Efficient Eager XPath Filtering over XML Streams

Kazuhito Hagio, Takashi Ohgami, Hideo Bannai, and Masayuki Takeda

Department of Informatics, Kyushu University, 744 Motooka, Nishi-ku, Fukuoka 819-0395, Japan
{kazuhito.hagio, takashi.oogami}@i.kyushu-u.ac.jp
{bannai, takeda}@inf.kyushu-u.ac.jp

Abstract. We address the embedding existence problem (often referred to as the filtering problem) over streaming XML data for Conjunctive XPath (CXP). Ramanan (2009) considered Downward CXP, a fragment of CXP that involves downward navigational axes only, and presented a streaming algorithm which solves the problem in $O(|P||D|)$ time using only $O(|P|height(D))$ bits of space, where $|P|$ and $|D|$ are the sizes of a query P and an XML data D, respectively, and $height(D)$ denotes the tree height of D. Unfortunately, the algorithm is *lazy* in the sense that it does not necessarily report the answer even after enough information has been gathered from the input XML stream. In this paper, we present an *eager* streaming algorithm that solves the problem with same time and space complexity. We also show the algorithm can be easily extended to Backward CXP a larger fragment of CXP.

1 Introduction

Efficient processing of XML streams is receiving much attention due to its growing range of applications such as stock and sports tickers, traffic information systems, electronic personalized newspapers, and entertainment delivery. Existing approaches assume that user interests are written as tree-shaped queries in *XPath*, a language for specifying the selection of element nodes within XML data trees. There are two variations of the problem: the embedding existence (EmbExist) and the query evaluation (QueryEval). The former is, given an XPath tree P and an XML data tree D, to determine whether there exists an embedding of P into D. The latter is, given P, D, and a node q_{out} of P, to determine the set of element nodes that q_{out} matches over all embeddings of P in D. A great deal of studies have been undertaken on the problems (see an excellent survey [1]). In this paper, we focus on EmbExist.

XPath supports a number of powerful modalities and it is rather expensive to process. In practice, many applications do not need the expressive power of the full language and use only a fragment of XPath. One such fragment is a conjunctive, navigational fragment named Conjunctive XPath (CXP). For non-streaming D, Gottlob et al. [2] and Ramanan [6] presented *in-memory* algorithms which solve QueryEval (and therefore EmbExist) for CXP in $O(|P||D|)$ time using $O(|D|)$ space. On the other hand, several studies have been undertaken on developing *streaming* algorithms for both the problems, with a restriction on navigational axes.

Downward CXP (DCXP) is a fragment of CXP where navigational axes are limited to the child and descendant axes. Ramanan [7] showed that for DCXP, there is a streaming algorithm which solves EmbExist in $O(|P||D|)$ time using only $O(|P|height(D))$ bits of space, where $height(D)$ denotes the tree height of D. Gou and Chirkova [3] also presented an algorithm which takes $O(|P||D|)$ time and $O(r(P,D)|P| \log height(D))$ bits of space, where $r(P,D)$ denotes the recursion depth

Kazuhito Hagio, Takashi Ohgami, Hideo Bannai, Masayuki Takeda: Efficient Eager XPath Filtering over XML Streams, pp. 30–44.
Proceedings of PSC 2011, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04870-2

of D w.r.t. Q[1]. Unfortunately, both the algorithms are *lazy* in the sense that they do not necessarily report the answer even after enough information has been gathered from input XML stream.

Main contribution. In this paper we present an *eager* streaming algorithm which solves EmbExist for DCXP in $O(|P||D|)$ time using $O(|P|\mathit{height}(D))$ bits of space. We then extend it to *Backward CXP* (BCXP), a larger fragment of CXP where some additional navigational axes are allowed.

The remainder of this paper is as follows. In Section 2 we define CXP and its fragments DCXP and BCXP, and then formulate our problem. In Section 3 we show a lazy algorithm which is essentially the same as the one presented by Ramanan in [7]. In Section 4 we describe how to modify the algorithm *eager*. In Section 5 we extend these two algorithms to BCXP. In Section 6 we mention related work and in Section 7 we conclude this paper.

2 Preliminaries

2.1 Notation

Let A be a finite alphabet. An element of A^* is called a *string*. A string y is said to be a *substring* of another string w if w can be written as $w = xyz$ for some strings x, z. For a string w, the i-th symbol of w is denoted by $w[i]$, and the substring of w that begins at position i and ends at position j is denoted by $w[i..j]$.

Let R, S be any binary relations on a set X. The *composition* of R and S is $R \circ S = \big\{\langle x, z\rangle \;\big|\; \langle x, y\rangle \in R \text{ and } \langle y, z\rangle \in S\big\}$. Let $R^0 = I_X = \{\langle x, x\rangle \mid x \in X\}$, and let $R^n = R \circ R^{n-1}$ for $n \geq 1$. Then, the *transitive closure* of R is $R^+ = \bigcup_{n=1}^{\infty} R^n$, and the *reflexive, transitive closure* of R is $R^* = \bigcup_{n=0}^{\infty} R^n$. The *inverse* of R is $R^{-1} = \{\langle x, y\rangle \mid \langle y, x\rangle \in R\}$. Let $R(y) = \{x \in X \mid \langle x, y\rangle \in R\}$.

2.2 XML data tree and XML data

Let Σ be a set of *tag names*. An *XML data tree* is an ordered tree with nodes v labeled by $\mathit{label}(v)$ in Σ, and is denoted by D. Let $\mathcal{N}_D$ denote the set of nodes in D. The cardinality of $\mathcal{N}_D$ is called the *size* of D and denoted by $|D|$. Let $<_{pre}$ denote the pre-order on $\mathcal{N}_D$.

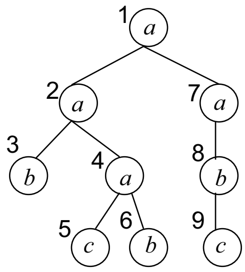
Let $\overline{\Sigma} = \{\bar{a} \mid a \in \Sigma\}$. For any $u \in \mathcal{N}_D$, let

$$\mathcal{S}(u) = \begin{cases} a\ \bar{a}, & u \text{ is a leaf;} \\ a\ \mathcal{S}(v_1)\cdots\mathcal{S}(v_k)\ \bar{a}, & u \text{ is an internal node with children } v_1, \ldots, v_k. \end{cases}$$

where $a = \mathit{label}(u)$. We note that $\mathcal{S}(u)$ is a string over $\Sigma \cup \overline{\Sigma}$. The *serialized representation* $\mathcal{S}(D)$ of an XML data tree D is defined to be $\mathcal{S}(r)$ where r is the root of D. The serialized representations of XML data trees are called the *XML data*. In this paper, we assume that the input XML data tree is given in the form of XML data, and identify an XML data tree D and its serialized representation $\mathcal{S}(D)$ if no confusion occurs. Thus we simply denote by $D[i]$ the symbol $\mathcal{S}(D)[i]$, and by $D[i..j]$ the substring $\mathcal{S}(D)[i..j]$, respectively. We often use N as the length of $\mathcal{S}(D)$.

An example of XML data tree and the corresponding XML data are shown in Fig. 1.

[1] Since $r(P, D) = O(\mathit{height}(D))$ the space requirement can be $O(|P|\mathit{height}(D) \log \mathit{height}(D))$ which is worse than $O(|P|\mathit{height}(D))$. On the other hand, $r(P, D)$ is often smaller than $\mathit{height}(D)$ in some practical cases.



t	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
$D[t]$	a	a	b	$\bar{b}$	a	c	$\bar{c}$	b	$\bar{b}$	$\bar{a}$	$\bar{a}$	a	b	c	$\bar{c}$	$\bar{b}$	$\bar{a}$	$\bar{a}$

Figure 1. An example of XML data tree D is displayed on the left and its serialized representation $D[1..N]$ is shown on the right. We have $|D| = |\mathcal{N}_D| = 9$ and $N = 18$. The node numbered 4 of D corresponds to interval $[5, 10]$ of $D[1..N]$.

In XML data $D = D[1..N]$, every $v \in \mathcal{N}_D$ corresponds to an interval $[s(v), e(v)]$ with $1 \le s(v) < e(v) \le N$ such that v *starts at position* $s(v)$ and *ends at position* $e(v)$. We note that symbols $a \in \Sigma$ and $\bar{a} \in \overline{\Sigma}$, respectively, correspond to start and end tags of XML data.

Proposition 1. *For any $u, v \in \mathcal{N}_D$, $u <_{pre} v \iff s(u) < s(v)$.*

2.3 Conjunctive XPath, embedding, occurrence

We consider two binary relations on $\mathcal{N}_D$

$$\begin{aligned}\mathsf{child} &= \{\langle u, v\rangle \mid u \text{ is a child of } v\},\\ \mathsf{nextSib} &= \{\langle u, v\rangle \mid u \text{ is the next sibling of } v\}\end{aligned}$$

and their inverses $\mathsf{parent} = \mathsf{child}^{-1}$ and $\mathsf{prevSib} = \mathsf{nextSib}^{-1}$. These four binary relations and their transitive and reflexive transitive closures are called *axes*. Additionally, the identity $\mathsf{self} = \{\langle v, v\rangle \mid v \in \mathcal{N}_D\}$, the abbreviation $\mathsf{following} = \mathsf{child}^* \circ \mathsf{nextSib}^+ \circ \mathsf{parent}^*$ and its inverse $\mathsf{preceding} = \mathsf{following}^{-1}$ are also axes[2].

A *conjunctive XPath (CXP) tree* is an unordered tree such that

- the nodes p are labeled by $label(p) \in \Sigma \cup \{\star\}$, where $\star$ is a special symbol not in Σ; and
- the edges are labeled by axes.

Let P be a CXP tree. The *size* of P, denoted by $|P|$, is the number of nodes. Let $P.rt$ denote the root of P. For any non-root node q of P, let $\chi(q)$ denote the label of the edge between q and its parent. For a node q of P, let $sub(q)$ denote the subtree of P rooted at q. An *embedding* of P into D is a function φ that maps nodes of P to nodes of D such that

- $label(q) \in \{\star, label(\varphi(q))\}$ for any node q of P; and
- $\langle \varphi(p), \varphi(q)\rangle \in \chi(q)$ for any non-root node q of P with parent p.

We note that function φ is not necessarily an injection, unlike the standard setting of tree pattern matching (see, e.g. [4]). Figure 2 illustrates embeddings of CXP tree into XML data tree.

[2] The descendant, descendant-or-self, ancestor, ancestor-or-self, preceding-sibling, and following-sibling axes of XPath1.0 (`http://www.w3.org/TR/xml`) correspond to child^+, child^*, parent^+, parent^*, $\mathsf{prevSib}^+$, and $\mathsf{nextSib}^*$, respectively. We note that the original definition of XPath1.0 excludes $\mathsf{nextSib}$, $\mathsf{nextSib}^*$, $\mathsf{prevSib}$, and $\mathsf{prevSib}^*$.

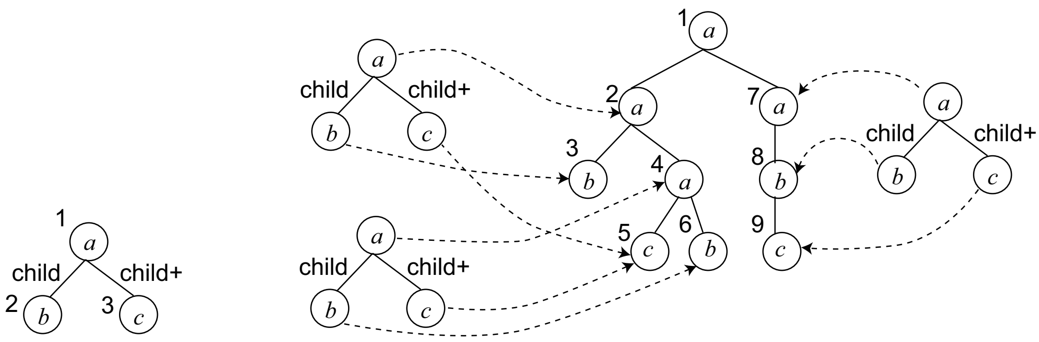


Figure 2. An example CXP tree is shown on the left, and its embeddings into the XML data tree of Fig. 1 are illustrated on the right.

A CXP tree P is said to *occur at* $v \in \mathcal{N}_D$ if there exists an embedding φ of P into D with $\varphi(P.rt) = v$. An *occurrence* of P in D is a node $v \in \mathcal{N}_D$ at which P occurs. Let $Occ(P, D)$ denote the set of occurrences of P in D.

A CXP tree P is said to be *unsatisfiable* if no node of D is an occurrence of P for any D, and *satisfiable*, otherwise. We assume that the input CXP tree is satisfiable throughout this paper.

2.4 Problem statement

*Problem 2 (*EmbExist*).* Given a CXP tree P and an XML data D, determine whether there exists an embedding of P into D.

*Problem 3 (*QueryEval*).* Given a CXP tree P, a node q_{out} of P, and an XML data D, compute $Eval(P, q_{out}, D) = \{\varphi(q_{out}) \mid \varphi \text{ is an embedding of } P \text{ into } D\}$.

EmbExist is often referred to as the filtering problem. The next is a slightly strengthened version of EmbExist.

*Problem 4 (*AllOcc*).* Given a CXP tree P and an XML data D, compute $Occ(P, D)$.

We note that AllOcc is essentially the same as EmbExist and is a special case of QueryEval where q_{out} is the root of P. In this paper we focus on AllOcc.

A *streaming algorithm* for AllOcc is an algorithm which scans an XML data $D = D[1..N]$ and emits, for every $x \in \mathcal{N}_D$, the pair $\langle x, b_x \rangle$ during one pass through $D[1..N]$, where b_x denotes a Boolean value indicating whether P occurs at x. A streaming algorithm for AllOcc is *eager* if it emits the pair $\langle x, b_x \rangle$ with minimum delay for every $x \in \mathcal{N}_D$.

2.5 Fragments of CXP

The *downward* axes are child, child$^+$, child*, and self. The *forward* (resp. *backward*) axes are the downward axes plus nextSib, nextSib$^+$, nextSib* and following (resp. prevSib, prevSib$^+$, prevSib* and preceding). The fragments of CXP with downward, forward and backward axes are denoted by DCXP, FCXP and BCXP, respectively. Figure 3 illustrates the fragments of CXP.

Theorem 5 ([7]). *There is a streaming algorithm that solves* AllOcc *for DCXP in* $O(|P||D|)$ *time using* $O(|P|height(D))$ *bits of space.*

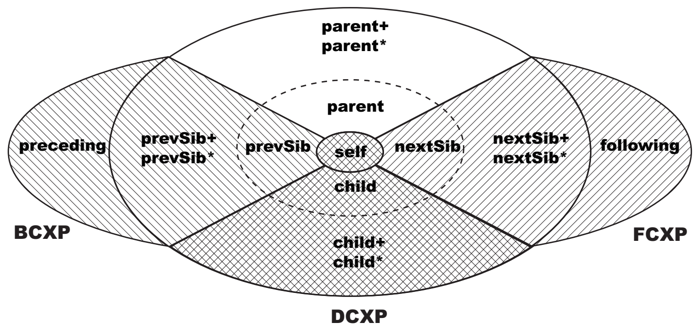


Figure 3. Fragments of CXP are illustrated.

3 Lazy Algorithm for DCXP

In this section, we describe a lazy algorithm for solving AllOcc for DCXP, which simplifies a predicate evaluator presented by Ramanan in [7]. Throughout this section D is any fixed XML data.

3.1 Introducing predicates M and T

Definition 6. *For any node p of a CXP tree P and any $u \in \mathcal{N}_D$, let*

$$M(p,u) = \mathsf{T} \Leftrightarrow sub(p) \textit{ occurs at } u.$$

Since $Occ(P,D)$ is the set of nodes $v \in \mathcal{N}_D$ such that $M(P.rt,v) = \mathsf{T}$, we consider computing the values $M(P.rt,v)$ for all $v \in \mathcal{N}_D$ for any P. For this purpose, we use another predicate $T(\cdot,\cdot)$ defined below.

For any non-root node q of P, let $sub^+(q)$ be the tree obtained from the tree $sub(q)$ by adding a new root node r with label $\star$ an edge from r to q labeled $\chi(q)$. (See an example in Fig. 4.)

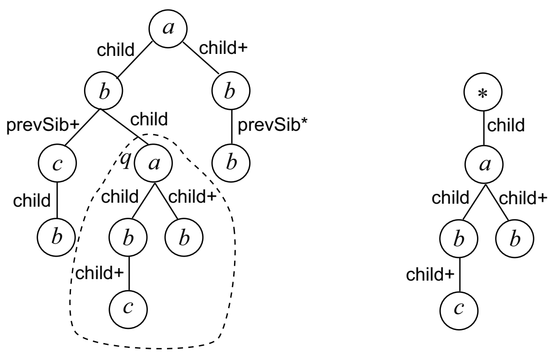


Figure 4. An example subtree $sub(q)$ of CXP tree is surrounded with broken line on the left, and the corresponding $sub^+(q)$ is displayed on the right.

Definition 7. *For any non-root node q of a CXP tree P and any $u \in \mathcal{N}_D$, let*

$$T(q,u) = \mathsf{T} \Leftrightarrow sub^+(q) \textit{ occurs at } u.$$

Examples of M and T can be found in Fig. 5.

$M(q, v)$

		v								
		1	2	3	4	5	6	7	8	9
	1	F	T	F	T	F	F	T	F	F
q	2	F	F	T	F	F	T	F	T	F
	3	F	F	F	F	T	F	F	F	T

$T(q, v)$

		v								
		1	2	3	4	5	6	7	8	9
	1									
q	2	F	T	F	T	F	F	T	F	F
	3	T	T	F	T	F	F	T	T	F

Figure 5. The values of functions M and T for the XML tree D and the CXP tree P of Fig. 2.

Proposition 8. *For any non-root node q of a CXP tree P and any $u \in \mathcal{N}_D$,*

$$T(q, u) = \bigvee_{\langle v,u \rangle \in \chi(q)} M(q, v).$$

We can prove the following lemma:

Lemma 9. *For any node p of a CXP tree P and any $u \in \mathcal{N}_D$,*

$$M(p, u) = \big(label(p) \in \{\star, label(u)\}\big) \wedge \big(\bigwedge_{q \text{ is a child of } p} T(q, u)\big).$$

Proof. Directly from the definitions of M and T. □

3.2 Algorithm

Now we assume that P is a DCXP tree. We have the following lemma:

Lemma 10. *For any node q of a DCXP tree P and any $u \in \mathcal{N}_D$,*

$$T(q, u) = \begin{cases} \bigvee_{v \in \mathsf{child}(u)} M(q, v), & \text{if } \chi(q) = \mathsf{child}; \\ \bigvee_{v \in \mathsf{child}(u)} (T(q, v) \vee M(q, v)), & \text{if } \chi(q) = \mathsf{child}^+; \\ M(q, u) \vee \bigvee_{v \in \mathsf{child}(u)} T(q, v), & \text{if } \chi(q) = \mathsf{child}^*; \\ M(q, u), & \text{if } \chi(q) = \mathsf{self}. \end{cases}$$

Proof. By Proposition 8. □

Algorithm 1 follows directly from Lemmas 9 and 10. It essentially processes the nodes v of D in post-order. Arrays $M[\cdot, \cdot]$ and $T[\cdot, \cdot]$ are used to store the values of $M(\cdot, \cdot)$ and $T(\cdot, \cdot)$, respectively. We note that the values of $M(q, u)$ and $T(q, u)$ should be stored only for the ancestors u of current v, and therefore the space requirement for M and T is $O(|P| height(D))$ bits.

Theorem 11. *Algorithm 1 (lazily) solves* AllOcc *for DCXP in $O(|P||D|)$ time using $O(|P| height(D))$ bits of space.*

4 Eager Algorithm for DCXP

In this section we modify Algorithm 1 to be eager.

Algorithm 1: A lazy streaming algorithm that solves AllOcc for DCXP.

```
class LazyDCXP
    void run(CXPTree P; XMLData D[1..N])
        initialize M[·,·] and T[·,·] to F;
        for t := 1 to N do
            if D[t] ∈ Σ then  do nothing;
            if D[t] ∈ Σ̄ then
                let v be the node of D with t = e(v);
                endTag(P, v);

    void endTag(CXPTree P; XMLDataNode v)
        foreach node q of P in post-order do  updateM(q, v);

    void updateM(CXPTreeNode q; XMLDataNode v)
        M[q,v] := (label(q) ∈ {⋆, label(v)}) ∧ (⋀_{c is a child of q} T[c,v]); // by Lemma 9
        if q is root node then
            emit ⟨v, M[q,v]⟩;
        else
            updateT_V(q, v);

    void updateT_V(CXPTreeNode q; XMLDataNode v) // by Lemma 10
        if χ(q) = self    then  T[q,v] := M[q,v];
        if χ(q) = child*  then  T[q,v] := T[q,v] ∨ M[q,v];
        if v has parent u then
            if χ(q) = child   then T[q,u] := T[q,u] ∨ M[q,v];
            if χ(q) = child+  then T[q,u] := T[q,u] ∨ M[q,v] ∨ T[q,v];
            if χ(q) = child*  then T[q,u] := T[q,u] ∨ T[q,v];
```

4.1 Precise definition of eagerness

First, we formally define what is meant by *eager*. To represent predicates M and T for varying D, we explicitly specify superscript D as M^D and T^D.

Definition 12. *Let P be any CXP tree and let $D = D[1..N]$ be an XML data. For any node p of P, any $u \in \mathcal{N}_D$, and any $t \in [s(u), N]$, let*

$$M_t^D(p,u) = \left\{ M^{D'}(p,u) \;\middle|\; D' \text{ is an XML data with } D'[1..t] = D[1..t] \right\}.$$

Intuitively, $M_t^D(p,u)$ is the set of possible values of $M^D(p,u)$ just after reading the t-th symbol of $D[1..N]$. It is thus a subset of $\{\mathsf{T},\mathsf{F}\}$, and can be either $\{\mathsf{T}\}$, $\{\mathsf{F}\}$, or $\{\mathsf{T},\mathsf{F}\}$. Let us denote the values $\{\mathsf{T}\},\{\mathsf{F}\},\{\mathsf{T},\mathsf{F}\}$ simply by $\mathsf{T},\mathsf{F},\mathsf{U}$, respectively. In what follows, we omit superscript D and simply write as $M_t(p,u)$ if no confusion occurs. Fig. 6 illustrates the values of M_t for the XML data tree D and the CXP tree P of Fig. 2 where $t = 1,\dots,18$.

Definition 13. *For any node p of a CXP tree P and for any $u \in \mathcal{N}_D$, let $\mathsf{time}_M(p,u)$ be the smallest integer $t \in [s(u), N]$ such that $M_t(p,u) \neq \mathsf{U}$.*

Proposition 14. *$M_t(p,u) = \mathsf{U}$ for any $t \in [s(u), \mathsf{time}_M(p,u) - 1]$ and $M_t(p,u) = M(p,u) \neq \mathsf{U}$ for any $t \in [\mathsf{time}_M(p,u), N]$.*

We are now ready to define the concept of eagerness.

		$M_t(q,v)$									$T_t(q,v)$								
		1 [1, 18]	2 [2, 11]	3 [3, 4]	4 [5, 10]	5 [6, 7]	6 [8, 9]	7 [12, 17]	8 [13, 16]	9 [14, 15]	1 [1, 18]	2 [2, 11]	3 [3, 4]	4 [5, 10]	5 [6, 7]	6 [8, 9]	7 [12, 17]	8 [13, 16]	9 [14, 15]
$t=1$	1	U																	
	2	**F**									U								
	3	**F**									U								
$t=2$	1	U	U																
	2	F	**F**								U	U							
	3	F	**F**								U	U							
$t=3$	1	U	U	**F**															
	2	F	F	**T**							U	**T**	U						
	3	F	F	**F**							U	U	U						
$t=4$	1	U	U	F															
	2	F	F	T							U	T	**F**						
	3	F	F	F							U	U	**F**						
$t=5$	1	U	U	F	U														
	2	F	F	T	**F**						U	T	F	U					
	3	F	F	F	**F**						U	U	F	U					
$t=6$	1	U	**T**	F	U	**F**													
	2	F	F	T	F	**F**					U	T	F	U	U				
	3	F	F	F	F	**T**					**T**	**T**	F	**T**	U				
$t=7$	1	U	T	F	U	F													
	2	F	F	T	F	F					U	T	F	U	**F**				
	3	F	F	F	F	T					T	T	F	T	**F**				
$t=8$	1	U	T	F	**T**	F	**F**												
	2	F	F	T	F	F	**T**				U	T	F	**T**	F	U			
	3	F	F	F	F	T	**F**				T	T	F	T	F	U			
$t=9,10,11$	1	U	T	F	T	F	F												
	2	F	F	T	F	F	T				U	T	F	T	F	**F**			
	3	F	F	F	F	T	F				T	T	F	T	F	**F**			
$t=12$	1	U	T	F	T	F	F	U											
	2	F	F	T	F	F	T	**F**			U	T	F	T	F	F	U		
	3	F	F	F	F	T	F	**F**			T	T	F	T	F	F	U		
$t=13$	1	U	T	F	T	F	F	U	**F**										
	2	F	F	T	F	F	T	F	**T**		U	T	F	T	F	F	**T**	U	
	3	F	F	F	F	T	F	F	**F**		T	T	F	T	F	F	U	U	
$t=14$	1	U	T	F	T	F	F	**T**	F	**F**									
	2	F	F	T	F	F	T	F	T	**F**	U	T	F	T	F	F	T	U	U
	3	F	F	F	F	T	F	F	F	**T**	T	T	F	T	F	F	**T**	**T**	U
$t=15$	1	U	T	F	T	F	F	T	F	F									
	2	F	F	T	F	F	T	F	T	F	U	T	F	T	F	F	T	U	**F**
	3	F	F	F	F	T	F	F	F	T	T	T	F	T	F	F	T	T	**F**
$t=16,17$	1	U	T	F	T	F	F	T	F	F									
	2	F	F	T	F	F	T	F	T	F	U	T	F	T	F	F	T	**F**	F
	3	F	F	F	F	T	F	F	F	T	T	T	F	T	F	F	T	T	F
$t=18$	1	**F**	T	F	T	F	F	T	F	F									
	2	F	F	T	F	F	T	F	T	F	**F**	T	F	T	F	F	T	F	F
	3	F	F	F	F	T	F	F	F	T	T	T	F	T	F	F	T	T	F

Figure 6. The values of functions M_t and T_t for the XML data tree D and the CXP tree P of Fig. 2, where $t = 1, \ldots, 18$. Value changes from the previous t are emphasized in boldface.

Definition 15. *A streaming algorithm that solves* AllOcc *is* eager *if, for every* $u \in \mathcal{N}_D$ *it emits* $\langle u, M(P.rt, u)\rangle$ *just after processing* $D[t^*]$ *where* $t^* = \mathsf{time}_M(P.rt, u)$.

For time_M, we can prove the following:

Proposition 16. *If* P *is a BCXP tree, then* $\mathsf{time}_M(p, u) \in [s(u), e(u)]$ *for any node* p *of* P *and for any* $u \in \mathcal{N}_D$.

Proof. Let φ be any embedding of $sub(p)$ into D with $\varphi(p) = u$, if exists. Since the axes of P are limited to backward ones, for any node q of $sub(p)$, $\varphi(q) \in \mathsf{preceding}(u) \cup \mathsf{child}^*(u)$ and therefore $e(\varphi(q)) \leq e(u)$. $\square$

4.2 Introducing T_t

We extend the Boolean operations $\wedge, \vee, \neg$ to domain $\{\mathsf{T}, \mathsf{F}, \mathsf{U}\}$ by: $\mathsf{T} \wedge \mathsf{U} = \mathsf{U} \wedge \mathsf{T} = \mathsf{U}$, $\mathsf{T} \vee \mathsf{U} = \mathsf{U} \vee \mathsf{T} = \mathsf{T}$, $\mathsf{F} \wedge \mathsf{U} = \mathsf{U} \wedge \mathsf{F} = \mathsf{F}$, $\mathsf{F} \vee \mathsf{U} = \mathsf{U} \vee \mathsf{F} = \mathsf{U}$, and $\mathsf{U} \wedge \mathsf{U} = \mathsf{U} \vee \mathsf{U} = \neg \mathsf{U} = \mathsf{U}$. For convenience, let $M_t(p, u) = \mathsf{U}$ for any $t \in [0, \ldots, s(u) - 1]$, although $M_t(p, u)$ is undefined for such t.

Definition 17. *For any non-root node* q *of a CXP tree* P*, any* $u \in \mathcal{N}_D$*, and for any* $t \in [s(u), N]$*, let*

$$T_t(q, u) = \bigvee_{\langle v,u\rangle \in \chi(q)} M_t(q, v).$$

Then we have:

Lemma 18. *For any node* p *of a CXP tree* P*, for any* $u \in \mathcal{N}_D$*, and for any* $t \in [0, N]$*,*

$$M_t(p, u) = \big(label(p) \in \{\star, label(u)\}\big) \wedge \big(\bigwedge_{q \text{ is a child of } p} T_t(q, u)\big).$$

Proof. By Lemma 9 and the definitions of M_t and T_t. $\square$

Define $\mathsf{time}_T(p, u)$ in a way similar to $\mathsf{time}_M(p, u)$. Then:

Proposition 19. *If* P *is a BCXP tree, then the following statements hold for any node* p *of* P *and for any* $u \in \mathcal{N}_D$.

- $\mathsf{time}_T(p, u) \in [s(u), e(u)]$.
- *If* $\chi(p) \in \{\mathsf{prevSib}, \mathsf{prevSib}^+, \mathsf{preceding}\}$ *then* $\mathsf{time}_T(p, u) = s(u)$.
- *If* $\chi(p) \in \{\mathsf{child}, \mathsf{child}^+, \mathsf{child}^*, \mathsf{self}, \mathsf{prevSib}^*\}$ *and* $T(p, u) = \mathsf{F}$ *then* $\mathsf{time}_T(p, u) = e(u)$ *(due to the assumption that* P *is satisfiable).*

Proof. Let φ be any embedding of $sub^+(p)$ into D with $\varphi(p') = u$ where p' is the parent of p, if exists. Since the axes of P are limited to backward ones, for any node q of $sub^+(p)$, $\varphi(q) \in \mathsf{preceding}(u) \cup \mathsf{child}^*(u)$ and therefore $e(\varphi(q)) \leq e(u)$. Thus we have $\mathsf{time}_T(p, u) \in [s(u), e(u)]$. Suppose $\chi(p) \in \{\mathsf{prevSib}, \mathsf{prevSib}^+, \mathsf{preceding}\}$. Then, $e(\varphi(q)) \leq e(\varphi(p)) < s(\varphi(p')) = s(u)$ and we have $\mathsf{time}_T(p, u) = s(u)$. Suppose $\chi(p) \in \{\mathsf{child}, \mathsf{child}^+, \mathsf{child}^*, \mathsf{self}, \mathsf{prevSib}^*\}$ and $T(p, u) = \mathsf{F}$. There is a possibility that a descendant v of u (possibly $u = v$) appears that makes $T(p, u)$ T until reading the end-tag of u. Thus we have $\mathsf{time}_T(p, u) = e(u)$. $\square$

4.3 Algorithm

Again, we restrict ourselves to the DCXP trees. We have:

Lemma 20. *For any node q of a DCXP tree P, for any $u \in \mathcal{N}_D$, and for any $t \in [s(u), e(u)]$,*

$$T_t(q,u) = T_{t-1}(q,u) \vee \bigvee_{\langle v,u\rangle \in \chi(q) \text{ and } t \in [s(v),e(v)]} ((M_{t-1}(q,v) = \mathsf{U}) \wedge (M_t(q,v) = \mathsf{T})).$$

Proof. Let $\langle v,u\rangle \in \chi(q)$. Since $\chi(q) \in \{\mathsf{child}, \mathsf{child}^+, \mathsf{child}^*, \mathsf{self}\}$, we have $[s(v), e(v)] \subseteq [s(u), e(v)]$. The lemma follows from Definition 17. □

Our eager algorithm follows from Lemmas 18 and 20. It can be summarized as Algorithm 2. It initializes the entries of arrays M and T by U and then incrementally rewrites them to T or F so that $M[q,u]$ and $T[q,u]$ are, respectively, identical to $M_t(q,u)$ and $T_t(q,u)$ for every $t \in [s(u), N]$. When a node v is found such that $M[q,v]$ just changes from U to T for some q with $\chi(q) = \mathsf{child}$ (resp. child^+, child^*, and self), it rewrites $T[q,u]$ for the parent u of v (resp. a proper ancestor u of v, an ancestor u of v, and $u = v$ itself). Whenever $T[q,u]$ changes, we evaluate $M[p,u]$ for parent p of q, and if $M[p,u]$ changes into T then we repeat this process. When a node v' is found such that $M[P.rt, u] \neq \mathsf{U}$, we output the pair $\langle u, M[P.rt, u]\rangle$.

Let us call the *t-th operating cycle* the t-th iteration of the **for**-loop in function *run*() of Algorithm 2.

Lemma 21. *For any DCXP tree P, after the t-th operating cycle of Algorithm 2, the value of $M[p,u]$ is identical to $M_t(p,u)$ for any node p of P and for any ancestor u of v, where v is the node of D such that $t = s(v)$ or $t = e(v)$.*

Proof. When p is a leaf, Lemma 18 implies that $M_t(p,u) = M_{s(u)}(p,u) = (\mathit{label}(p) \in \{\star, \mathit{label}(u)\})$ for any $t \in [s(u), N]$. At $t = s(u)$, the algorithm sets $M[p,u]$ to $(\mathit{label}(p) \in \{\star, \mathit{label}(u)\})$ in execution of $\mathit{updateM}(p,u)$ and then never changes it. Thus $M[p,u]$ holds $M_t(p,u)$ for $t \in [s(u), N]$ for leaves p of P. For internal nodes p, $M_t(p,u)$ depends on the values $T_t(q,u)$ for the children q of p. The algorithm invokes $\mathit{updateM}(p,u)$ whenever $T[q,u]$ changes from U into T, and each execution of $\mathit{updateM}(p,u)$ updates the value $M[p,u]$ according to Lemma 18. Thus $M[p,u]$ correctly holds $M_t(p,u)$ if $T[q,u]$ correctly holds $T_t(q,u)$ for every child q of p.

In execution of $\mathit{updateM}$, the algorithm invokes *liftUp* and updates $T[q,u]$ from $T_{t-1}(q,u)$ to $T_t(q,u)$ according to Lemma 20, whenever $M[q,v]$ changes from U into T for a child v of u (resp. a proper descandant v of u, a descendant v of u, and $u = v$ itself), if $\chi(p) = \mathsf{child}$ (resp. child^+, child^*, and self). Hence $T[q,u]$ correctly holds $T_t(q,u)$. □

Lemma 22. *Algorithm 2 runs in $O(|P||D|)$ time using $O(|P|\mathit{height}(D))$ bits of space.*

Proof. The space complexity is $O(|P|\mathit{height}(D))$ bits as for Algorithm 1. To estimate the time complexity, we have only to consider the total cost of executing *liftUp*. We note that *liftUp* is invoked only when the value $M[q,v]$ is changed from U into T and that the value $T[q,v]$ is changed from U into T in each execution of the **while**-loop in *liftUp*. Thus the total time is $O(|P||D|)$. □

Theorem 23. *Algorithm 2 eagerly solves* AllOcc *for DCXP in $O(|P||D|)$ time using $O(|P|\mathit{height}(D))$ bits of space.*

Proof. By Lemmas 21 and 22. □

Algorithm 2: An eager streaming algorithm that solves AllOcc for DCXP.

```
class EagerDCXP
    void run(CXPTree P; XMLData D[1..N])
        initialize M[·,·] and T[·,·] to U;
        for t := 1 to N do
            if D[t] ∈ Σ then
                let v be the node of D with t = s(v);
                startTag(P, v);
            if D[t] ∈ Σ̄ then
                let v be the node of D with t = e(v);
                endTag(P, v);

    void startTag(CXPTree P; XMLDataNode v)
        foreach node q of P in post-order do updateM(q, v);

    void endTag(CXPTree P; XMLDataNode v)
        foreach node q of P in post-order do
            if M[q, v] = U then updateM(q, v);
            if T[q, v] = U then T[q, v] := F;

    void updateM(CXPTreeNode q; XMLDataNode v)
        M[q, v] := (label(q) ∈ {⋆, label(v)}) ∧ (⋀_{c is a child of q} T[c, v]);
        if q is root node then
            if M[q, v] ≠ U then emit the pair ⟨v, M[q, v]⟩;
        else
            if M[q, v] = T then updateT_V(q, v);

    void updateT_V(CXPTreeNode q; XMLDataNode v)
        let u be the parent of v;
        if χ(q) = child   then liftUp(q, u, T);
        if χ(q) = child⁺  then liftUp(q, u, F);
        if χ(q) = child*  then liftUp(q, v, F);
        if χ(q) = self    then liftUp(q, v, T);

    void liftUp(CXPTreeNode q; XMLDataNode u; bool once)
        let p be the parent of q;
        while u ≠ nil and T[q, u] = U do
            T[q, u] := T; updateM(p, u);
            if once then return;
            u := the parent of u;
```

5 Extension to BCXP

5.1 Lazy algorithm for BCXP

The statement of Lemma 10 is extended to BCXP trees by adding four cases:

Lemma 24. *For any node q of a BCXP tree P and any $v \in \mathcal{N}_D$,*

$$T(q,v) = \begin{cases} M(q,w), & \text{if } \chi(q) = \mathsf{prevSib}; \\ T(q,w) \vee M(q,w), & \text{if } \chi(q) = \mathsf{prevSib}^+; \\ T(q,w) \vee M(q,v), & \text{if } \chi(q) = \mathsf{prevSib}^*; \\ T(q,z) \vee ((z \notin \mathsf{parent}(v)) \wedge M(q,z)), & \text{if } \chi(q) = \mathsf{preceding}, \end{cases}$$

where w is the previous sibling of v and z is the previous node of v w.r.t. $<_{pre}$. (Let $M(q,w) = T(q,w) = \mathsf{F}$ when w does not exist and let $M(q,z) = T(q,z) = \mathsf{F}$ when z does not exist.)

Proof. It is rather straightforward in the cases of $\chi(q) = \mathsf{prevSib}$, $\mathsf{prevSib}^+$, and $\mathsf{prevSib}^*$. We consider the case of $\chi(q) = \mathsf{preceding}$. Let z be the previous node of u w.r.t. $<_{pre}$. There are two cases.
Case 1: When v is not a leftmost sibling. Let w be the immediately left sibling of v. Then z is the rightmost descendant of w. In this case we have $\mathsf{preceding}(v) = \mathsf{preceding}(z) \cup \{z\}$.
Case 2: When v is a leftmost sibling. Then z is the parent of v. In this case we have $\mathsf{preceding}(v) = \mathsf{preceding}(z)$. □

Based on Lemmas 9 and 24, Algorithm 1 is extended as Algorithm 3 to cope with BCXP trees. What needs to be stored are (1) the values of $M(q,u)$ and $T(q,u)$ for the ancestors u of v; (2) the values of $M(q,w)$ and $T(q,w)$ for the previous sibling w of v; and (3) the values of $M(q,z)$ and $T(q,z)$ for the previous node z of v w.r.t. $<_{pre}$. The space requirement for M and T is still $O(|P|\mathit{height}(D))$ bits.

Theorem 25. *Algorithm 3 (lazily) solves* AllOcc *for BCXP in $O(|P||D|)$ time using $O(|P|\mathit{height}(D))$ bits of space.*

Algorithm 3: A lazy streaming algorithm that solves AllOcc for BCXP.

```
LazyBCXP extends LazyDCXP
    // methods run(), endTag(), updateT_V() inherit from LazyDCXP
    // method updateM() overrides the one in LazyDCXP
    // method updateT_H() is a newly added method
    void updateM(CXPTreeNode q; XMLDataNode v)
        M[q,v] := (label(q) ∈ {⋆, label(v)}) ∧ (⋀_{c is a child of q} T[c,v]);
        if q is root node then
            emit ⟨v, M[q,v]⟩;
        else
            updateT_V(q,v);
            updateT_H(q,v); // inserted

    void updateT_H(CXPTreeNode q; XMLDataNode v) // by Lemma 24
        if v has previous sibling w then
            if χ(q) = prevSib   then T[q,v] := M[q,w];
            if χ(q) = prevSib^+ then T[q,v] := T[q,w] ∨ M[q,w];
            if χ(q) = prevSib^* then T[q,v] := T[q,w];
        if χ(q) = prevSib^* then T[q,v] := T[q,v] ∨ M[q,v];
        if v has previous node z w.r.t. <_pre then
            if χ(q) = preceding then
                if v has previous sibling then T[q,v] := T[q,z] ∨ M[q,z];
                else T[q,v] := T[q,z];
```

Algorithm 4: An eager streaming algorithm that solves AllOcc for BCXP.

1 EagerBCXP **extends** EagerDCXP
  // methods *run*(), *endTag*(), *updateM*(), *liftUp*() inherit from EagerDCXP
  // methods *startTag*(), $updateT_V()$ override the ones in EagerDCXP
2   **void** *startTag*(CXPTree P; XMLDataNode v)
3   **foreach** node q of P in post-order **do**
4       $updateM(q, v)$;
5   LazyBCXP::$updateT_H(q, v)$; // added
6   **void** $updateT_V$(CXPTreeNode q; XMLDataNode v)
7   let u be the parent of v;
8   **if** $\chi(q) = \mathsf{child}$ **then** $liftUp(q, u, \mathsf{T})$;
9   **if** $\chi(q) = \mathsf{child}^+$ **then** $liftUp(q, u, \mathsf{F})$;
10   **if** $\chi(q) = \mathsf{child}^*$ **then** $liftUp(q, v, \mathsf{F})$;
11   **if** $\chi(q) = \mathsf{self}$ **then** $liftUp(q, v, \mathsf{T})$;
12   **if** $\chi(q) = \mathsf{prevSib}^*$ **then** $liftUp(q, v, \mathsf{T})$; // added

5.2 Eager algorithm for BCXP

Lemma 26. *For any non-root node q of a BCXP tree P with $\chi(q) \in \{\mathsf{prevSib}, \mathsf{prevSib}^+, \mathsf{prevSib}^*, \mathsf{preceding}\}$, for any $v \in \mathcal{N}_D$, and for any $t \in [s(v), N]$,*

$$T_t(q, v) = \begin{cases} M_{s(v)-1}(q, w), & \text{if } \chi(q) = \mathsf{prevSib}; \\ T_{s(v)-1}(q, w) \vee M_{s(v)-1}(q, w), & \text{if } \chi(q) = \mathsf{prevSib}^+; \\ T_{s(v)-1}(q, w) \vee M_t(q, v), & \text{if } \chi(q) = \mathsf{prevSib}^*; \\ T_{s(v)-1}(q, z) \vee ((z \notin \mathsf{parent}(v)) \wedge M_{s(v)-1}(q, z)), & \text{if } \chi(q) = \mathsf{preceding}, \end{cases}$$

where w is the previous sibling of v and z is the previous node of v w.r.t. $<_{pre}$.

Proof. Recall Lemma 24. By Propositions 16 and 19, we have $T(q, v) = T_{e(v)}(q, v) \neq \mathsf{U}$ and $M(q, v) = M_{e(v)}(q, v) \neq \mathsf{U}$. Since $e(w) \leq s(v) - 1$, we also have $M(q, w) = M_{e(w)}(q, w) = M_{s(v)-1}(q, w) \neq \mathsf{U}$ and $T(q, w) = T_{e(w)}(q, w) = T_{s(v)-1}(q, w) \neq \mathsf{U}$. Thus the lemma holds for the cases of $\chi(q) = \mathsf{prevSib}$, $\mathsf{prevSib}^+$, and $\mathsf{prevSib}^*$. Since $e(z) \leq s(v) - 1$, we have $M(q, z) = M_{e(z)}(q, z) = M_{s(v)-1}(q, z)$ and $T(q, z) = T_{e(z)}(q, z) = T_{s(v)-1}(q, z)$. Thus the lemma holds for the case of $\chi(q) = \mathsf{preceding}$. ☐

Our eager algorithm for BCXP is obtained as an extension of Algorithm 2 and is summarized as Algorithm 4. Lemma 26 tells us that for $\chi(q) = \mathsf{prevSib}$, $\mathsf{prevSib}^+$, or $\mathsf{preceding}$, the values $T_t(q, v)$ are determined when the start-tag of v is read, namely, at $t = s(v)$. Line 5 is thus added to *startTag*(). On the other hand, the values $T_t(q, v)$ can change until reading the end-tag of v for $\chi(q) = \mathsf{prevSib}^*$, and therefore Line 12 is added to $updateT_V()$.

The statement of Lemma 21 also holds for Algorithm 4:

Lemma 27. *For any BCXP tree P, after the t-th operating cycle of Algorithm 4, the value of $M[p, u]$ is identical to $M_t(p, u)$ for any node p of P and for any ancestor u of v, where v is the node of D such that $t = s(v)$ or $t = e(v)$.*

Proof. Comparing to the proof of Lemma 21, we have only to prove that $T[q, v]$ correctly holds $T_t(q, v)$ for any node q of P such that $\chi(q) = \mathsf{prevSib}$, $\mathsf{prevSib}^+$, $\mathsf{prevSib}^*$, or $\mathsf{preceding}$, assuming that $M[q, v]$ correctly holds $M_t(q, v)$.

In the three cases except $\chi(q) = \mathsf{prevSib}^*$, the values $T_t(q, v)$ are determined to T or F at $t = s(u)$, and the algorithm sets $T[q, v]$ to $T_t(q, v)$ by calling $update_H T(q, v)$ of $\mathsf{LazyBCXP}$. In the case of $\chi(q) = \mathsf{prevSib}^*$, the values $T_t(q, v)$ can be U even for $t > s(u)$. Thus, it invokes $liftUp(q, v)$ to update $T[q, v]$ whenever $M[q, v]$ changes into T in execution of $updateM$. □

Lemma 28. *Algorithm 4 runs in $O(|P||D|)$ time using $O(|P|height(D))$ bits of space.*

Proof. It requires only $O(|P|height(D))$ bits of space as Algorithm 3 does. To show its $O(|P||D|)$ time complexity, we have only to consider the total cost of executing *liftUp*. By the same discussion in the proof of Lemma 22, the total time is $O(|P||D|)$. □

Theorem 29. *Algorithm 4 eagerly solves* AllOcc *for BCXP in $O(|P||D|)$ time using $O(|P|height(D))$ bits of space.*

Proof. By Lemmas 27 and 28.

6 Related Work

By '*streaming algorithms*' we mean algorithms that perform the task in a single pass through the XML document, while keeping only small critical portions of the data in main memory for later use. Allowing $O(|D|)$ space enables us to store the whole streaming data in a buffer, to which any in-memory algorithm could be applied. Hence it is natural to allow only $o(|D|)$ space in the data complexity.

However, it is known that solving QueryEval over XML streams requires storing candidates for the answer nodes which take $\Omega(|D|)$ space in the worst case. For this reason, the space requirement is usually measured in terms of $maxcands(P, D)$, defined to be the maximum number of nodes of D that can be candidates for output, at any one instant.

Olteanu [5] presented an algorithm that uses $O(height(D)^2|P| + height(D) \cdot n \cdot maxcands(P, D))$ space and $O(height(D)|P||D|)$ time, where n is the number of location steps in P (i.e., the number of ancestors of q_{out}). Gou and Chirkova [3] presented an algorithm that uses $O(r(P, D)|P| + maxcands(P, D))$ space and $O(|P||D|)$ time, they claim. However, Ramanan [8] recently showed an $\Omega(n \cdot maxcands(P, D))$ lower bound for QueryEval for worst case P. This means that there is no algorithm for QueryEval that uses $O(f(height(D), |P|) + maxcands(P, D))$ space, for any function f, and therefore the claimed space upper bound of [3] is not achievable. On the other hand, Ramanan [7] presented an eager algorithm for QueryEval that runs in $O((|P| + height(D) \cdot n)|D|)$ time using $O(height(D)|P| + n \cdot maxcands(P, D))$ space. This space requirement matches the lower bound by Ramanan [8].

7 Conclusion

In this paper we addressed AllOcc. Efficiently solving AllOcc is of importance since it is useful not only in XML stream filtering but also in evaluating predicates in solving QueryEval. In such applications *eagerness* is a desirable feature. The previous AllOcc algorithm is due to Ramanan [7], which was presented as a predicate evaluator in his QueryEval algorithm. It takes only $O(|P|height(D))$ bits of space and $O(|P||D|)$ time but one drawback is its laziness as pointed out in [7]. We simplified the algorithm and then successfully modified it to be *eager*, without increasing time and space complexities.

References

1. M. Benedikt and C. Koch: *XPath leashed.* ACM Comput. Surv., 41(1) 2008.
2. G. Gottlob, C. Koch, and R. Pichler: *Efficient algorithms for processing XPath queries.* ACM TODS, 30(2) June 2005, pp. 444–491.
3. G. Gou and R. Chirkova: *Efficient algorithms for evaluating XPath over streams*, in SIGMOD'07, 2007, pp. 269–280.
4. P. Kilpeläinen and H. Mannila: *Ordered and unordered tree inclusion.* SIAM J. Comput., 24(2) 1995, pp. 340–356.
5. D. Olteanu: *SPEX: Streamed and progressive evaluation of XPath.* IEEE Transactions on Knowledge and Data Engineering, 19(7) 2007, pp. 934–949.
6. P. Ramanan: *Covering indexes for XML queries: Bisimulation − simulation = negation*, in VLDB'03, 2003, pp. 165–176.
7. P. Ramanan: *Worst-case optimal algorithm for XPath evaluation over XML streams.* J. Comput. Syst. Sci., 75(8) 2009, pp. 465–485.
8. P. Ramanan: *Memory lower bounds for XPath evaluation over XML streams.* J. Comput. Syst. Sci., 2010, in press.