Incremental XPath Evaluation

Incremental XPath Evaluation

Henrik Björklund[*]
TU Dortmund

Wouter Gelade[†]
Hasselt University and
Transnational University of
Limburg
School for Information
Technology

Marcel Marquardt
TU Dortmund

Wim Martens[‡]
TU Dortmund

ABSTRACT

We study the problem of incrementally maintaining an XPath query on an XML database under updates. The updates we consider are node insertion, node deletion, and node relabeling. Our main results are that downward XPath queries can be incrementally maintained in time $\mathcal{O}(\text{depth}(D) \cdot \text{poly}(Q))$ and conjunctive forward XPath queries in time $\mathcal{O}(\text{depth}(D) \cdot \log(\text{width}(D)) \cdot \text{poly}(Q))$, where D is the size of the database, Q the size of the query, and $\text{depth}(D)$ and $\text{width}(D)$ are the nesting depth and maximum number of siblings in the database, respectively. The auxiliary data structures for maintenance are linear in D and polynomial in Q in all these cases.

1. INTRODUCTION

The XPath language, proposed by the World Wide Web Consortium (W3C), is essentially a query language for selecting nodes in an XML document. As node-selection is one of the most basic operations on XML documents, XPath lies at the core of most of today's data processing languages for XML. For example, it forms an essential component of languages such as XQuery, XSLT, XML Schema (which uses XPath for defining keys), etc.

The most fundamental algorithmic question concerning XPath is *query evaluation*. That is, given an XPath query Q and an XML database D, return all elements that are selected by Q in D. The query evaluation problem for various fragments of XPath has been researched quite intensely over the last decade (see [4] for an overview). In this paper, we are interested in the *incremental XPath evaluation* problem. That is, given an XPath query Q and XML data D, we assume that the answer for Q on D is already known. However, when D is updated to D', we want to be able to infer the updated answer for Q on D' as quickly as possible. The idea is, of course, to maintain extra information such that the updated answer to Q on D' can be computed without having to re-evaluate Q from scratch.

There are many motivations for incremental XPath evaluation. An obvious motivating scenario comes from a database system that is interested in whether a trigger condition is satisfied in a database or not. If the precondition of the trigger is stated by means of an XPath query, the system may be interested in knowing very quickly after an update whether the event of the trigger needs to be carried out. Another scenario comes from exchanging data on the Web. When a community exchanges data, it is often the case that a certain user X is interested in the result of a fixed query on the data of another user Y. The data of Y may change often, while the interests of X remain the same. Of course, it would be beneficial for both parties if the query does not have to be completely recomputed every time Y's data changes. In such a setting, X may herself have a representation of the current result of her query. So, after an update of Y's data, it would be relevant for Y to be able to quickly determine the *changes* that X has to make to her old result, rather than sending her the complete new result, which may be much larger than the update.

Two Versions of the Problem.

Incrementally evaluating queries on a relational database is an intensively researched topic in database theory. In the literature, it is also known as *incremental view maintenance* (see, e.g., [20, 10]).

From our two motivating scenarios above, we can immediately infer two versions of the incremental XPath evaluation problem that we believe to be important in practice. The first is the *Boolean* version, and the second is the *view maintenance* version. In brief, in the Boolean version, we are simply interested in whether an XPath expression is satisfied or not after performing an update on the database. In the second version, the set of outputs of the XPath query is maintained and, after an update of the database, we want to compute an update to this set of outputs.

In this paper we focus mostly, but not exclusively, on the Boolean version of the problem. It is interesting in its own right and we also believe that it contains the core of the

[*]Supported by the DFG Grant SCHW678/3-1.

[†]Research Assistant of the Fund for Scientific Research – Flanders (Belgium).

[‡]Supported by a grant of the North-Rhine Westfalian Academy of Sciences.

ICDT 2009, March 23–25, 2009, Saint Petersburg, Russia.

view maintenance version. The view maintenance version is more general, but it turns out that the Boolean version is already quite challenging. We believe that the difficulties of the Boolean version need to be understood before view maintenance can be properly tackled.

Our Contributions.

Our main results are summarized in Table 1. Here, $|D|$ and $|Q|$ are the sizes of the XML document and the XPath query, respectively. The results are for Boolean incremental evaluation, and the time complexities *per update*. That is, we assume that a database gets an update of the form: relabel node x, delete the subtree rooted at x, or insert a node x in position y. Each such update can be handled by the algorithms in the time given in Table 1. Some of the time complexities contain a factor depth(D), i.e., the depth of the tree representation of D, rather than the actual size of D. We believe this to be very important for practical purposes, as the depth of D is, in practice, extremely small when compared to the actual size of D. The size-entries in the table show the size of the auxiliary data structure that is needed for query maintenance. The left column shows the XPath fragment for which the results hold. The full fragment "XPath Patterns" is defined in Section 2 but can be seen as Core XPath [4] with the addition of the next-sibling and previous-sibling axes. The other rows denote the fragments of XPath Patterns that only allow the listed operators and axes. Here, $\downarrow$, $\Downarrow$, $\rightarrow$, $\Rightarrow$ denote the axes child, descendant, nextsibling,[1] and following-sibling, respectively.

In case (3) in the table, we can also do view maintenance for the query, but only in a restricted form. Essentially, we can maintain the set of the nodes where the root of the query matches D. The good news is that all new updates to the set of results, for one single update, can be computed in time depth(D) $\cdot$ poly(Q). On the other hand, this is because all positions where the truth value of the root of the query changes lie on the path from the change in D to its root.

Finally, we want to bring the incremental XPath evaluation problem to the attention of the database theory community. To the best of our knowledge, this is the first theory paper that deals with this problem, and the first paper that provides worst-case upper bounds on the incremental evaluation problem for XPath.

Related work.

There are several practically oriented papers dealing with incremental XPath evaluation [12, 14, 16, 17]. None of these papers give any worst-case complexity bounds. The papers [12, 17] only consider leaf deletion and insertion, and [14] considers deletion and insertion of entire subtrees and can thus, in worst case, not be better than re-evaluation.

Incremental XPath evaluation can be seen as a generalization of the XPath evaluation problem on XML streams (see, e.g., [2, 9, 18]). In streaming XPath evaluation, one reads the XML document as a sequence of SAX-events, i.e., the sequence of opening and closing tags in the ordering in which they occur in the XML file. When viewing an XML document as a tree, this ordering corresponds to the depth-first left-to-right ordering of the tree. Streaming XPath evaluation can then be seen as incremental XPath evaluation in which the only update operation is that nodes can be added at the last position in the depth-first left-to-right ordering.

The incremental validation of XML schemas [1, 3] is closely related to our problems. In incremental schema evaluation, one is asked to maintain satisfaction of an XML document by an XML schema, where the document can be updated. Balmin et al. describe an algorithm for incrementally maintaining whether a tree is accepted by a tree automaton [1] which we use to infer some upper bounds in this paper.

The (non-incremental) XPath evaluation problem has been studied quite extensively in the literature [5, 6, 7, 8]. We refer to [4] for a more detailed overview. There is a large amount of work on indexing on XML documents (e.g, [13]), but indexing schemes are usually aimed towards answering a large class of XPath queries and require time at least linear in the database for complicated queries.

2. PRELIMINARIES

By Σ we always denote a fixed but infinite set of labels. We abstract away from actual XML documents by viewing them as rooted, ordered, finite, labeled, unranked trees, which are directed from the root downwards. That is, we consider trees with a finite number of nodes and in which nodes can have arbitrarily many children. We view an XML document D as a relational structure over a finite number of unary labeling relations $a(\cdot)$, where each $a \in \Sigma$, and binary relations child$(\cdot,\cdot)$ and next-sibling$(\cdot,\cdot)$. Here, $a(u)$ expresses that u is a node with label a, and child(u,v) (respectively, next-sibling(u,v)) expresses that v is a child (respectively, next sibling) of u. We also use the notations descendant(u,v) and following-sibling(u,v) for the respective transitive closures of the above relations. The label of a node u must be unique and is denoted by lab(u). We write Nodes(D) and Edges(D) for the sets of nodes and edges of a tree (document) D. As usual, Edges(D) is the set of pairs (u,v) such that child(u,v) holds in D. The root of D is denoted by root(D). We define the *size of* D, denoted by $|D|$, as the number of nodes of D.

Notice that we have an infinite set of labels from which our (finite) trees can choose. This reflects how trees occur in an XML-context: an XML tree is a finite structure, but there is no restriction on how it should be labeled (if no schema is provided).

2.1 XPath Patterns

We assume that the reader is familiar with XPath. Instead of working directly on XPath queries, we will usually use *XPath Patterns*, which are a more convenient technical tool for our algorithms. XPath Patterns allow us to reason about *nodes* and *edges* in the pattern. Our XPath Patterns will make use of *axes*. The XPath axes in this paper are fairly standard: self, child ($\downarrow$), descendant ($\Downarrow$), descendant-or-self ($\downarrow^*$), parent ($\uparrow$), ancestor ($\Uparrow$), ancestor-or-self ($\uparrow^*$), next-sibling ($\rightarrow$), following-sibling ($\Rightarrow$), previous-sibling ($\leftarrow$), preceding-sibling ($\Leftarrow$). We note that the remaining XPath axes (i.e., following and preceding) can be expressed by these axes using only a linear blow-up.

XPath Patterns are defined as follows.

DEFINITION 2.1. *An* XPath Pattern *is a rooted, unordered, finite, labeled tree in which the nodes and edges bear* types. *The type of a node u, denoted by type(u) can either be* label

[1]Next-sibling is strictly speaking not a primitive axis in XPath, but can be expressed using following-sibling::$*$[position() = 1].

	Fragment	Complexity	Section
(1)	XPath Patterns	Time: $\mathcal{O}(\text{polylog}(\|D\|)) \cdot 2^{\mathcal{O}(\|Q\|)}$ Size: $\mathcal{O}(\|D\|) \cdot 2^{\mathcal{O}(\|Q\|)}$	Section 3
(2)	XPath Patterns	Time: $\mathcal{O}(\text{depth}(D) \cdot \log(\text{width}(D))) \cdot 2^{\mathcal{O}(\|Q\|)}$ Size: $\mathcal{O}(\|D\|) \cdot 2^{\mathcal{O}(\|Q\|)}$	Section 3
(3)	$\downarrow, \Downarrow, \wedge, \vee, \neg$	Time: $\mathcal{O}(\text{depth}(D) \cdot \|Q\|)$ Size: $\mathcal{O}(\|D\| \cdot \|Q\|)$	Section 4
(4)	$\rightarrow, \Rightarrow, \wedge$	Time: $\mathcal{O}(\log(\|D\|) \cdot \text{poly}(\|Q\|))$ Size: $\mathcal{O}(\|D\| \cdot \|Q\|^3)$	Section 5
(5)	$\downarrow, \Downarrow, \rightarrow, \Rightarrow, \wedge$	Time $\mathcal{O}(\text{depth}(D) \cdot \log(\text{width}(D)) \cdot \text{poly}(\|Q\|))$ Size: $\mathcal{O}(\|D\| \cdot \|Q\|^3)$	Section 6

1: Overview of results. The size complexities refer to the size of the auxiliary datastructure that has to be maintained.

or syntax. *When the type of a node is* label *then the label must be in* $\Sigma \uplus \{*\}$. *When the type of a node is* syntax*, the label must be one of* $\wedge, \vee, \neg$. *The type of an edge* e*, denoted by type*(e)*, can be* syntax*, or any XPath axis.*

We assume that XPath Patterns are *well-formed*, that is, *(i)* all incoming edges to nodes typed *syntax* must also be typed *syntax*; *(ii)* no other edges are typed *syntax*; *(iii)* a syntax node labeled $\neg$ has only one child. For a set X of axes and boolean operators, we denote by XPath(X) the set of XPath Patterns using only the operators and axes from X.

Throughout the paper, we will use the letter D to denote the XML document, and Q to denote the XPath Pattern. The semantics of XPath Patterns are defined inductively on the structure of the pattern. Given a document D, a node $u \in \text{Nodes}(D)$ and an XPath Pattern Q, we will, for each node i and each edge e in Q, define the two notions $D \models^u Q[i]$ and $D \models^u Q[e]$. Loosely speaking they will express that the subpattern (i.e., subtree) of Q that starts in node i (edge e, resp.) is satisfied in the document D starting at node u.

We say that $D \models^u Q[i]$ iff one of the following conditions holds:

- i is of type *label*, its label coincides with the label of u (where $*$ is considered to coincide with any label) and for all edges $e = (i, j)$ it holds that $D \models^u Q[e]$,
- $\text{label}(i) = \wedge$ and for all edges $e = (i, j)$ it holds that $D \models^u Q[e]$,
- $\text{label}(i) = \vee$ and there exists an edge $e = (i, j)$ such that $D \models^u Q[e]$ or
- $\text{label}(i) = \neg$ and, for the unique edge $e = (i, j)$, we have that $D \not\models^u Q[e]$.

Moreover, for each edge $e = (i, j)$ in Q, $D \models^u Q[e]$ iff

- $\text{type}(e) = \text{syntax}$ or $\text{type}(e) = \text{self}$, and $D \models^u Q[j]$ or
- $\text{type}(e) = \downarrow$ (resp., $\Downarrow$, $\downarrow^*$, $\uparrow$, $\Uparrow$, $\uparrow^*$, $\rightarrow$, $\Rightarrow$, $\leftarrow$, $\Leftarrow$) and there exists a child v of u (resp., descendant, descendant-or-self, parent, ancestor, ancestor-or-self, next sibling, following sibling, previous sibling, preceding sibling v of u) such that $D \models^v Q[j]$.

Finally we say that the document D models the XPath Pattern Q ($D \models Q$) iff $D \models^{\text{root}(D)} Q[\text{root}(Q)]$. We also abbreviate $D \models^u Q[\text{root}(Q)]$ with $D \models^u Q$. If $D \models Q$ we also sometimes write that D *satisfies* Q.

Figure 1(a) illustrates an example of an XPath Pattern that is satisfied in the same set of trees as the XPath query $/a[.//b$ or $(\text{not}(./c))]/*//d$. Figure 1(b) shows a document tree modeling the XPath Pattern from Figure 1(a).

When consideing only XPath Patterns Q without negation or disjunction, $D \models Q$ iff there exists a homomorphic mapping $\phi : \text{Nodes}(Q) \rightarrow \text{Nodes}(D)$ (we say Q *can be matched onto* D) such that Q's root is matched onto D's root and for all edges $e = (i, j) \in Q$ we have

- $\text{type}(i) = \text{syntax}$ or $\text{type}(e) = \text{self}$ implies $\phi(i) = \phi(j)$,
- $\text{type}(e) = \downarrow$ (resp., $\Downarrow$, $\downarrow^*$, $\uparrow$, $\Uparrow$, $\uparrow^*$, $\rightarrow$, $\Rightarrow$, $\leftarrow$, $\Leftarrow$) implies that $\phi(j)$ is a child (resp., descendant, descendant-or-self, parent, ancestor, ancestor-or-self, next sibling, following sibling, previous sibling, preceding sibling) of $\phi(i)$.

We say that ϕ is a *root matching* for Q onto D. When we don't require that ϕ maps $\text{root}(Q)$ onto $\text{root}(D)$, we say that ϕ is a *matching*. By $L(Q)$ we denote the set of documents that model Q.

REMARK 2.2. *XPath and Core XPath also allow the use of the (binary) union operator, but, in the XPath 1.0 specification, this operator can only be used at the* highest *level in the parse tree of the query. That is, in our setting one would be allowed to take the union of several XPath Patterns, but not use the union operator in the patterns themselves. We chose not to add union explicitly as, in our boolean setting, this restricted union operator can simply be simulated by the* $\vee$ *operator without blow-up.*

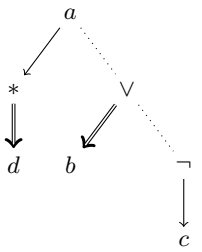
2.2 The Incremental Evaluation Problem

We treat the incremental evaluation problem for XPath Patterns similarly as Balmin, Papakonstantinou, and Vianu treated the incremental validation problem for XML schemas [1].

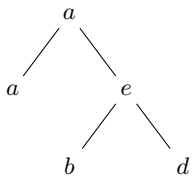
We consider the *Boolean incremental XPath evaluation problem*. That is, given an XPath Pattern Q, a tree D such that $D \models Q$, and an update to D yielding another tree D', we wish to efficiently check if $D' \models Q$. In particular, the cost should be less than evaluating Q on D' from scratch. The individual updates are the following:

(a) replace the current label of a specified node by another label,

(b) insert a new leaf node as the next sibling of a specified node,



(a) XPath Pattern for the query $/a[.//b \text{ or } (\text{not}(./c))]/*//d$.



(b) Document modeling the XPath Pattern of Figure 1(a).

(c) insert a new leaf node as the first child of a specified node, and

(d) delete a specified node; if the node is an internal one, the subtree rooted at the node is also deleted.

It should be noted that in other work it is sometimes also allowed to insert entire subtrees into the document, instead of single nodes. However, as the above updates allow to insert nodes at any position in the tree, this can be accomodated in our framework by inserting the nodes of the subtree one by one.

We allow some cost-free one-time pre-processing, such as computing an automaton representation of a pattern. We will also initialize and then maintain an auxiliary structure $A(D)$ to help in the validation. The cost of the incremental validation algorithm is evaluated w.r.t.:

(a) the time needed to test whether $D' \models Q$ using D and $A(D)$, as a function of $|D|$ and $|Q|$,

(b) the time needed to compute $A(D')$ from D and $A(D)$, as a function of $|D|$ and $|Q|$,

(c) the size of the auxiliary structure $A(D)$ as a function of $|D|$ and $|Q|$.

The complexity results are summarized in Table 1. When we state only one time bound, it holds for both (a) and (b). We assume a RAM-model that can store, e.g., counter values of size $\mathcal{O}(|D| + |Q|)$ in constant space (1 register).

3. FULL XPATH PATTERNS

We start with an approach to incremental evaluation for full XPath Patterns. It builds heavily on well-known techniques for translating XPath into finite-state tree automata (see, e.g., [19, 21]). The following lemma, makes this connection explicit. For an automaton A let $L(A)$ denote the set of trees accepted by A.

LEMMA 3.1. *Let Q be an XPath Pattern. A non-deterministic unranked tree automaton A with $L(A) = L(Q)$ can be constructed in time $2^{\mathcal{O}(|Q|)}$.*

Here, an unranked tree automaton is a tree automaton working directly on unranked trees. Lemma 3.1 was independently discovered (in terms of single-run query automata) by Libkin and Sirangelo [11]. It was already known that standard constructions allowed to construct A in time $2^{\mathcal{O}(\text{poly}(|Q|))}$. The emphasis of Lemma 3.1 is that $2^{\mathcal{O}(|Q|)}$ suffices.

Balmin et al. [1] have shown that given an unranked tree automaton A one can incrementally decide membership of an XML document D in $L(A)$ in time[2] either $\mathcal{O}(\log^2(|D|) \cdot \text{poly}(|A|)$ or $\mathcal{O}(\text{depth}(D) \cdot \log(\text{width}(D)) \cdot \text{poly}(A))$. This immediately implies the following.

THEOREM 3.2. *Boolean incremental evaluation for an XPath Pattern Q and an XML document D can be performed in*

(1) time $\mathcal{O}(\log^2(|D|) \cdot 2^{\mathcal{O}(|Q|)})$ per update; or

(2) time $\mathcal{O}(\mathit{depth}(D) \cdot \log(\mathit{width}(D)) \cdot 2^{\mathcal{O}(|Q|)})$ per update;

both with an auxiliary data structure of size $|D| \cdot 2^{\mathcal{O}(|Q|)}$.

4. DOWNWARD XPATH

As seen in the previous section, an automata-theoretic approach combined with the results from [1] easily yields a maintenance algorithm that is polylogarithmic in the document and exponential in the query size, with auxiliary data which is linear in the document and exponential in the query. This may work as long as the query is very small, but for larger queries, the complexity becomes prohibitive.

One might think that the following approach could work: compute a 2-way alternating tree-automaton for the query, and try to maintain accepting runs of this automaton directly, rather than first translating to a non-deterministic automaton. Unfortunately, each run of the alternating automaton is a tree, and we would have to maintain sets of cuts of such trees, i.e., sets of sets of states. Actually, maintaining acceptance of the alternating automaton more or less amounts to the same as maintaining acceptance for its nondeterministic counterpart. This would again make both running time and auxiliary data exponential in the query.

In the remainder of the paper we present direct maintenance algorithms for two important XPath fragments: downward XPath and conjunctive forward XPath. These algorithms do not involve translations into automata.

In this section, we provide an algorithm for incrementally maintaining a downward XPath pattern, i.e., an XPath($\downarrow$, $\Downarrow, \wedge, \vee, \neg$)-pattern. We show the following result:

THEOREM 4.1. *Boolean incremental evaluation for an XPath($\downarrow, \Downarrow, \wedge, \vee, \neg$) pattern Q and XML document D can be performed in time $\mathcal{O}(\mathit{depth}(D) \cdot |Q|)$ per update. The size of the auxiliary data structure is $\mathcal{O}(|D| \cdot |Q|)$.*

[2]Actually, they state a complexity bound of $\mathcal{O}(\text{depth}(D) \cdot \log(D) \cdot \text{poly}(A))$, but a slightly more detailed analysis shows that the complexity is also $\mathcal{O}(\text{depth}(D) \cdot \log(\text{width}(D)) \cdot \text{poly}(A))$

The algorithm works as follows. For each node u in D, with children $u_1, \ldots, u_k$, we store a record R_u consisting of

- the set of query nodes $S^u = \{q \in \text{Nodes}(Q) \mid D \models^u Q[q]\}$,
- for every query node q, the cardinality $\text{countS}^u(q)$ of the set $\{u_i \mid q \in S^{u_i}\}$,
- the set of query nodes $D^u = \{q \mid \exists u'.\text{descendant}(u, u') \wedge D \models^{u'} Q[q]\}$, and
- for every query node q, the cardinality $\text{countD}^u(q)$ of the set $\{u_i \mid q \in D^{u_i}\}$.

Then, D satisfies Q iff $\text{root}(Q)$ is in $S^{\text{root}(D)}$. So, once the auxiliary data structure is computed, testing whether $D \models Q$ is trivial. The size of each record R_u is $\mathcal{O}(|Q|)$, so the size of the entire auxiliary data structure is $\mathcal{O}(|D| \cdot |Q|)$.

It now suffices to show that we can incrementally update the auxiliary data structure in time $\mathcal{O}(\text{depth}(D) \cdot |Q|)$. Notice that, for each of the updates of a node u in D (label change, node insertion, and node deletion), only the data records R_v for nodes v on the path from u to the root of D change. We recompute these data records in a bottom-up fashion.

Let u be the next node for which the record must be updated, let v be its parent node, and $u_1, \ldots, u_k$ its children. When visiting u we assume that the updated values $\text{countS}^u_{\text{new}}$ and $\text{countD}^u_{\text{new}}$, and $S^{u_i}_{\text{new}}$ and $D^{u_i}_{\text{new}}$, for all $i \in \{1, \ldots, k\}$ are given, and compute S^u_{new}, D^u_{new}, $\text{countS}^v_{\text{new}}$, and $\text{countD}^v_{\text{new}}$. If u is a leaf, we have $\text{countS}^u_{\text{new}}(q) = \text{countD}^u_{\text{new}}(q) = 0$ for all q. The recomputation works as follows:

(1) S^u_{new}: We compute the set S^u_{new} by inspecting Q in a bottom-up fashion.

In Q, we have *syntax* and *label* nodes, and for the latter we distinguish *child* and *descendant* nodes depending on whether the incoming edge is a child or descendant edge. We say that a node q of Q is *satisfied (w.r.t. node u in D)* if (a) q is a syntax node and $q \in S^u_{\text{new}}$, (b) q is a child node and $\text{countS}^u_{\text{new}}(q) > 0$, or (c) q is a descendant node and $\text{countS}^u_{\text{new}}(q) > 0$ or $\text{countD}^u_{\text{new}}(q) > 0$.

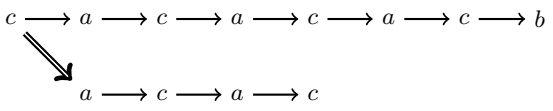
Now, let q be a query node with children $q_1, \ldots, q_\ell$. Then, $q \in S^u_{\text{new}}$ if (a) q is a label node, the label of q matches the label of u, and all children $q_1, \ldots, q_\ell$ are satisfied w.r.t. u; (b) q is a syntax node labeled $\wedge$ and all $q_1, \ldots, q_\ell$ are satisfied w.r.t. u; (c) q is a syntax node labeled $\vee$ and at least one of $q_1, \ldots, q_\ell$ is satisfied w.r.t. u; or (d) q is a syntax node labeled $\neg$ and its (unique) child is not satisfied w.r.t. u.

(2) $\text{countS}^v_{\text{new}}$: For all q,

$$\text{countS}^v_{\text{new}}(q) = \begin{cases} \text{countS}^v(q) + 1 & \text{if } q \in S^u_{\text{new}} \text{ and } q \notin S^u \\ \text{countS}^v(q) - 1 & \text{if } q \notin S^u_{\text{new}} \text{ and } q \in S^u \\ \text{countS}^v(q) & \text{otherwise.} \end{cases}$$

(3) D^u_{new}:

$$D^u_{\text{new}} = \{q \mid \text{countS}^u_{\text{new}}(q) > 0\} \cup \{q \mid \text{countD}^u_{\text{new}}(q) > 0\}$$



2: XPath$(\rightarrow, \Rightarrow, \wedge)$ query.

(4) $\text{countD}^v_{\text{new}}$: For all q,

$$\text{countD}^v_{\text{new}}(q) = \begin{cases} \text{countD}^v(q) + 1 & \text{if } q \in D^u_{\text{new}} \text{ and } q \notin D^u \\ \text{countD}^v(q) - 1 & \text{if } q \notin D^u_{\text{new}} \text{ and } q \in D^u \\ \text{countD}^v(q) & \text{otherwise.} \end{cases}$$

This algorithm requires time $\mathcal{O}(\text{depth}(D) \cdot |Q|)$. Indeed, we only have to update $\text{depth}(D)$ records, each of which can easily be done in time $\mathcal{O}(|Q|)$. Furthermore, as the membership of nodes in the set $\{u \in \text{Nodes}(D) \mid D \models^u Q\}$ only changes for nodes on the path from the update to D's root, it is also easy to output the changes to this set in time $\mathcal{O}(\text{depth}(D) \cdot |Q|)$.

5. XPath$(\rightarrow, \Rightarrow, \wedge)$ ON STRINGS

In the previous section, we presented an algorithm to efficiently maintain downward navigational queries. Our goal in the rest of the paper will be to extend this fragment by adding the next-sibling and following-sibling axes. This will, however, prove to be non-trivial and will come at the cost of dropping negation and disjunction in the queries.

In this section, we present an algorithm for incrementally evaluating XPath$(\rightarrow, \Rightarrow, \wedge)$ on strings. This algorithm will then be used in Section 6 to extend the algorithm of the previous section to also handle the next- and following-sibling axes. More specifically, this section is devoted to proving the following result.

THEOREM 5.1. *Boolean incremental evaluation for an XPath$(\rightarrow, \Rightarrow, \wedge)$ pattern Q and a string D can be performed in time $\mathcal{O}(\log(|D|) \cdot \text{poly}(|Q|))$ per update with an auxiliary data structure of size $\mathcal{O}(|D| \cdot |Q|^3)$.*

This is the most technically difficult result of the paper, and a reader willing to assume the above theorem could skip ahead to Section 6 and return to this point later. Moreover, the present paper does not provide the correctness proof for the algorithm presented in this section.

A slight change in presentation.

In order to simplify the presentation, we will in the present section slightly modify the query that is the input of our problem. The query Q will be adapted to a new query Q' by adding a new root, labeled $*$, which has an outgoing edge to the old root of the query. This edge is a child edge if we are looking for a root matching and a descendant edge if we are looking for any matching. Similarly, every old leaf of the query will have an outgoing descendant edge to a new leaf labeled $*$. So, from now on, we assume that Q is always a pattern with a unique edge leaving the root. The query in Figure 3(a) (without the numbers 1–7), for example, is an adapted version of the query in Figure 2.

5.1 Evaluating an NFA on Strings

First of all, we explain the intuition behind incrementally evaluating a non-deterministic finite automaton (NFA) on

strings, and the challenges that arise when trying to adapt this algorithm for incrementally evaluating XPath$(\rightarrow, \Rightarrow, \wedge)$ on strings. The following technique was first described by Patnaik and Immerman [15] and worked out in more detail by Balmin et al. [1]. Let $N = (\text{States}(N), \text{Alpha}(N), \text{Rules}(N), \text{init}(N), \text{Final}(N))$ be an NFA and assume that we have a string $w = a_1 \cdots a_n$ for which we incrementally want to maintain whether $w \in L(N)$.

We first describe the auxiliary data structure we will maintain to do this efficiently. For each i, j, $1 \leq i < j \leq n$, let T_{ij} be the transition relation $\{(p, q) \mid p, q \in \text{States}(N), p \overset{a_i \cdots a_j}{\rightarrow} q\}$, where $p \overset{a_i \cdots a_j}{\rightarrow} q$ denotes that N can reach state q when it starts in state p and reads $a_i \cdots a_j$. Note that $T_{ij} = T_{ik} \circ T_{(k+1)j}$, $i < k < j$, where $\circ$ denotes composition of binary relations.

For simplicity, assume first that n is a power of 2, say $n = 2^k$. The main idea is to keep as auxiliary information just the T_{ij} for intervals $[i, j]$ obtained by recursively splitting $[1, n]$ into halves, until $i = j$. More precisely, consider the transition relation tree $\mathcal{T}_n$ whose nodes are sets T_{ij}, defined inductively as follows:

- the root is T_{1n};
- each node T_{ij} for which $j - i > 0$ has children T_{ik} and $T_{(k+1)j}$ where $k = i - 1 + \frac{j-i+1}{2}$; and
- the T_{ii} are the leaves, for all $1 \leq i \leq n$.

Note that $\mathcal{T}_n$ has $n + (n/2) + \cdots + 2 + 1 = 2n - 1$ nodes and has depth $\log n$. Thus, the size of the auxiliary structure is $\mathcal{O}(n|\text{States}(N)|^2)$.

First, notice that given $\mathcal{T}_n$ it is easy to decide whether $w \in L(N)$. Indeed, $w \in L(N)$ iff $(q, f) \in T_{1n}$ for some $q \in \text{init}(N)$ and $f \in \text{Final}(N)$. Therefore, we only have to show that this auxiliary data structure can efficiently be updated.

For simplicity, consider the case when one update occurs, changing the label of the symbol at position k of w to b. That is, the new string is $w = a_1 \cdots a_{k-1} b a_{k+1} \cdots a_n$. Note that the relations $T_{ij} \in \mathcal{T}_n$ that are affected by the updates are those lying on the path from the leaf T_{kk} to the root of $\mathcal{T}_n$. Denote the set of these relations by I and notice that it contains at most $\log n$ relations. The tree $\mathcal{T}_n$ can now be updated by recomputing the T_{ij}'s in I bottom-up as follows: First, the leaf relation T_{ii} is set according to $\text{Rules}(N)$ and b. Then each $T_{ij} \in I$ with children T' and T'', of which one has been recomputed, is replaced by $T' \circ T''$. Thus, at most $\log n$ relations have been recomputed, each in time $\mathcal{O}(|\text{States}(N)|^2 \log |\text{States}(N)|)$, yielding a total time of $\mathcal{O}(|\text{States}(N)|^2 \log |\text{States}(N)| \log n)$.

The above approach can easily be adapted to strings whose length is not a power of 2. Further, the auxiliary data structure has size $\mathcal{O}(n \cdot |\text{States}(N)|^2)$. Finally, handling updates in which elements are inserted or deleted is also done in [1], but then some precautions have to be taken in order to make sure that the tree $\mathcal{T}_n$ remains properly balanced.

5.2 From NFAs to XPath$(\rightarrow, \Rightarrow, \wedge)$

We now extend this approach to evaluate XPath patterns on strings. The most straightforward approach would be to translate the XPath query into a finite automaton and apply similar ideas as above. However, to do this translation efficiently, an alternating automaton would be required. Indeed, the branching in the pattern should be handled by universal states, whereas the $\Rightarrow$ axis requires guessing and hence existential states. As already mentioned, incrementally maintaining an alternating automaton seems to require space exponential in the automaton, and is thus not feasible. Therefore, we describe a more direct algorithm below.

In this section, we are only concerned with matching query patterns on strings. Therefore, if a query node u has two children v_1 and v_2, and $\text{type}((u, v_1)) = \text{type}((u, v_2)) = \rightarrow$, then any matching of the pattern on a string must match v_1 and v_2 to the same string position. This means that we might as well merge v_1 and v_2 into a single query node v (if v_1 and v_2 have conflicting labels, we simply conclude that there is no string onto which the pattern can be matched). For this reason we assume in the rest of this section, that no query node has two outgoing edges with type "$\rightarrow$".

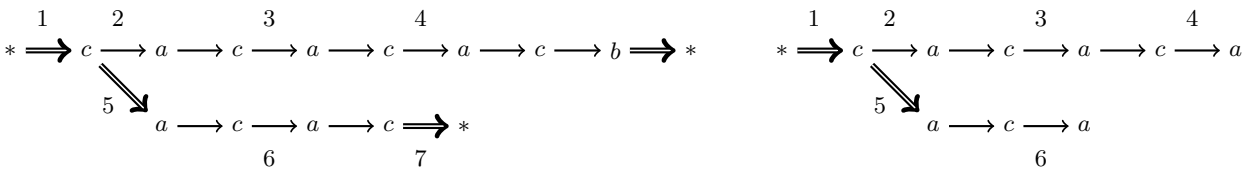
Essentially the incremental algorithm for an NFA remembers, for each relation T_{ij}, a function mapping each state p onto a set P, where $P = \{q \mid (p, q) \in T_{ij}\}$ (but it stores this function directly as a binary relation). We will remember something similar for XPath$(\rightarrow, \Rightarrow, \wedge)$, which we first illustrate by means of an example. For an edge $e = (x, y)$ we refer to x as the *source* and y as the *target* of e.

Example 5.2. Consider the query Q in Figure 3(a) and the string $D = cacac$. Intuitively, D should be seen as a substring of a much larger string, for which we want to compute the information for $T_{i,j}$. Intuitively, we will remember all pairs $(e_1, e_2) \in \text{Edges}(Q) \times \text{Edges}(Q)$ such that the part of the query from the target of e_1 to the source of e_2 can be matched inside D, much like in the NFA case. We talk about edges here because when combining a matching of a part of a query on a part of the string with another matching for a consecutive part of the string, what really interests us is which query *edges* lead from one string part to the next.

For $D = cacac$, we remember the pairs $(1, 2), (1, 3), (1, 4), (1, 5), (1, 6), (1, 7)$. The intuition is that, if we read D from left to right and we start matching in, e.g., edge[3] 1, then pair $(1, 4)$ tells us that we can match until edge 4 at the end of D, *ignoring all paths in Q that branch away* from the path from 1 to 4. So we essentially treat each path in Q as an NFA, where the edges are its states. The difference from the NFA approach is that single pairs do not do not tell us the whole story. For instance, the pair $(1, 4)$ doesn't tell us what happens with the path that branches away, the one from edge 5 to edge 7. Thus we have to combine pairs in order to get matchings that span more than one path in Q. For example, $(1, 4)$ and $(1, 6)$ can be combined to form a partial matching of Q in the following way. Let Q' be obtained from Q by cutting off everything left of 1 and everything right of 4 and 6 (see Figure 3(b)). We can now match Q' into D such that the target of 1 is matched precisely onto the leftmost symbol and the sources of 4 and 6 onto the rightmost symbol, so the matching could continue to the right of D. Notice that we do not care (yet) about the target labels of 4 and 6.

Note, however, that we cannot combine pairs arbitrarily: $(1, 3)$ and $(1, 7)$ cannot be combined into a correct partial matching. When combining matchings for $(1, 3)$ and $(1, 7)$ naively, the descendant requirement (edge 5) will not be satisfied. To solve this problem, note that we have to be careful

[3]For the sake of the argument, the reader can assume that the source node of edge 1 is already matched one position before the first position of D.



(a) The query Q of Example 5.2.

(b) The query Q' of Example 5.2.

about which information to store. A naive generalization of the NFA approach would store pairs (p, P) such that p is a start edge and P is a set of edges which, together, describe a good partial matching (for example, the pair $(1, \{4, 6\})$). But as there can be exponentially many good partial matchings on a string in general,[4] maintaining this information would require exponential space and time. Therefore, we have to adopt a smarter approach.

Towards the Incremental Algorithm.

The rough outline of the algorithm for incrementally evaluating an XPath$(\rightarrow, \Rightarrow, \wedge)$ pattern can now be described. It works quite similar to the algorithm for NFAs described in Section 5.1, with the three crucial differences that

- the relations T_{ij} store different information;
- the algorithm for joining two relations T_{ik} and $T_{(k+1)j}$ into T_{ij} is completely different; and
- the test for acceptance that needs to be performed at $T_{1,n}$ is different.

Here, we describe what information will be stored in T_{ij}. Section 5.3 treats the problem of joining the information. To this end, we need some terminology.

A *path* in a pattern Q is a sequence $\rho = (x_1, y_1) \cdots (x_k, y_k)$ of edges of Q such that $y_i = x_{i+1}$ for all $i \in \{1, \ldots, k-1\}$. If x_1 is the root and y_k is a leaf, we say that ρ is a *maximal path*. A *cut* of a pattern Q is a subset C of Edges(Q) such that every maximal path in Q has exactly one edge in C.

Let C be a cut and $e = (x, y)$ be an edge of Q such that e is not below any edge in C. The *induced subquery of Q w.r.t. e and C*, denoted subQ(Q, e, C), is the pattern obtained from Q by considering only descendants of e, and removing everything below C. More formally, subQ(Q, e, C) is the query Q' where

- Nodes(Q') is $\{x, y\} \uplus \{z \mid \text{descendant}(y, z)$ in Q and $\nexists (u, v) \in C$ such that descendant$(v, z)\}$;
- the edges in Q' are the same as in Q; and
- all edges and nodes in Q' inherit their types from Q.

To simplify notation further on, we use subQ$(Q, \top, C)$ to denote subQ(Q, e, C) where e is the unique edge leaving root(Q), and subQ$(Q, e, \bot)$ to denote subQ(Q, e, C) where C is the cut consisting of all edges entering the leaves of Q.

As we assumed in the beginning of this section that Q contains a new root node, above its old root, we have the following property for all queries and subqueries:

[4]Take a query that has only wildcards, has depth k on each path from root to leaf and the root edge has k outgoing "$\Rightarrow$"-edges.

REMARK 5.3. *In this section, all queries and induced subqueries of Q have a unique edge leaving their root.*

We now define the notion of a partial matching of a subquery into a string D. Our terminology will be slightly more refined — we use *full* and *top* matchings. Intuitively, a full matching ϕ of a query Q will map all the nodes of Q into D, except for its root node. A top matching of Q will be a partial matching that only matches some upper part of Q into D, also excluding the root node.

DEFINITION 5.4 (TOP MATCHING, FULL MATCHING). *Let x be the root node of Q, and $e = (x, y)$ the unique edge leaving x. Let C be a cut of Q and $C_{low} = \{c_1, \ldots, c_n\} = \{v \mid \exists u.(u, v) \in C\}$. Let inner$(Q, C)$ be all nodes of Q that are descendants of x and have a descendant in C_{low}.*

Then ϕ : inner$(Q, C) \rightarrow$ Nodes(D) is a top matching *of Q if the following hold:*

- *ϕ is a matching from inner(Q, C) to D;*
- *if e is a "$\rightarrow$"-edge, then $\phi(y)$ is the first position in D; and*
- *for each $j = 1, \ldots, n$, if there is a u such that $(u, c_j) \in C$ is a "$\rightarrow$"-edge, then $\phi(u)$ is the last position in D.*

We say that C is a witness for ϕ *and, for any $C' \subseteq C$, we also say that ϕ is a* top matching w.r.t. C'. *We say that ϕ is a* full matching of Q on D, *if C_{low} is the set of leaves of Q. Hence, every full matching is also a top matching.*

The Incremental Algorithm.

Recall that, in the incremental evaluation algorithm, T_{ij} denotes the auxiliary data record for the string $a_i \cdots a_j$. Example 5.2 suggests that a naive generalization of the algorithm for NFAs could be to store all pairs (e, C) in T_{ij}, where e is an edge and C a cut, such that there is a full matching of subQ(Q, e, C) on $a_i \cdots a_j$. Unfortunately, in general, there are exponentially many such sets C.

Intuitively, our algorithm stores a ternary relation over edges in each T_{ij}. This relation is a combination of the binary relation shown in Example 5.2, which contains pairs of edges (e_1, e_2) such that there exists a top matching of subQ$(Q, e_1, \bot)$ w.r.t. $\{e_2\}$, and a *co-matchability* constraint, which allows us to infer which such pairs can be combined to form a consistent matching. We formalize this in terms of *matching triples*, for which we first introduce some notation.

DEFINITION 5.5. *An edge $e = (x, y)$ in Q is* direct *if either*

- *e is a $\rightarrow$-edge; or*

- *e is a $\Rightarrow$-edge and all other edges (x, z) are also $\Rightarrow$-edges.*

All other edges in Q are called bridge edges. *A path in Q that consists only of direct edges is a* direct path. *Let $e_1 e_2 \cdots e_k$ be a path in Q. Then the* bridge distance *of e_k from e_1, denoted $\|e_1, e_k\|$ is the number of bridge edges (i.e., non-direct edges) in $\{e_2, \ldots, e_k\}$ (i.e., we count e_k, but not e_1). If $\|e_1, e_k\| = 0$, we also say that e_k is a* direct descendant *of e_1. The set of all bridge edges on the path from e_1 to e_k will be denoted as Bridges(e_1, e_k).*

Notice that each bridge edge is a $\Rightarrow$-edge and that $\|e_1, e_2\|$ is only defined if e_2 is a descendant of e_1 in Q.

DEFINITION 5.6 (MATCHING TRIPLE). *Let e_{top}, e_{bot}, e be query edges of Q such that e is a descendant and e_{bot} a direct descendant of e_{top}. Then (e_{top}, e_{bot}, e) is a* matching triple for D *if there is a top matching w.r.t. $\{e_{bot}, e\}$ for subQ$(Q, e_{top}, \bot)$ on D. We denote the set of matching triples of D by $T(D)$.*

For the incremental update algorithm, we maintain a similar tree $\mathcal{T}_n$ as in Section 5.1, but it is computed differently:

(A) For each position i in the string, compute T_{ii}, i.e., the set of all matching triples for a_i.

(B) For each T_{ij} in the data structure, compute T_{ij} from T_{ik} and $T_{(k+1)j}$, where $k = i - 1 + \frac{j-i+1}{2}$, by adding, for each pair of edges $e_{\text{top}}, e_{\text{bot}}$ in Q such that e_{bot} is a direct descendant of e_{top}, the triples $(e_{\text{top}}, e_{\text{bot}}, e)$ that are computed by Join(Q,T_{ik},$T_{(k+1)j}$,e_{top},e_{bot}). The Join-procedure, which is the main technical difficulty in this paper, is explained in Section 5.3. For the time being, it is only important to know that it runs in polynomial time in Q and that it computes the matching triples for T_{ij} correctly.

At the root $T_{1,n}$ of the data structure we have that $D \models Q$ if and only if there exists a direct descendant e_1 of e_{top} such that for all leaf edges e_2 of Q we have $(e_{\text{top}}, e_1, e_2) \in T_{1,n}$, where e_{top} is the unique root edge of Q (Lemma 5.7 below). Clearly, this can be tested in polynomial time. The size of the auxiliary data structure $\mathcal{T}_n$ is $\mathcal{O}(n \cdot |Q|^3) = \mathcal{O}(|D| \cdot |Q|^3)$.

LEMMA 5.7. *Let Leaf(Q) be the set of edges entering a leaf of Q. Let e_{top} be the root edge of Q, and $e_{bot} \in$ Leaf(Q) be a direct descendant of e_{top}. Then, there is a full matching of Q on D, iff for all $e \in$ Leaf(Q), $(e_{top}, e_{bot}, e) \in T(D)$.*

When a position of D is updated, the incremental update mechanism is exactly the same as in Section 5.1, with the only difference that the updates in $\mathcal{T}_n$ follow the rules (A) and (B) above for recomputing the T_{ij}'s on the path from a leaf to the root. Such an update takes poly(Q) time for (A), and $\mathcal{O}(\log(D) \cdot \text{poly}(Q))$ time for the iteration in (B). Finally, testing if the root condition is fulfilled again takes time polynomial in Q. Addressing node insertions and node deletions can be taken care of by keeping $\mathcal{T}_n$ balanced, which can be done similarly as in the paper by Balmin et al. [1].

5.3 Joining the Data for Two Substrings

In this section we will present the join algorithm (Algorithm 1). Before we can state the algorithm, we first have to give some additional definitions.

DEFINITION 5.8 (BRIDGE WIDTH). *Let r be the unique edge leaving the root of Q. For a query Q, the* bridge width *of Q, denoted $\|Q\|_{bw}$, is the maximal bridge distance in Q, i.e., $\|Q\|_{bw} = \max\{\|e_1, e_2\| \mid e_1, e_2 \in Edges(Q)\}$. The sub-query of Q with* bridge width i, *denoted $bw_i(Q)$, is the query obtained from Q by removing all edges e such that $\|r, e\| > i$, and removing all nodes thus disconnected from r.*

By $D_1 \cdot D_2$ we denote the concatenation of strings D_1 and D_2. Finally, for $C \subseteq$ Edges(Q), let $low(C)$ be the set obtained by removing all edges from C which have a descendant in C.

The core problem for the join algorithm is the following: Given strings D_1 and D_2, the sets of matching triples $T(D_1)$ and $T(D_2)$, the query Q, and edges e_{top} and e_{bot} from Q such that e_{bot} is a direct descendant of e_{top}, compute all triples $(e_{\text{top}}, e_{\text{bot}}, e)$ that belong to $T(D_1 \cdot D_2)$. We can then compute the set of all possible matching triples by iterating over all choices of e_{top} and e_{bot}. A procedure for the core problem is given as Algorithm 1. To get a feel for what the algorithm must do, we consider an example.

EXAMPLE 5.9. Consider the query pattern Q in Figure 4. Each double line denotes a $\Rightarrow$-edge, and a single line denotes a sequence of $\rightarrow$-edges. Notice that we depicted Q sideways, so all edges are directed from left to right. We now assume that we already have the matching triples $T(D_1)$ and $T(D_2)$ for strings D_1 and D_2 and we want to compute $T(D)$, where $D = D_1 \cdot D_2$. For our example, the matching triples for D_1 and D_2 are the ones given in Figure 4(b).

Strictly speaking, $T(D_1)$ and $T(D_2)$ would contain many more matching triples, such as $(e_{\text{top}}, e_{\text{top}}, e_{\text{top}})$, but we limit ourselves to an interesting subset here. We cross out a few triples because we want to explicitly assume that they are *not* matching triples. (Notice that it is possible that (e, c_1, c_3) is a matching triple, while (e, c_1, c_4) is not.) We depict the triples in $T(D_2)$ with dotted lines in Figure 4.

The algorithm takes the two sets of matching triples as input, together with the edges e_{top} and e_{bot}. It will infer every edge e such that $(e_{\text{top}}, e_{\text{bot}}, e)$ is a matching triple. The algorithm iterates over edges with increasing bridge distance from e_{top}. In the figure, the edges on the path from e_{top} to e_{bot} have bridge distance 0, the e-edge, together with the edges on the middle line have bridge distance 1, and all other edges have bridge distance 2.

Initially, on lines (3)–(8) of the algorithm, we join (e_{top}, c', c') with $(c', e_{\text{bot}}, e_{\text{bot}})$ into $(e_{\text{top}}, e_{\text{bot}}, e_{\text{bot}})$, which is our first triple in $T(D)$. During the computation we iteratively construct a set C of edges (x, y) in Q that contains edges for which we matched the source x in D_1 and the target y in D_2, and which forms a cut through Q. Essentially, C should be a "greedy cut", i.e., we remember the edges that are as low in Q as possible. Therefore, on line 5, we set $C = \{c'\}$ which forms a cut through the part of Q only containing the edges at distance 0.

We then proceed to the loop that begins on line 12. In the first iteration, when $j = 0$, we should consider all edges e with distance 0 from e_{top}. However, in our example, all such edges lie on the path from e_{top} to e_{bot}. We can never have a matching triple $(e_{\text{top}}, e_{\text{bot}}, e)$ where $e \neq e_{\text{bot}}$ lies on the same path as e_{bot}, since only one of them can be matched at the end of D. Thus we already covered this case by adding $(e_{\text{top}}, e_{\text{bot}}, e_{\text{bot}})$ to $T(D)$.

Therefore, we proceed to the second iteration of the while

Algorithm 1 Join algorithm. The algorithm takes a query Q, two sets of matching triples $T(D_1)$ and $T(D_2)$ and two edges $e_{\text{top}}, e_{\text{bot}}$ as input. It is assumed that e_{bot} is a direct descendant of e_{top}. The algorithm computes all matching triples $(e_{\text{top}}, e_{\text{bot}}, e)$ of $D = D_1 \cdot D_2$, where e is a descendant of e_{top}.

Join(Query Q, Triples $T(D_1)$, Triples $T(D_2)$, Edge e_{top}, Edge e_{bot})
2: $P \leftarrow \text{subQ}(Q, e_{\text{top}}, \bot)$
if $\exists c' : (e_{\text{top}}, c', c') \in T(D_1) \wedge (c', e_{\text{bot}}, e_{\text{bot}}) \in T(D_2)$ **then**
4: $c' \leftarrow$ the lowermost such edge
$C \leftarrow \{c'\}$
6: $T_0(D) \leftarrow \{(e_{\text{top}}, e_{\text{bot}}, e_{\text{bot}})\}$
else
8: **return** $\emptyset$
end if
10: $j \leftarrow 0$
$T_1(D) \leftarrow \cdots \leftarrow T_{\|P\|_{\text{bw}}}(D) \leftarrow \emptyset$
12: **while** $j \leq \|P\|_{\text{bw}}$ **do**
$C' \leftarrow \emptyset$
14: **for all** $e \in \text{Edges}(P)$ s.t. $\|e_{\text{top}}, e\| = j$ **do**
if $(c', e_{\text{bot}}, e) \in T(D_2)$ **then**
16: $T_j(D) \leftarrow T_j(D) \cup \{(e_{\text{top}}, e_{\text{bot}}, e)\}$
else if $\exists e' \in C \setminus \{c'\}\ \exists e'' : (e', e'', e) \in T(D_2)$ **then**
18: $T_j(D) \leftarrow T_j(D) \cup \{(e_{\text{top}}, e_{\text{bot}}, e)\}$
else if $\exists e' : (e_{\text{top}}, c', e') \in T(D_1) \wedge (e', e, e) \in T(D_2) \wedge \forall e'' \in \text{Bridges}(e_{\text{top}}, e') : \|e_{\text{top}}, e''\| < j$
20: $\rightarrow \exists c'' \in C : (e'', c'', e') \in T(D_1)$ **then**
$e' \leftarrow$ the lowermost such edge
22: $C' \leftarrow C' \cup \{e'\}$
$T_j(D) \leftarrow T_j(D) \cup \{(e_{\text{top}}, e_{\text{bot}}, e)\}$
24: **end if**
end for
26: $C \leftarrow C \cup low(C')$
$j \leftarrow j + 1$
28: **end while**
return $\cup_{j=0}^{\|P\|_{\text{bw}}} T_j(D)$

loop in which all edges at distance 1 are considered. Here, the if-statement on lines 19-20 applies, and we can combine

- $(e_{\text{top}}, c', c_1) \in T(D_1)$ with $(c_1, e_1, e_1) \in T(D_2)$ into $(e_{\text{top}}, e_{\text{bot}}, e_1)$.

By doing so, we add c_1 to C' on line 22. This is the set of candidate edges to be added to the cut C. As at the end of the while loop, we still have $C' = \{c_1\}$, we obtain $C = \{c', c_1\}$ after the second iteration. Notice that $\{c', c_1\}$ indeed forms a cut of $\text{bw}_1(Q)$, the part of Q only containing the edges with distance at most 1.

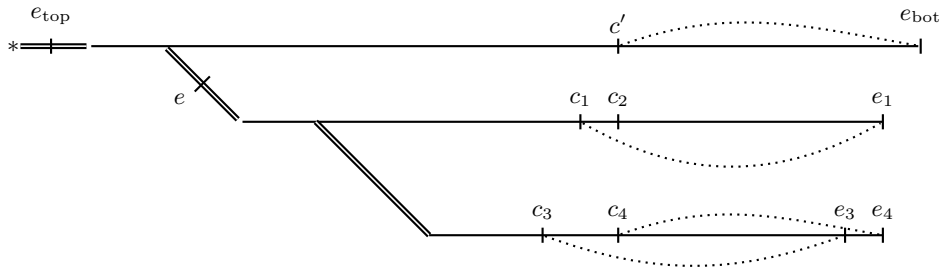
The third iteration of the while loop is the first one that becomes interesting, as now we have to consider a combination of three paths in the query, while our triples only store information about pairs of paths. First, we discuss how naive joins may go wrong. One may be tempted to conclude from $(e_{\text{top}}, c', c_4) \in T(D_1)$, $(c', e_{\text{bot}}, e_{\text{bot}}) \in T(D_2)$, and $(c_4, e_4, e_4) \in T(D_2)$ that $(e_{\text{top}}, e_{\text{bot}}, e_4)$ is a matching triple for D. However, it is not! And, indeed, there is no rule in the algorithm which would add $(e_{\text{top}}, e_{\text{bot}}, e_4)$ to the set of matching triples. Essentially, the reasons are that *(i)* we could match up to c_2 in D_1 (since $(e_{\text{top}}, c', c_2) \in T(D_1)$), but we could not continue this matching in D_2 (since $(c_2, e_1, e_1) \notin T(D_2)$) and *(ii)* $(e, c_1, c_4) \notin T(D_1)$. In other words, we cannot achieve a matching that is consistent with the lowest possible cut $C = \{c', c_1\}$ that we have computed thus far. This is the point where we need to make use of C to make the correct combinations, and decide which matching can be combined and which cannot. Therefore, by applying lines 19-20 we combine, among others,

- $(e_{\text{top}}, c', c_3)$ and (e, c_1, c_3) from $T(D_1)$ with (c_3, e_3, e_3) from $T(D_2)$ into $(e_{\text{top}}, e_{\text{bot}}, e_3)$.

After iteration three, we will, for this query, have computed all the matching triples and a cut $C = \{c', c_1, c_3\}$. Should the query be larger, C would be used to witness further matchings. This concludes Example 5.9.

We now present some more general ideas behind Algorithm 1. We already explained in the example that the algorithm investigates triples $(e_{\text{top}}, e_{\text{bot}}, e)$ in order of increasing bridge distance between e_{top} and e, and that C is a greedy cut containing edges (x, y) for which x is matched in D_1 and y in D_2. We assume that the algorithm has computed all the triples with $\|e_{\text{top}}, e\| < j$ and that C is the lowermost set of edges with bridge distance smaller than j such that there is a top matching of $P = \text{subQ}(Q, e_{\text{top}}, \bot)$ with respect to C on D_1. Now, consider what the algorithm does for an edge e with $\|e_{\text{top}}, e\| = j$. This edge is submitted to three tests: the if-statements on lines 15, 17, and 19-20. The situations the tests look for are depicted in Figure 5. Each double line denotes a $\Rightarrow$-edge, and a single line denotes a sequence of $\rightarrow$-edges. All edges are directed from left to right.

The first test, on line 15, looks for the situation depicted in Figure 5(a), i.e., when the path from e_{top} to e branches of from the path from e_{top} to e_{bot} after the edge c'. If



(a) Abstract query.

$T(D_1)$					$T(D_2)$		
(e_{top}, c', c')	$(e_{\text{top}}, c', c_1)$	$(e_{\text{top}}, c', c_2)$	$(e_{\text{top}}, c', c_3)$	$(e_{\text{top}}, c', c_4)$	$(c', e_{\text{bot}}, e_{\text{bot}})$	(c_1, e_1, e_1)	~~(c_2, e_1, e_1)~~
(e, c_1, c_3)	(e, c_2, c_3)	(e, c_2, c_4)	~~(e, e_1, c_4)~~	...	(c_3, e_3, e_3)	(c_4, e_4, e_4)	...

(b) Matching triples for D_1 and D_2.

4: Abstract query and matching triples for Example 5.9.

this is the case, the algorithm only has to check whether $(c', e_{\text{bot}}, e) \in T(D_2)$.

The second test, on line 17, looks for the situation in Figure 5(b). Here, there is an edge e', different from c', that lies on the path from e_{top} to e and already belongs to C (which implies $\|e_{\text{top}}, e'\| < j$). The algorithm now only needs to test whether, for some e'' the triple (e', e'', e) belongs to $T(D_2)$.

The third test, on lines 19-20, is the most complicated. It looks for the situation in Figure 5(c). Here, no edge on the path from e_{top} to e yet belongs to C. This means that the algorithm has to verify that the edge e' that is on the path from e_{top} to e and goes from D_1 into D_2 in the intended matching, is also consistent with the cut C constructed thus far.

By applying Algorithm 1 for all possible edges e_{bot} and e_{top} we obtain the following.

LEMMA 5.10. *Given the query Q and the sets $T(D_1)$ and $T(D_2)$ of matching triples for Q on strings D_1 and D_2, the set $T(D)$ of matching triples for Q on $D = D_1 \cdot D_2$ can be computed in time polynomial in $|Q|$.*

PROOF. We first analyze the running time of Algorithm 1. Its first part (lines 3-9) takes time linear in $|Q|$. (We can assume constant time lookup in the sets $T(D_1)$ and $T(D_2)$, which can be implemented, e.g., as tree-dimensional matrices.) In the second part (lines 10-28) all edges that are descendants of e_{top}, in order of increasing bridge distance, are considered. The for-loop that starts on line 14 makes one iteration per such edge. Within the loop, three cases are distinguished by the if-statements on lines 15, 17 and 19-20. The most complicated of these is the third, lines 19-20, which looks at cubically many triples, and thus runs in cubic time. All in all, Algorithm 1 runs in time $\mathcal{O}(|Q|^4)$.

To compute all matching triples in $T(D)$ we have to call the join algorithm for all pairs $(e_{\text{top}}, e_{\text{bot}})$, i.e., a quadratic number of times, so the total running time is $\mathcal{O}(|Q|^6)$. □

The auxiliary data structure needed for incremental evaluation over a string D has a number of tree nodes that is linear in $|D|$. Each tree node contains a set of matching triples, and thus needs space $\mathcal{O}(|Q|^3)$. In total, the size of the auxiliary data structure is $\mathcal{O}(|D| \cdot |Q|^3)$. Together with Lemma 5.10, this gives us Theorem 5.1.

6. CONJUNCTIVE FORWARD XPATH

After the preparatory work in the previous section, we are now ready to extend the algorithm of Section 4 to also handle the next-sibling ($\rightarrow$) and following-sibling ($\Rightarrow$) axes. However, in order to do this we disallow disjunction ($\vee$) and negation ($\neg$) in the pattern, leaving us with the fragment XPath($\downarrow, \Downarrow, \rightarrow, \Rightarrow, \wedge$), which we refer to as conjunctive forward XPath. All such queries can be thought of as tree pattern queries with only label nodes, so we don't need to consider syntax nodes. Every branching in the pattern implicitly denotes a conjunction. We will show the following.

THEOREM 6.1. *Boolean incremental evaluation for an XPath($\downarrow, \Downarrow, \rightarrow, \Rightarrow, \wedge$) pattern Q and an XML document D runs in time $\mathcal{O}(\mathit{depth}(D) \cdot \log(\mathit{width}(D)) \cdot \mathit{poly}(|Q|))$ per update, with an auxiliary data structure of size $\mathcal{O}(|D| \cdot |Q|^3)$.*

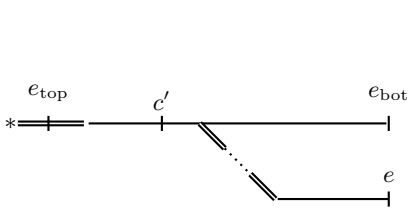
For a node q of Q, let the *subpattern without siblings of* q, denoted subtreeNoSibling(q), be the subtree of Q rooted at q from which all outgoing sibling edges (and corresponding subtrees) leaving q are removed. Here, by *sibling edge*, we mean both "$\rightarrow$"- and "$\Rightarrow$"-edges. Notice that we only removed sibling edges that are directly attached to q, so subtreeNoSibling(q) can still contain sibling edges deeper in the pattern.

Further, for a node q of Q, the *subpattern with only siblings of* q, denoted subtreeOnlySibling(q), is the subtree of Q rooted at q and containing all nodes reachable from q by following only sibling edges. Notice that subtreeOnlySibling(q) only contains sibling edges, and is thus a query in the fragment XPath($\rightarrow, \Rightarrow, \wedge$) treated in the previous section.

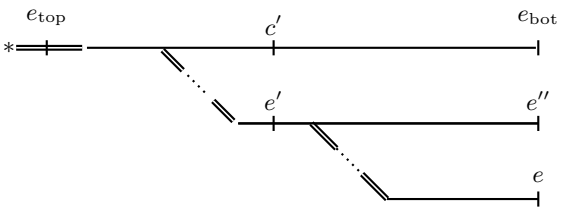
Let downNodes(Q) be the subset of Nodes(Q) such that for each $q \in$ downNodes(Q), the unique incoming edge to q in Q has type $\downarrow$ or $\Downarrow$.

The algorithm works as follows. For each node u in D we store a record R_u consisting of:

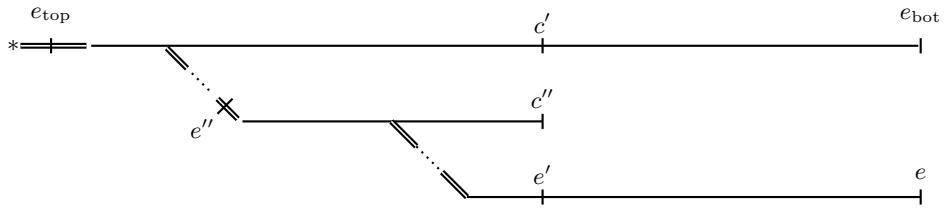
- the set of query nodes $S^u = \{q \in \text{Nodes}(Q) \mid D \models^u \text{subtreeNoSibling}(q)\}$,
- the set of query nodes $C^u = \{q \in \text{downNodes}(Q) \mid \exists u'.\text{child}(u, u') \wedge D \models^{u'} Q[q]\}$,
- the set of query nodes



(a) Algorithm 1, Line 15.



(b) Algorithm 1, Line17.



(c) Algorithm 1, Lines 19-20.

5: Illustrations of the situations identified by the if-statements in the for-loop of Algorithm 1.

$D^u = \{q \in \text{Nodes}(Q) \mid \exists u'.\text{descendant}(u,u') \wedge D \models^{u'} Q[q]\}$,

- for each query node q the integer $\text{countD}^u(q) = |\{u' \mid \text{child}(u,u') \wedge \exists u''\ \text{descendant}(u',u'') \wedge D \models^{u''} Q[q]\}|$.

Without loss of generality, we assume that the root node of Q does not have outgoing sibling edges. Indeed, if it has, it can never be mapped to the root of D. Therefore, it suffices to check whether $S^{\text{root}(D)}$ contains root(Q) to decide whether $D \models Q$.

However, to maintain the above records, we still need to store some additional information. For each node u of D and each query node $q \in \text{downNodes}(Q)$ we also store the data structures needed to incrementally maintain the membership of a string w in subtreeOnlySibling(q). Here, w is a string formed by relabeled versions of the children of u in D, and the data structures are these maintained by the algorithm in Section 5. The concrete details about the string w are given below.

We now show how the records R_u can be updated. As in Section 4, it suffices to recompute this information for all nodes on the path of the updated node to the root. Let u be the next node to be updated, v be its parent, and $u_1, \ldots, u_k$ its children. We use S^u_{new}, C^u_{new}, etc. to refer to the record values after the update. We assume that C^u_{new}, D^u_{new}, and $\text{countD}^u_{\text{new}}(q)$ are given and show how to compute S^u_{new}, C^v_{new}, D^v_{new}, and $\text{countD}^v_{\text{new}}(q)$. If u is a leaf node, we have $C^u = D^u = \emptyset$ and $\text{countD}^u = 0$, for all query nodes q.

- S^u_{new}: For a child q' of q, we say that q' is a $\downarrow$-child (resp., $\Downarrow$-child), if $\text{type}((q,q')) = \downarrow$ (resp., $\Downarrow$). A $\downarrow$-child q' is *satisfied* if $q' \in C^u_{\text{new}}$, and a $\Downarrow$-child q' if $q' \in C^u_{\text{new}} \cup D^u_{\text{new}}$. Then, $q \in S^u_{\text{new}}$ if q's label matches the label of u and all its $\downarrow$- and $\Downarrow$-children are satisfied.

- C^v_{new}: To know whether $D \models^{u'} Q[q]$ for some child u' of v, we have to consider all query nodes which are reachable from q by following edges typed with sibling axes, i.e., all nodes in subtreeOnlySibling(q). Indeed, $q \in C^v$ should hold if these reachable query nodes can be matched to children of v in such a manner that the matching is (1) consistent with the sibling edges of the query and (2) every query node q' that is reachable from q by sibling edges is matched to such a child node u' such that $D_{u'} \models \text{subtreeNoSibling}(q')$, where $D_{u'}$ is the subtree of D rooted at u', i.e., $q' \in S^{u'}$.

  The existence of such a matching can efficiently be decided (and maintained) as follows. First, consider the string $w = \overline{v_1} \cdots \overline{u} \cdots \overline{v_n}$, corresponding to the sequence $v_1 \cdots u \cdots v_n$ of children of v where the $\overline{v_i} = S^{v_i}$ (and $\overline{u} = S^u_{\text{new}}$), i.e., the label is formed by the set of query nodes whose subpattern without siblings can be matched here. Second, consider the query $Q_{\text{sib}} = \text{subtreeOnlySibling}(q)$. Then, we say that a query node q' of Q_{sib} *matches* a string symbol $\overline{v_i} = S^{v_i}$ iff $q' \in S^{v_i}$. Now, $q \in C^v$ iff there exists a matching of Q_{sib} on w. This matching does not need to be a root matching, but can be any matching. Furthermore, notice that at most one label, namely the one for u, in w changes when an update occurs. Therefore, we can use the algorithm presented in Section 5 to incrementally maintain tree pattern queries over strings, in this slightly altered semantics, to efficiently decide whether $q \in C^v_{\text{new}}$.

- $\text{countD}^v_{\text{new}}$: For all q, $\text{countD}^v_{\text{new}}(q)$
$$= \begin{cases} \text{countD}^v(q) + 1 & \text{if } q \in D^u_{\text{new}} \text{ and } q \notin D^u_{\text{old}} \\ \text{countD}^v(q) - 1 & \text{if } q \notin D^u_{\text{new}} \text{ and } q \in D^u_{\text{old}} \\ \text{countD}^v(q) & \text{otherwise} \end{cases}$$

- D^v_{new} : $D^v_{\text{new}} = C^v_{\text{new}} \cup \{q \mid \text{countD}^v_{\text{new}}(q) > 0\}$.

We argue that the algorithm works in time $\mathcal{O}(\text{depth}(D) \cdot \log(\text{width}(D)) \cdot \text{poly}(|Q|))$. We have to update at most depth(D) nodes, so it suffices to argue that handling one node can be done in time $\mathcal{O}(\log(\text{width}(D)) \cdot \text{poly}(|Q|))$. All sets except C^v can easily be updated in time $\mathcal{O}(|Q|)$. Further, for updating C^v, we have to apply the incremental maintenance algorithm for strings, which has complexity $\mathcal{O}(\log(w) \cdot \text{poly}(|P|))$, where $w \leq \text{width}(D)$ (see Theorem 5.1) and P is the query to be maintained. This algorithm has to be run for all query nodes $q \in \text{downNodes}(Q)$ and corresponding queries subtreeOnlySibling(Q). These subpatterns are all disjoint,

and therefore the sum of their sizes is at most $|Q|$. Furthermore, the slightly altered semantics of the matching relation (that a query node matches a document node if the label of the query node is in the set defined by the document node) also does not increase the complexity of the algorithm in Section 5 beyond $\mathcal{O}(\log(w) \cdot \text{poly}(|P|))$. This all together means that the update of C^v can be done in time $\mathcal{O}(\text{width}(D) \cdot \text{poly}(|Q|))$.

Finally, we show that the data structure can be stored in space $\mathcal{O}(|D| \cdot |Q|^3)$. As in Section 4, the node records can be stored in space $\mathcal{O}(|D| \cdot |Q|)$. However, this is dominated by the space needed to store the auxiliary data structures for the incremental maintenance algorithm for strings. According to Theorem 5.1 these data structures can be stored in space $\mathcal{O}(|w| \cdot |P|^3)$, where w is the string and P the query to be evaluated. Then, as all subpatterns we use for incremental string maintenance are disjoint, and every document node has only one parent, it follows that all information can be stored in space $\mathcal{O}(|D| \cdot |Q|^3)$.

7. CONCLUSIONS AND FURTHER DIRECTIONS

We have shown that incremental evaluation of XPath queries can be performed significantly more efficiently than re-evaluation, for several practically interesting fragments of XPath.

Of course, our study is far from complete and this work should be seen as an initial theoretical step in this line of work. We hope that we were able to show that, incremental evaluation for some seemingly very innocent fragments of XPath (essentially the tree pattern fragment) is already quite non-trivial, even if the XML data is structured as a string instead of a tree (Section 5). These are the most important directions we want to investigate in the future:

- Extending the algorithm/complexity result from Section 5 to trees.
- More expressive fragments of XPath, for an algorithm that is more efficient than the one in Section 3 (e.g., w.r.t. negation and 2-way navigation).
- Investigate adding data values to XPath fragments.
- Strengthen the view maintenance approach. Ideally, we would like to be able to maintain a set of designated *output nodes* in XPath Patterns, that produce a relation as output of the query.
- Investigate whether the approach in Section 3 can be strengthened by using Simon's theorem, which has already proved to be useful for XPath evaluation (see [5]).

Acknowledgments

We are grateful to the anonymous reviewers, whose valuable suggestions helped improve the presentation of this paper.

8. REFERENCES

[1] A. Balmin, Y. Papakonstantinou, and V. Vianu. Incremental validation of XML documents. *ACM TODS*, 29(4):710–751, 2004.

[2] Z. Bar-Yossef, M. Fontoura, and V. Josifovski. On the memory requirements of XPath evaluation over XML streams. *JCSS*, 73(3):391–441, 2007.

[3] D. Barbosa, A.O. Mendelzon, L. Libkin, L. Mignet, and M. Arenas. Efficient incremental validation of XML documents. In *ICDE*, pages 671–682, 2004.

[4] M. Benedikt and C. Koch. XPath leashed. *ACM Comp. Surveys*, 2008. To appear.

[5] M. Bojańczyk and P. Parys. XPath evaluation in linear time. In *PODS*, pages 241–250, 2008.

[6] G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. *ACM TODS*, 30(2):444–491, 2005.

[7] G. Gottlob, C. Koch, R. Pichler, and L. Segoufin. The complexity of XPath query evaluation and XML typing. *J. ACM*, 52(2):284–335, 2005.

[8] M. Götz, C. Koch, and W. Martens. Efficient algorithms for the tree homeomorphism problem. In *DBPL*, pages 17–31, 2007.

[9] M. Grohe, C. Koch, and N. Schweikardt. Tight lower bounds for query processing on streaming and external memory data. *TCS*, 380(1–2):199–217, 2007.

[10] A. Gupta, I. Singh Mumick, and V.S. Subrahmanian. Maintaining views incrementally. In *SIGMOD*, pages 157–166, 1993.

[11] Leonid Libkin and Cristina Sirangelo. Reasoning about XML with temporal logics and automata. In *LPAR*, pages 97–112, 2008.

[12] H. Matsumura and K. Tajima. Incremental evaluation of a monotone XPath fragment. In *CIKM*, pages 245–246, 2005.

[13] P.E. O'Neil, E.J. O'Neil, S. Pal, I. Cseri, G. Schaller, and N. Westbury. ORDPATHs: Insert-friendly XML node labels. In *SIGMOD*, pages 903–908, 2004.

[14] M. Onizuka, F. Yee Chan, R. Michigami, and T. Honishi. Incremental maintenance for materialized XPath/XSLT views. In *WWW*, pages 671–681, 2005.

[15] S. Patnaik and N. Immerman. Dyn-FO: A parallel, dynamic complexity class. *JCSS*, 55(2):199–209, 1997.

[16] A. Sawires, J. Tatemura, O. Po, D. Agrawal, A. El Abbadi, and K. Selçuk Candan. Maintaining XPath views in loosely coupled systems. In *VLDB*, pages 583–594, 2006.

[17] A. Sawires, J. Tatemura, O. Po, D. Agrawal, and K. Selçuk Candan. Incremental maintenance of path-expression views. In *SIGMOD*, pages 443–454, 2005.

[18] N. Schweikardt. Machine models and lower bounds for query processing. In *PODS*, pages 41–52, 2007.

[19] T. Schwentick. XPath query containment. *Sigmod RECORD*, 33(1):101–109, 2004.

[20] O. Shmueli and A. Itai. Incremental view maintenance. In *SIGMOD*, pages 240–255, 1984.

[21] B. ten Cate and C. Lutz. The complexity of query containment in expressive fragments of XPath 2.0. In *PODS*, pages 73–82, 2007.