path and let \( S = \{v_1, \ldots, v_m\} \). For \( \alpha \in \Sigma^\prime \) and a transition \( (p, a, \gamma, q) \) such that \( p = q_m \), we define \( \text{merge}(t, (p, a, \gamma, q)) := (p_1, \alpha, S, \gamma, q) \). For \( \alpha \in \Sigma^* \) and a transition \( (p, a, \gamma, q) \) such that \( p = q_m \), we define \( \text{merge}(t, (p, a, q, \gamma)) := (p_1, \alpha, S, q, \gamma) \). For \( a \in \Sigma^\dagger \) and a transition \( (p, a, q) \) such that \( p = q_m \), we define \( \text{merge}(t, (p, a, q)) := (p_1, a, S, q) \). We now
We modify the enumeration algorithm slightly so that for each output which we deduce that the statement.

case is by determinizing the EVPA constructed in Claim 32. This can be done in time $O(\mu)$ the expected output in $O(\tau)$ traversal over the transitions in element in $A$ that exists only one mapping to the same position in in the order of the elements inside each group of contiguous captures, which will be asociated $r$. can be seen that for every accepting path of input/output pair that is accepted by $T$ there exists an accepting path in $A$ that can be built from the captures present in $r$.

To show accepting pairs for $T$ correspond to a valid counterpart in $A$ let $(d, \omega)$ be an input/output pair that is accepted by $T$. Note that $d = d'\#$ from our definition of $\Delta'$. It can be seen that for every accepting path of $T$ over $(d, \omega)$ there exists at least one ref-word $r$ built from $d$ and $\omega$. However, note that for every such ref-word $r$ the only difference may be in the order of the elements inside each group of contiguous captures, which will be asociated to the same position in $\mu'$. From this, it follows that for each accepting pair $(d, \omega)$ there exists only one mapping $\mu \in [A\Delta](d')$ that can be built from $(d, \omega)$.

The size of $\Delta$ is bounded by the number of valid v-paths there could exist in $A$. Recall that $A_G$ is functional, an thus every v-path in $A_G$ contains at most one instance of each element in $C_X$. From this it follows that the size of $T$ is in $O(|\Delta|2^{|C_X|})$. Furthermore, since the transitions in $\Delta$ form a DAG over $Q$, each of these v-paths can be found by a single traversal over $A_G$, so building $T$ takes time $O(|\Delta|)$.

By using the algorithm detailed in Section 6 we can enumerate the set $[T_G](d)$ with update-time $O(|T_G|^3)$ and output-linear delay. However, with a more fine-grained analysis of the algorithm, we note that the update-time is bounded by $|Q|^3|\Delta| \in O(|Q|^3|\Delta|2^{|C_X|})$. We modify the enumeration algorithm slightly so that for each output $\omega \in [T_G](d)$ we build the expected output in $[G](d)$. We do this by checking $w$ symbol by symbol and building a mapping $\mu \in [G](d)$, and this can be done in time $O(|\mu|)$. As the set $X$ is fixed, it follows that this enumeration can be done with update-time $O(|G|^3)$ and output-linear delay.

Finally, we adress the case where $G$ is an arbitrary VPEG. The way we deal with this case is by determining the EVPA constructed in Claim 32. This can be done in time $O(2^{2|A_G|})$. From here, we can follow the reasoning given for the unambiguous case to prove the statement.