XPath Whole Query Optimization

Sebastian Maneth
NICTA and UNSW
Sydney, Australia
sebastian.maneth@nicta.com.au

Kim Nguyen
NICTA
Sydney, Australia
kim.nguyen@nicta.com.au

ABSTRACT

Previous work reports about SXSI, a fast XPath engine which executes tree automata over compressed XML indexes. Here, reasons are investigated why SXSI is so fast. It is shown that tree automata can be used as a general framework for fine grained XML query optimization. We define the "relevant nodes" of a query as those nodes that a minimal automaton must touch in order to answer the query. This notion allows to skip many subtrees during execution, and, with the help of particular tree indexes, even allows to skip internal nodes of the tree. We efficiently approximate runs over relevant nodes by means of on-the-fly removal of alternation and non-determinism of (alternating) tree automata. We also introduce many implementation techniques which allows us to efficiently evaluate tree automata, even in the absence of special indexes. Through extensive experiments, we demonstrate the impact of the different optimization techniques.

1. INTRODUCTION

The XPath query language plays a central role in XML processing: it is deeply uprooted in almost every XML technology, starting from query languages such as XQuery and XSLT, to access control languages such as XACML, to JavaScript engine of popular web browsers. Thus, efficient XPath evaluation is essential for any time-critical XML processing. In this paper we show how tree automata can be used as framework for fine-grained and novel types of XPath query optimizations. The experiments with our prototype show that, together with appropriate indexes for the XML document tree, these optimizations give rise to unprecedented execution speed for XPath queries, outperforming the fastest existing XPath engines.

The first breakthrough in efficient XPath execution was Koch *et al.*'s seminal paper [6] (see also [7]) where it is shown that Core XPath can be evaluated in time $O(|D| \cdot |Q|)$ where $|D|$ is the size of the document and $|Q|$ is the size of the query. Core XPath refers to the tree navigational fragment of XPath. Considering the time bound of Koch's algorithm, there are two obvious ways of reducing this complexity in practice:

(1) reduce the number of query steps ("$|Q|$-optimization") and

(2) reduce the number of nodes to consider ("$|D|$-optimization").

Proceedings of the VLDB Endowment, Vol. 3, No. 1

Extreme $|Q|$-Optimization: A top-down deterministic tree automaton (TDTA) processes an input tree starting in its initial state, at the root node. It then applies a unique rule which says, for a given state and label of a node, how to process the children of that node. A node is selected as a result, if the unique state reached by the automaton on that node and the label of that node are elements of a special "set of selection pairs". After compiling a (restricted) XPath query into such an automaton (which takes $O(|Q|)$ time), the run function only requires a single look-up at each node of the input tree (plus possibly an insertion of the current node into the result list. Since the function visits the nodes in document order and only once, this insertion can be performed in constant time, keeps the result sorted and duplicate-free). Thus, the evaluation runs in $O(|D|)$ time, giving the extreme case of $|Q|$-optimization to $|Q| = 1$. Similar automata for XML processing have been considered [12–14]. However, implementations of such automata cannot compete with state-of-the-art XPath engines.The reasons for this deficiency are that (1) performance depends on the speed of firstChild and nextSibling operations in the XML tree data structure, (2) the automaton needs to visit *every* node of D and (3) the compilation into TDTA only works for a very restricted subset of Core XPath.

To address (1), many implementations use in-memory pointer structures. However, this blows up the memory requirement by a factor of 5-10 over the size of the original XML document. Hence, such implementations can only work over small documents. We solve this problem by using state-of-the-art succinct trees [18], a recent development in data structures.

Solutions to problems (2) and (3) are the main subject of this paper. We study ways to restrict the nodes of the document which must be visited by the run function of the automaton. This gives rise to the notion of *relevant nodes*, one of our key contributions. To address (3), we work with non-deterministic alternating tree automata and carefully develop on-the-fly determinization and alternation elimination algorithms. This allows to retain most the benefits of deterministic automata while increasing the expressive power to full Core XPath. Altogether, our implementation of these solutions to (1) – (3) provides XPath execution speed competitive with the best known engines [1]. While we restrict ourselves for didactic reasons to a fragment of Core XPath, our prototype "SXSI" implements Core XPath plus text predicates [1]; we are currently adding other XPath 1.0 features such as number functions and aggregates.

$|D|$-Optimization using Relevant Nodes Consider the query Q_0 = //a//b which selects all b-descendants of a-labeled nodes. A TDTA for this query starts at the root in a state q_0. When it encounters an a-node it changes to a state q_1. Any b-node encountered in q_1 is selected as result. For such an automaton we say that a node is *relevant*, whenever the automaton changes state, or selects a node. Thus, all top-most a-nodes and all their b-labeled descendants are relevant. Note that for this query, one could use the staircase join [9] to restrict the set of all a-nodes to the top-most ones, and only then select b-descendants; in this way only the relevant b-nodes are touched (but some non-relevant a-nodes might be touched in the first step). Here, we first give an algorithm that executes an arbitrary TDTA so that *only relevant nodes are visited*. This is achieved by executing the automaton over an index that allows at any node to "jump" to the next σ-labeled descendant (for any label σ) or to the next σ-labeled following node (according to XPath), for any σ. For bottom-up deterministic tree automata (BDTA), we can define relevant nodes in a similar way. We sketch an algorithm for BDTAs that *only touches relevant nodes*, given an index that allows access to all bottom-most nodes with a given label and allows to jump to labeled ancestors (due to space constraint and the fact that the bottom-up algorithm has to handle more cases than the top-down one to ensure that nodes are only visited once, we do not give it fully in this paper).

Given a query, it is not always possible to determine which one of the bottom-up or top-down evaluation is the most efficient (*i.e.* visits fewer nodes). For instance, for query Q_0, if the input document has less b nodes than a nodes, a bottom-up traversal seems more efficient. Following this idea, we extend our evaluation algorithm to support *Start Anywhere Runs*: for a query such as //a//b//c, if the global count of b-nodes is low, we can jump to these b-nodes, and from there execute simultaneously a bottom-up run which checks for a-nodes and a top-down run which selects c-nodes.

Non-Deterministic Automata To determine the relevant nodes for a TDTA or BDTA, we actually first have to *minimize* the automaton. Intuitively, a non-minimal automaton can do many useless state-changes. While minimization can efficiently be done for deterministic automata, it poses a big problem for non-deterministic automata. Here, minimization is EXPTIME-complete, and, there need not even exist a unique minimal automaton. Unfortunately, for XPath we *must* deal with non-deterministic automata: consider Q_1 = //a[.//b]//c. If we execute it top-down and are below an a-node, then for a c-node we cannot know whether to select it (this depends on the presence of b-nodes which might be below). Similarly, the query //a//c cannot be done in a deterministic bottom-up way. There is an elegant way to characterize relevant nodes for non-deterministic automata, using equivalence between sub-automata.

This notion proves too complex to implement in practice (equivalence is EXPTIME-complete), but we give an *on-the-fly* algorithm which soundly *approximates* the relevant nodes of a nondeterministic tree automaton, while evaluating the automaton on an input tree. Our experiments show that for typical XPath queries our on-the-fly algorithms perform well: the approximation of the set of relevant nodes that we compute is close to the real set allowing us to only visit a small fraction of the complete document.

Plan Section 2 gives the definitions and introduces our model of selecting tree automata. Section 3 formally defines the concept of relevant nodes and studies two optimal algorithms for minimal top-down and bottom-up selecting tree automata. Section 4 introduces our variant of alternating tree automata, their encoding of XPath queries, and presents the approximating algorithm as well as a collection of implementation techniques. The impact of these techniques is validated by experiments given in Section 5. Some non-crucial aspects are detailed in the Appendix.

Related Work

Skipping of complete subtrees has been considered before, in several different contexts. For instance, the application of the staircase join [9] can be seen as an instance of skipping: for the descendant axis, only the top-most independent context nodes are considered, i.e., their subtrees are skipped; in a similar way, even ancestor paths can be skipped by this join. Skipping of subtrees is also common practice in advanced compilers for pattern matching in programming languages. In [11] selecting tree automata are compiled into mutually recursive functions of an ML-style target language. They define "loop breaker" states, intuitively, a state with transition $q, l \rightarrow (q, q)$. This is similar to non-relevant nodes, according to our definition, and is used there to enforce the termination of the generated code There is a large body of work on optimizations for evaluation of attribute grammars (see, e.g., [15]) some of which correspond to skipping of subtrees; note that attribute grammars can simulate selecting TDTA and BDTAs. In [5] automata are used for tree pattern matching and subtrees are skipped according to type information. Tree automata have been used for XPath, but mainly in the context of streaming: Koch [10] runs BDTAs over a reversed XML document followed by a top-down run, to evaluate XPath. Suciu *et al.* [8] use automata to evaluate many queries in parallel, over a stream. We are not aware of any work that executes automata over tree indexes, such as we do. In fact, even for usual DFAs over strings, there is no prior work on executing DFAs or evaluating regular expressions over indexed strings (where the index allows to skip regions of the string, based on labels); the closest work is [2]. Also comparable is the idea of running DFAs on grammar-compressed strings. The THOR system [16, 17], uses data structures that support the same jumping operations as we do. However, they do step-wise evaluation of XPath a la Koch and therefore cannot use these structures to restrict evaluation to only relevant nodes.

It should be noted that the presented work is an in-depth presentation of the automata-based technique used in [1], where the interested reader can find comprehensive experiments (against both MonetDB and Qizx/DB) and a description of the use of automata with custom indexes.

2. SELECTING TREE AUTOMATA

We define our notion of tree automata over binary trees. When applying them to XML we use the well-known "first-child/next-sibling" encoding: the first-child of a node in the XML tree becomes the left child in the binary tree, and the next-sibling of a node in the XML tree becomes the right child in the binary tree. We also do not consider text nodes or attributes (but a straightforward encoding is given in [1]). Let Σ be an alphabet, i.e., a finite set of symbols. The *set of binary trees over* Σ, denoted $T(\Sigma)$, is the smallest set T such that (i) the leaf symbol $\#$ is in T and (ii) if $t_1, t_2 \in T$ and $l \in \Sigma$, then $l(t_1, t_2)$ is in T. In the examples, we will often omit $\#$ for concision. A *node* is a finite (possibly empty) sequence over $\{1, 2\}$. For a given tree $t \in T(\Sigma)$ its *set of nodes*, denoted $\mathcal{D}om(t)$, is the smallest finite set such that (i) the empty sequence ε is in $\mathcal{D}om(t)$ and (ii) if two sequences $\pi \cdot 1$ and $\pi \cdot 2$ are in $\mathcal{D}om(t)$, then $\pi \in \mathcal{D}om(t)$. The label of the node π in the tree t is denoted by $t(\pi)$; for $t = l(t_1, t_2)$ it is defined as l if $\pi = \varepsilon$, and as $t_i(\pi')$ if $\pi = i \cdot \pi'$; moreover, for $t = \#$ we have $t(\varepsilon) = \#$. As we can see, ε denotes the root node, and $\pi \cdot 1$ and $\pi \cdot 2$ denote the left and right-child of the node π, respectively. When talking about the *followings* of a node π, we mean all the nodes visited after π during a pre-order traversal, that are not descendants of π.

Definition 2.1 A *selecting tree automaton* (STA) $\mathcal{A}$ is a 6-tuple $(\Sigma, Q, \mathcal{T}, \mathcal{B}, \mathcal{S}, \delta)$ where Σ is an alphabet of *input symbols*, Q is a finite set of *states*, $\mathcal{T} \subseteq Q$ is the set of *top states*, $\mathcal{B} \subseteq Q$ is the set of *bottom states*, $\mathcal{S} \subseteq Q \times \Sigma$ is the set of *selecting configurations*, and δ is a finite set of *transitions*. A transition is tuple (q, L, q_1, q_2), where $q, q_1, q_2 \in Q$ and L is a non-empty subset of Σ.

From now on we let $\mathcal{A} = (\Sigma, Q, \mathcal{T}, \mathcal{B}, \mathcal{S}, \delta)$ be a fixed (but arbitrary) automaton, unless otherwise specified. We often write $q, L \rightarrow (q_1, q_2)$ to denote that $(q, L, q_1, q_2) \in \delta$, and similarly $q, L \Rightarrow (q_1, q_2)$ to denote that $(q, L, q_1, q_2) \in \delta$ and $(q, l) \in \mathcal{S}$ for every $l \in L$. Before defining the semantics of $\mathcal{A}$ via runs, we fix a few useful definitions. Let $q, q_1, q_2 \in Q$ and $l \in \Sigma$. The *destination* and *source states*, denoted $\delta(q, l)$ and $\delta(q_1, q_2, l)$, respectively, are defined as

$$\begin{aligned}\delta(q,l) &= \{(q',q'') \mid \exists L \subseteq \Sigma \text{ s.t. } l \in L \text{ and } (q,L,q',q'') \in \delta\}\\ \delta(q_1,q_2,l) &= \{q \mid \exists L \subseteq \Sigma \text{ s.t. } l \in L \text{ and } (q,L,q_1,q_2) \in \delta\}.\end{aligned}$$

An automaton $\mathcal{A}$ is a *top-down deterministic selecting tree automaton* (TDSTA) if $\mathcal{T}$ is a singleton and, for every $q \in Q$ and $l \in \Sigma$, $\delta(q, l)$ is a singleton. Similarly, $\mathcal{A}$ is a *bottom-up deterministic selecting tree automaton* (BDSTA) if $\mathcal{B}$ is a singleton and, for every $q_1, q_2 \in Q$ and $l \in \Sigma$, $\delta(q_1, q_2, l)$ is a singleton. Note that if $\mathcal{S}$ is empty, then a TDSTA is exactly the same as a classical deterministic top-down tree automaton (TDTA): the single state in $\mathcal{T}$ is the initial state and the states in $\mathcal{B}$ are the final states; similarly, a BDSTA is a classical deterministic bottom-up tree automaton (BDTA): the single state in $\mathcal{B}$ is its initial state and the states in $\mathcal{T}$ are its final states. The semantics of an STA is given by the set of trees it recognizes (as for usual tree automata) and by the set of nodes it selects. To formalize these notions, we introduce the concept of run.

Definition 2.2 (Run of an STA) Let $t \in T(\Sigma)$. A *run* of $\mathcal{A}$ over t is a total function $R : \mathcal{D}om(t) \rightarrow Q$ such that for all $\pi \in \mathcal{D}om(t)$ with $t(\pi) \in \Sigma$,

$$R(\pi) \in \delta(R(\pi \cdot 1), R(\pi \cdot 2), t(\pi)).$$

The run R is *accepting* if and only if

- $R(\varepsilon) \in \mathcal{T}$
- for all $\pi \in \mathcal{D}om(t)$ with $t(\pi) = \#$, $R(\pi) \in \mathcal{B}$.

We denote by $R^t_{\mathcal{A}}$ the set of all accepting runs of $\mathcal{A}$ over t.

An STA is *top-down complete*, if for every $q \in Q$ and $l \in \Sigma$, $\delta(q, l)$ is non-empty. Similarly, an STA is *bottom-up complete*, if for every $q_1, q_2 \in Q$ and $l \in \Sigma$, $\delta(q_1, q_2, l)$ is non-empty. Top-down complete TDSTAs $\mathcal{A}$ and bottom-up complete BDSTAs have a unique run for any input tree t.

Definition 2.3 Let $\mathcal{A}$ be an STA. The *language of* $\mathcal{A}$, denoted $\mathcal{L}(\mathcal{A})$, is the set

$$\mathcal{L}(\mathcal{A}) = \{t \in T(\Sigma) \mid R^t_{\mathcal{A}} \neq \varnothing\}.$$

The *set of selected nodes of* $\mathcal{A}$, denoted $\mathcal{A}(t)$, is the set

$$\mathcal{A}(t) = \{\pi \in \mathcal{D}om(t) \mid (R(\pi), t(\pi)) \in \mathcal{S} \text{ and } R \in R^t_{\mathcal{A}}\}.$$

We say that two STAs $\mathcal{A}$ and $\mathcal{A}'$ are *equivalent*, denoted $\mathcal{A} \equiv \mathcal{A}'$, if $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$ and for every $t \in T(\Sigma)$, $\mathcal{A}(t) = \mathcal{A}'(t)$.

Example 2.1 (STA for //a//b)

$$\mathcal{A}_{//a//b} = (\underbrace{\{\mathtt{a},\mathtt{b},\mathtt{c}\}}_{\Sigma}, \underbrace{\{q_0,q_1\}}_{Q}, \underbrace{\{q_0\}}_{\mathcal{T}}, \underbrace{\{q_0,q_1\}}_{\mathcal{B}}, \underbrace{\{(q_1,\mathtt{b})\}}_{\mathcal{S}}, \delta)$$

$$\delta \;=\; \begin{array}{ll} q_0, \{\mathtt{a}\} \rightarrow (q_1, q_0) & q_1, \{\mathtt{b}\} \Rightarrow (q_1, q_1) \\ q_0, \Sigma \setminus \{\mathtt{a}\} \rightarrow (q_0, q_0) & q_1, \Sigma \setminus \{\mathtt{b}\} \rightarrow (q_1, q_1) \end{array}$$

The TDSTA $\mathcal{A}_{//a//b}$ of Example 2.1 is *not* deterministic bottom-up. This is because its set $\mathcal{B}$ of bottom states is not a singleton. In fact, we claim that there does not exist any BDSTA that is equivalent to $\mathcal{A}_{//a//b}$, i.e., which selects the same nodes. Intuitively, when a bottom-up automaton sees a b-node, it does not know whether this node should be accepted or not (this depends on the existence of an a-labeled ancestor). We claim similarly that there exists BDSTAs for which there is no equivalent TDSTA. The automaton implementing the query //a[.//b] is such an example (which we detail in Appendix A). To conclude with the formal definitions, we characterize several kinds of states that we use in the following sections.

Definition 2.4 Let $\mathcal{A}$ be an STA. A state $q \in Q$ is *non-changing* if and only if $\forall l \in \Sigma, \delta(q, l) = \{(q, q)\}$. For a non-changing state q, if $q \in \mathcal{B}$, q is a *top-down universal state*; if $q \in \mathcal{T}$, q is a *bottom-up universal state*; if $q \notin \mathcal{B}$, q is a *top-down sink state*; if $q \notin \mathcal{T}$, q is a *bottom-up sink state*.

Minimal Selecting Tree Automata In Appendix A.2 it is shown that for every TDSTA (resp. BDSTA) there is a *unique minimal* one, where minimal means with the smallest number of states. For a minimal TDSTA $\mathcal{A}$: (i) at most one state is top-down universal state and (ii) at most one state is a top-down sink state. If any of these states exist, then we denote them by $q_\top$ and $q_\bot$, respectively. The similar properties hold for BDTAs. Another property that will be important for us in the next section is that, in a minimal TDSTA or BDSTA, if a state q is *not* in $\{q_\top, q_\bot\}$, then there must exist a label l such that $\delta(q, l)$ contains a pair different from (q, q). We say that l is an *essential label for* q (in $\mathcal{A}$).

3. RELEVANT NODES

As we have explained in the Introduction, our goal is to improve query answering time by reducing the number of nodes that have to be visited by the evaluation function. A common optimization technique for tree automata (especially used in *pattern-matching* and type-checking), is to avoid visiting a subtree. For instance, consider the simple DTD "<!ELEMENT a ANY>" which states that an input document must have an a-labeled root node and any well-formed content below it. A recognizer automaton which checks the validity of a tree against this DTD is

$$\mathcal{A} = (\Sigma, \underbrace{\{q_0, q_\top, q_\bot\}}_{Q}, \underbrace{\{q_0\}}_{\mathcal{T}}, \underbrace{\{q_\top\}}_{\mathcal{B}}, \underbrace{\varnothing}_{\mathcal{S}}, \delta)$$

$$\delta \;=\; \begin{array}{ll} q_0, \{\mathtt{a}\} \rightarrow (q_\top, q_\top) & q_\top, \Sigma \rightarrow (q_\top, q_\top) \\ q_0, \Sigma \setminus \{\mathtt{a}\} \rightarrow (q_\bot, q_\bot) & q_\bot, \Sigma \rightarrow (q_\bot, q_\bot) \end{array}$$

Since the automaton only changes state at the root node, only this node is "relevant"; no information is gained at any other node. A clever evaluator may skip all non-relevant subtrees. As we can see, whenever the automaton enters a *non-changing state*, we can skip the current subtree. Of course, there are automata equivalent to the one above which change state in the subtrees under the root node (even though this is not "required"). How can we make sure that our automaton only changes state when this is really necessary? The answer is simple: we *minimize* the automaton. If the minimal automaton changes state, then any other automaton for the query does too; thus it uniquely determines the relevant nodes. Moreover, as mentioned after Definition 2.4, the minimal automaton has at most one state $q_\bot$ and one state $q_\top$. It is therefore easy to determine when a subtree can be skipped. Of course, in a selecting tree automaton, all selected nodes must be relevant, because we cannot select them without visiting them. Consequently, given a TDSTA $\mathcal{A}$ and a tree t we say that node π of t is relevant if the minimal automaton $\mathcal{A}_{min}$ of $\mathcal{A}$ changes state at π. We now give a general definition that can be used for *non-deterministic* automata; instead of minimality, the definition uses equivalence between sub-automata.

Definition 3.1 (Relevant nodes) Let $\mathcal{A}$ be an STA. Let $t \in T(\Sigma)$ and $R \in R^t_{\mathcal{A}}$. Let $\pi \in \mathcal{D}om(t)$ such that $\pi \cdot 1 \in \mathcal{D}om(t)$ and $\pi \cdot 2 \in \mathcal{D}om(t)$. The node π is *relevant* for the run R if and only if either $(R(\pi), t(\pi)) \in \mathcal{S}$ or *none* of the following hold:

- $\mathcal{A}[R(\pi)] \equiv \mathcal{A}[R(\pi \cdot 1)] \equiv \mathcal{A}[R(\pi \cdot 2)]$;
- $\mathcal{A}[R(\pi)] \equiv \mathcal{A}[R(\pi \cdot 1)]$ and $\mathcal{A}[R(\pi \cdot 2)] \equiv \mathcal{A}_\top$;
- $\mathcal{A}[R(\pi)] \equiv \mathcal{A}[R(\pi \cdot 2)]$ and $\mathcal{A}[R(\pi \cdot 1)] \equiv \mathcal{A}_\top$;

where $\mathcal{A}_\top$ is such that $\mathcal{L}(\mathcal{A}_\top) = T(\Sigma)$ and for all $t \in T(\Sigma)$, $\mathcal{A}_\top(t) = \varnothing$. $\mathcal{A}[q]$ denotes the restriction of $\mathcal{A}$ to q (*i.e.* where $\mathcal{T}$ is replaced by $\{q\}$) and is formally defined in Appendix A.

This definition generalizes the intuition we gave earlier. First, a selected node is relevant. Then, a node can be skipped (*i.e.* is *not* relevant) if the automaton performs the same computation on the node and on both its children (informally the automaton "loops" both on the left and right child). Or a node can be skipped if the automaton loops on the left child and "ignores" the right child, *i.e.* is in a state that accepts $T(\Sigma)$ and does not mark any node. Symmetrically, a node can be skipped if the automaton loops on the right child and ignores the left one. While Definition 3.1 gives a proper semantic characterization of relevant nodes, we cannot use it to derive an efficient evaluation procedure for STAs since:

(i) it requires the accepting run to be known, while we want to deduce relevant nodes *while* computing the run;

(ii) it checks for equivalence of sub-STAs, an EXPTIME-complete problem, even for recognizers.

We present two exact algorithms for particular STAs, namely minimal TDSTAs and minimal BDSTAs, and show how a particular index can be used to skip not only subtrees but also internal nodes.

3.1 Deterministic Top-Down Evaluation

3.1.1 Top-down Relevance

As we have explained, testing the relevance of a node in the accepting run of an automaton $\mathcal{A}$ consists in checking the equivalence of several sub-automata. It is possible to perform this check efficiently for *minimal* TDSTAs. Indeed, in a minimal TDSTA, q recognizes $T(\Sigma)$ if and only if q is a top-down universal state. More generally, given two states q and q' of $\mathcal{A}$:

$$\mathcal{A}[q] \not\equiv \mathcal{A}[q'] \iff q \neq q'.$$

This is a consequence of the definition of a minimal automaton. Given a TDSTA and a run, we can easily characterize the set of relevant nodes:

Lemma 3.1 (Top-down relevant nodes) *Let $\mathcal{A}$ be a minimal top-down complete TDSTA, $t \in T(\Sigma)$, $R \in R^t_{\mathcal{A}}$ and $\pi \in \mathcal{D}om(t)$ such that $\pi \cdot 1 \in \mathcal{D}om(t)$ and $\pi \cdot 2 \in \mathcal{D}om(t)$. π is* top-down relevant *in R if and only if either $(R(\pi), t(\pi)) \in \mathcal{S}$ or if none of the following hold:*

- $R(\pi) = R(\pi \cdot 1) = R(\pi \cdot 2)$
- $R(\pi) = R(\pi \cdot 1)$ *and* $R(\pi \cdot 2) = q_\top$
- $R(\pi) = R(\pi \cdot 2)$ *and* $R(\pi \cdot 1) = q_\top$

For a given run of a minimal TDSTA, the relevant nodes are either the selected nodes or nodes for which a state-change occurs. An important observation is that for TDSTAs, a state change is exactly determined by the set of *essential* labels. For instance, in the automaton $\mathcal{A}_{//\mathsf{a}//\mathsf{b}}$ of Example 2.1, the set of essential labels for state q_0 is $\{\mathsf{a}\}$: the automaton changes state only if it encounters an a-labeled node during the top-down run.

3.1.2 Top-Down Jumping Functions

Based on this observation, we define particular jumping functions in a tree which extend the basic firstChild and nextSibling moves. The implementation of such functions using state of the art tree indexes is later discussed in Section 5.

Definition 3.2 (Top-down jumping functions) Let t be a tree in $T(\Sigma)$. We define the functions $\mathbf{d}_t, \mathbf{f}_t, \mathbf{l}_t, \mathbf{r}_t$ as:

- $\mathbf{d}_t : \mathcal{D}om(t) \times 2^\Sigma \to \mathcal{D}om(t) \cup \{\Omega\}$ where $\mathbf{d}_t(\pi, L)$ returns the first descendant π' of π (in document-order) such that $t(\pi') \in L$;
- $\mathbf{f}_t : \mathcal{D}om(t) \times 2^\Sigma \times \mathcal{D}om(t) \to \mathcal{D}om(t) \cup \{\Omega\}$ and $\mathbf{f}_t(\pi, L, \pi_0)$ returns the first following node π' of π such that $\pi' \in L$ and π' is a descendant of π_0.
- $\mathbf{l}_t : \mathcal{D}om(t) \times 2^\Sigma \to \mathcal{D}om(t) \cup \{\Omega\}$ where $\mathbf{l}_t(\pi, L)$ returns the first descendant π' of π whose label is in L and such that $\pi' = \pi \cdot 1 \ldots \cdot 1$ (left-most path);
- $\mathbf{r}_t : \mathcal{D}om(t) \times 2^\Sigma \to \mathcal{D}om(t) \cup \{\Omega\}$ where $\mathbf{r}_t(\pi, L)$ returns the first descendant π' of π whose label is in L and such that $\pi' = \pi \cdot 2 \ldots \cdot 2$ (right-most path).

All these function returns a special error node Ω if there is no $\pi' \in \mathcal{D}om(t)$ which fits their definitions.

Using these functions, the set of top-most nodes $\pi_0, \ldots, \pi_n$ whose labels are in L, in a subtree rooted at π can be computed by:

$\pi_0 = \mathbf{d}_t(\pi, L)$ and then $\pi_{n+1} = \mathbf{f}_t(\pi_n, L, \pi)$, until $\pi_n = \Omega$.

3.1.3 Jumping Top-Down Algorithm

We use the jumping functions defined in the previous section to compute a partial run for a minimal TDSTA and an input tree t. More specifically, the algorithm returns a mapping from nodes to states. If there is no accepting run, the algorithm aborts and returns an empty mapping. We describe informally the algorithm (its pseudo code is given in Appendix B.1). The algorithm is implemented by the mean of a recursive function *topdown_jump* which takes as argument a node π in the input tree t and a state q (initially the root node ε and the initial state q_0 of the TDSTA). This function works like the usual top-down evaluation procedure for a TDSTA. First, if π is a leaf (a $\#$-labeled node in our context) then the automaton checks whether $q \in \mathcal{B}$. If this is the case, the function returns the mapping $\{\pi \mapsto q\}$ and fails otherwise. More interestingly if π is not a leaf, then function computes the states $(q_1, q_2) = \delta(q, t(\pi))$. If either q_1 or q_2 is the sink state, then the function fails (there is no accepting run). Otherwise, the function performs a case analysis on q_i to determine the set of top-most relevant nodes in the subtree rooted at $\pi \cdot i$ (for $i \in \{1, 2\}$). The function considers the three cases given in Lemma 3.1:

- $q_i, L' \to (q_i, q_i)$ and $q_i, L \to (q', q'')$ with q' or q'' distinct from q_i. The function performs its recursion on all the top-most descendants of $\pi \cdot i$ whose label is in L;
- $q_i, L' \to (q_i, q_\top)$ and $q_i, L \to (q', q'')$ with q' distinct from q_i. The function is called recursively on the node $\mathbf{l}_t(\pi \cdot i, L)$ (the automaton loops on the left-most path below the current node).
- $q_i, L' \to (q_\top, q_i)$ and $q_i, L \to (q', q'')$ and q'' distinct from q_i. The function is called recursively on the node $\mathbf{r}_t(\pi \cdot i, L)$

If none of the above hold, $\pi \cdot i$ is relevant and the function is recursively called on $\pi \cdot i$ itself. Lastly, the function returns the mapping $\{\pi \mapsto q\}$ augmented by the mappings returned by the recursive calls on the left and right subtrees. This function computes the optimal traversal with respect to relevant nodes:

Theorem 3.1 *Let $t \in T(\Sigma)$. Let $\mathcal{A}$ be a minimal TDSTA. Let R be the run of $\mathcal{A}$ over t and $R' =$ topdown_jump$(t, \mathcal{A})$.*

- *if R is an accepting run, then for all $\pi \in \mathcal{D}om(t)$, $R'(\pi) = R(\pi)$ if an only if π is top-down relevant for R;*
- *if R is not an accepting run, then $R' = \varnothing$.*

3.2 Deterministic Bottom-Up Evaluation

While a top-down run of an automaton can be translated into a natural top-down tree traversal, bottom-up runs are more complicated. Assuming that a parent move and access to the sequence of leaves of an input tree are supported, we can devise a "pure bottom-up" evaluation function, which starts from the sequence of leaves and works its way up to the root. The pseudo code of this algorithm is given in Appendix B.2. From the sequence $(\pi_1, q_0), \ldots, (\pi_n, q_0)$

of all leaves π_i and initial state q_0 the algorithm proceeds to "reduce" them (by replacing two siblings by their parent and corresponding state) until the root node is obtained. If the first two nodes in the current list are not siblings, the algorithm first reduces recursively the tail of the list, pushes back the first element on the reduced tail (whose size decreased) and reduces the new list. For BDSTA, relevance is once again defined in terms of state change, but in a more complex way.

Lemma 3.2 (Bottom-up relevant nodes) *Let $\mathcal{A}$ be a complete minimal bottom-up BDSTA. Let $\mathcal{B} = \{q_0\}$. Let t be a tree. Let R be the accepting run for $\mathcal{A}$ and t (if it exists). Let $\pi \in \mathcal{D}om(t)$ such that $\pi \cdot 1 \in \mathcal{D}om(t)$ and $\pi \cdot 2 \in \mathcal{D}om(t)$. The node π is* relevant *if and only if $(R(\pi), t(\pi)) \in \mathcal{S}$ or none the following conditions holds:*

- $R(\pi) = q_\top$
- $R(\pi) = R(\pi \cdot 1) = R(\pi \cdot 2)$*;*
- $R(\pi) = R(\pi \cdot 1)$ *and* $R(\pi \cdot 2) \in \{q_0, q_\top\}$*;*
- $R(\pi) = R(\pi \cdot 2)$ *and* $R(\pi \cdot 1) \in \{q_0, q_\top\}$*;*

We do not give the proof that these conditions on states coincide with the relevance of nodes as given by Definition 3.1, but illustrate them by an example given in Appendix B.2.

In the same way we generalized firstChild to $\mathbf{d}_t$ and $\mathbf{l}_t$ and nextSibling to $\mathbf{f}_t$ and $\mathbf{r}_t$ for the top-down case, the moves used in the bottom-up algorithm can be generalized. The sequence of all leaves is replaced by the sequence of bottom-most nodes with a particular label and the parent move can be replaced by either a jump to an ancestor with a particular label, or the restriction of this jump to the left-most or right-most path leading to the current node. Also, testing whether two nodes are siblings in generalized into getting the common ancestor of two nodes. We dub the generalized bottom-up jumping algorithm *bottomup_jump*, but the many cases it handles (intuitively, when trying to jump above two nodes π_1 and π_2 we must not jump above their common ancestor, or we could miss some nodes) makes its presentation verbose even in the form of pseudo-code. Second, the tree indexes that we use in our implementation do not implement the *ancestor* jumps efficiently (they amount to a sequence of parent calls). We therefore limit ourselves to state the existence of algorithm *bottomup_jump*, and give its theoretical properties:

Theorem 3.2 *Let $t \in T(\Sigma)$. Let $\mathcal{A}$ be a minimal BDSTA. Let R be the run of $\mathcal{A}$ over t and $R' = bottomup\_jump(t, \mathcal{A})$. (1) If R is an accepting run, then for all $\pi \in \mathcal{D}om(t)$. $R'(\pi) = R(\pi)$ if an only if π is bottom-up relevant for R; (2) If R is not an accepting run, then $R' = \varnothing$.*

4. AUTOMATA FOR XPATH

We present in this section our compilation target for XPath expressions, namely alternating selecting tree automata (ASTA). We then consider a particular fragment of XPath for which we illustrate our compilation scheme. Afterwards we introduce a technique for evaluating an ASTA in a jumping fashion, using a sound approximation of the sets of relevant nodes of the query. We also present various implementation techniques to further improve the complexity in practice of the evaluation of ASTAs.

4.1 Alternating Selecting Tree Automata

We introduce a compact variation of STAs which works with Boolean formulas over states.

Definition 4.1 (Alternating Selecting Tree Automata (ASTA)) An ASTA $\mathcal{A}$ is a tuple $(\Sigma, \mathcal{Q}, \mathcal{T}, \delta)$, where Σ is the alphabet of input symbols, Q is the finite set of states, $\mathcal{T} \subseteq Q$ is the set top states, and δ is a set of tuples (q, L, τ, ϕ), called transitions, where $q \in Q$, $L \subseteq \Sigma$, $\tau \in \{\rightarrow, \Rightarrow\}$ and ϕ is a *Boolean formula* generated by the following EBNF.

$$\phi \quad ::= \quad \top \mid \bot \mid \phi \vee \phi \mid \phi \wedge \phi \mid \neg\phi \mid \ \downarrow_1 q \mid \ \downarrow_2 q \quad (q \in Q)$$

The semantics of such automata combine the rules for a classical alternating automaton, with the rules of a selecting tree automaton. The complete rules for the evaluation of formula and the selection of nodes is given in Appendix C.

4.2 From XPath to Automata

The fragment of XPath we consider in this presentation is the forward fragment of Core XPath, containing `descendant` and `child` axes as well as arbitrarily nested predicates using `or`, `and` and `not` Boolean connective over path expressions. The full EBNF description of this fragment is given in Appendix C. We illustrate how to compile an XPath expression of this fragment into an ASTA.

Example 4.1 (ASTA for the query `//a//b[c]`) Let

$$\mathcal{A}_{\text{//a//b[c]}} = (\Sigma, \{q_0, q_1, q_2\}, \{q_0\}, \delta)$$

where δ is:

$q_0, \{a\} \rightarrow \downarrow_1 q_1$	$q_1, \{b\} \Rightarrow \downarrow_1 q_2$	$q_2, \{c\} \rightarrow \top$
$q_0, \Sigma \ \rightarrow \downarrow_1 q_0 \vee \downarrow_2 q_0$	$q_1, \Sigma \ \rightarrow \downarrow_1 q_1 \vee \downarrow_2 q_1$	$q_2, \Sigma \ \rightarrow \downarrow_2 q_2$

It is easy to see with this example that such automata can be built by a simple traversal of the parse tree of the XPath query. The compilation scheme we follow associates one state for each step of the query, and each state has at most two transitions. The first one represents a "progress" from the current step to the next step (in the XPath query). The second transition represents a recursion on the first child, the second child or both. Note that non-determinism is used here in an essential way. For instance, in $\mathcal{A}_{\text{//a//b[c]}}$, in state q_1, if the current node is labelled `b`, then the automaton selects a node if its first child is in state q_2 and at the same time remains in state q_1 for both the first child and the second child.

While this automaton does not seem to justify the use of alternation, we give in Appendix C a query whose corresponding ASTA is linear in size but whose STA (even non-deterministic) is exponentially larger.

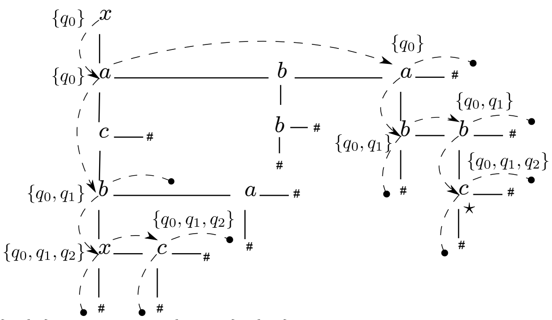
On this example, we observe that the particular ASTAs we consider share many common traits with the minimal deterministic TDSTAs of Section 3.1. First a state change occurs whenever the automaton gains new knowledge toward answering the query. Second, a top-down universal state correspond to the presence of $\top$ in a formula (that is, $(q_\top, q_\top)$) or the absence of a $\downarrow_1$ or $\downarrow_2$ move (for instance $\downarrow_2 q$ is the counterpart of $(q_\top, q)$ in our previous model). In such automata, a state change has the same meaning as in a minimal deterministic one.

4.3 Bottom-Up Evaluation with Top-Down Pre-Processing and Jumping

Before discussing how to evaluate such automata using only relevant nodes, we give a "non-jumping" run function for ASTAs.

Algorithm 4.1 (Evaluation of an ASTA)
Input: $\mathcal{A} = (\Sigma, Q, \mathcal{T}, \delta), t, \pi, r$ **Output:** Γ
where $\mathcal{A}$ is the automaton, t the input tree, r a set of states and Γ is a result set. Initially $\pi = \varepsilon$ and $r = \mathcal{T}$.

```
function eval_asta (A, t, π, r) =
  if t(π) = # then return ∅ else
  let trans = {(q, L, τ, φ) ∈ δ | q ∈ r and t(π) ∈ L} in
  let r_i = {q |↓_i q ∈ φ, ∀φ ∈ trans} in
  let Γ1 = eval_asta (A, t, π · 1, r1)
  and Γ2 = eval_asta (A, t, π · 2, r2)
  in return eval_trans(Γ1, Γ2, π, trans)
```



$$
\begin{array}{ll}
\{q_0\}, \{a\} & \rightarrow \{q_0, q_1\}, \{q_0\} \\
\{q_0\}, \Sigma \setminus \{a\} & \rightarrow \{q_0\}, \{q_0\} \\
\{q_0, q_1\}, \{b\} & \rightarrow \{q_0, q_1, q_2\}, \{q_0, q_1\} \\
\{q_0, q_1\}, \Sigma \setminus \{b\} & \rightarrow \{q_0, q_1\}, \{q_0, q_1\} \\
\{q_0, q_1, q_2\}, \{b\} & \rightarrow \{q_0, q_1, q_2\}, \{q_0, q_1, q_2\} \\
\{q_0, q_1, q_2\}, \{c\} & \rightarrow \{q_0, q_1\}, \{q_0, q_1\} \\
\{q_0, q_1, q_2\}, \Sigma \setminus \{b\} & \rightarrow \{q_0, q_1\}, \{q_0, q_1, q_2\}
\end{array}
$$

Figure 1: Top-down approximation for `//a//b[c]` and corresponding jumps

The function *eval_asta* evaluates an ASTA over an input tree t. It returns a result set Γ which is a mapping from states to the sets nodes selected in that state. In the usual non-selecting, algorithm, Γ is simply the set of states which accept the current node π.

We have already described in details how node selection works for such automata in [1], we focus on the main novelty of this work, relevant node approximation. The interested reader can refer to Appendix C for the complete semantics of ASTA (including node selection) as well as a commented example. This process is abstracted by the function *eval_trans* on Line 7 which handles both selection and evaluation of formulas.

The parameter r of the function *eval_asta* allows one to restrict bottom-up runs of $\mathcal{A}$ to only those which end-up in a top-state at the root node. What this algorithm does is to run first a *deterministic top-down* automaton $\mathcal{A}_{\text{approx}}$ during the recursive descent. This automaton is a sound approximation of $\mathcal{A}$ in the sense that for any $t \in T(\Sigma)$, $t \notin \mathcal{L}(\mathcal{A}_{\text{approx}}) \Rightarrow t \notin \mathcal{L}(\mathcal{A})$. We can make further use of this automaton $\mathcal{A}_{\text{approx}}$ by only jumping to a super-set of its relevant nodes.

Definition 4.2 (Top-down approximation) Let $\mathcal{A} = (\Sigma, Q, \mathcal{T}, \delta)$ be an ASTA. The top-down approximation of $\mathcal{A}$ is the automaton $tda(\mathcal{A}) = (\Sigma, 2^Q, \{\mathcal{T}\}, \delta_a)$ where

$$
\begin{aligned}
\delta_a = \{(S, \sigma, \rightarrow, S_1, S_2) \mid S \subseteq Q, \sigma \in \Sigma,& \\
S_i = \{q \in Q \mid \exists q' \in S, \downarrow_i q \in \delta(q', \sigma)\}\}&
\end{aligned}
$$

The exponential blow-up exhibited by this construction is avoided by computing the top-down approximation on-the-fly. The interesting part is now: what relevant nodes can be computed —and therefore which jumps can be performed— if we consider the states in $tda(\mathcal{A})$. Figure 1 illustrates the top-down approximation for the automaton $\mathcal{A}_{\text{//a//b[c]}}$ as well as the jumps that can be computed from its non-changing states. As we can see in the figure, the top-down approximation allows us to jump quite precisely in the tree. If the destination state for a subtree is $\{q_0\}$ the automaton can jump to the top-most a node in the subtree. If the destination state is $\{q_0, q_1\}$, the automaton can jump to a top-most `b` node in the subtree. If the destination state is $\{q_0, q_1, q_2\}$, no jump is possible, the automaton must perform a firstChild or nextSibling move. However, once in state $\{q_0, q_1, q_2\}$, if the label is `c` then the automaton returns in state $\{q_0, q_1\}$ and can therefore jump to find new `b` nodes. Due to space constraints, we give a more detailed description of Figure 1 in Appendix C.

Q01 /site/regions
Q02 /site/regions/europe/item/mailbox/mail/text/keyword
Q03 /site/closed_auctions/closed_auction/annotation/description/parlist/listitem
Q04 /site/regions/*/item
Q05 //listitem//keyword
Q06 /site/regions/*/item//keyword
Q07 /site/people/person[address and (phone or homepage)]
Q08 //listitem[.//keyword and .//emph]//parlist
Q09 /site/regions/*/item[mailbox/mail/date]/mailbox/mail
Q10 /site[.//keyword]
Q11 /site//keyword
Q12 /site[.//keyword]//keyword
Q13 /site[.//keyword or .//keyword/emph]//keyword
Q14 /site[.//keyword//emph]/descendant::keyword
Q15 /site[.//*//*]//keyword

Figure 2: Tree queries used in the experiments

4.4 Implementation Techniques

Hybrid Evaluation The main drawback of the top-down approximation of relevant nodes is to force a "top-down view" of the query. For instance for query //a//b[c], if a document contains a lot of `a`-nodes and few `b` nodes, the former ones will be needlessly visited since they are part of the top-down approximation of the relevant nodes. To alleviate this problem, we propose an alternative evaluation strategy dubbed hybrid evaluation. The idea is to start anywhere in the query and the document. In the case of query //a//b[c], this means starting evaluation at all `b`-nodes in the document, and check in a recursive top-down+bottom-up fashion the filter "[c]" in their subtrees and the path "//a" in their upward context. Such strategy can be effective if the count of `b`-nodes is low.

Memoization If we consider Algorithm 4.1, we see that the computations performed at Line 3 (and 7) have complexity $O(|\delta|)$. They contribute the $|Q|$ factor to the complexity $O(|Q| \cdot |D|)$ of the evaluation function. We can memoize these computations which only depends on r and $t(\pi)$ for Line 3 and r,$t(\pi)$, r_1 and r_2 for Line 7. This technique amortises the $|Q|$ factor over the whole run: except for a few "warm-up" nodes for which the all the transitions must be scanned, the rest of the run consists of a succession of look-ups in a table, one for each node visited during the run.

Information Propagation During the traversal, a node is "seen" three times by the evaluation function: (i) when reaching the node during the top-down traversal, (ii) when returning from the evaluation of the first child (iii) when returning from the evaluation of the second child. Instead of waiting (iii) to evaluate the transitions, we can already evaluate them in (ii) having only the knownledge for the first child. This reduces the number of states to verify while visiting the second child. In particular it ensures that for an XPath predicate, only one witness is checked by the automaton, the first one in pre-order (existential semantics). This is inspired from the evaluation of Non-Uniform Automata of [5].

Result Sets Since the nodes are traversed in document order and only once, result sets can be implemented as simple lists with constant time concatenation for the union of two result-sets.

5. EXPERIMENTS

We use several experiments to illustrate the behaviour of the algorithms we introduced and gauge precisely the impact of each of the optimizations and implementation techniques we presented. Due to space constraints, we do not try to give in this paper the bare performances of our implementation. The interested reader can refer to [1] where a large experimental section compares our implementation to state of the art query engines (MonetDB/XQuery and Qizx/DB), for a richer set of queries (both tree oriented and text oriented). Nevertheless, we provide for the sake of completeness a comparison of our implementation with the MonetDB/XQuery engine in Appendix D.

	Q01	Q02	Q03	Q04	Q05	Q06	Q07	Q08	Q09	Q10	Q11	Q12	Q13	Q14	Q15
(1)	1	3518	8860	22620	36511	42955	9885	5026	21851	1	73070	73070	73070	73070	73070
(2)	2	27943	42333	22628	76391	65583	66256	75727	80846	2	73071	73071	73071	73072	73074
(3)	20	353122	422060	67898	# nodes	1892764	515305	# nodes	1030955	33	# nodes	# nodes	# nodes	# nodes	# nodes
(4)	4	24	20	19	7	24	33	20	32	4	5	7	7	11	9
(5)	50	12.5	20.9	99.9	47.7	65.4	14.9	6.63	27.0	50	99.9	99.9	99.9	99.9	99.9

(1): Number of selected nodes **(2)**: Number of visited nodes with jumping **(3)**: Number of visited nodes without jumping
(4): Number of memoized transitions **(5)**: Ratio of selected nodes vs. approximated top-down relevant nodes (in %) # nodes = 5673051

Figure 3: Number of selected and visited nodes (w and wo jumping), and number of memoized configurations

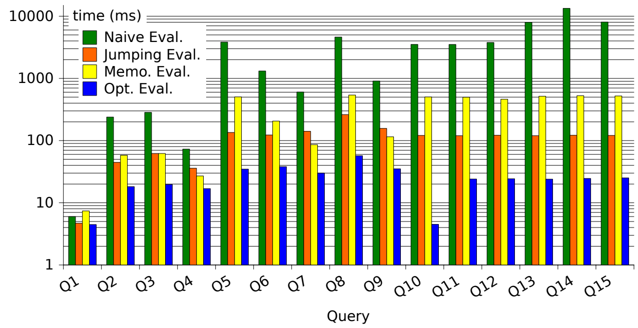


Figure 4: Impact of the jumping and memoization on query evaluation time

Implementation Our implementation[1] features a bottom-up with top-down pre-processing evaluation function ("top-down+bottom-up" as we refer to it in the rest of the section) which uses the jumping primitives described in [1]. These indexes support jumping to the first descendant and following nodes whose label is in a set L in time $O(|L|)$. As for the hybrid evaluation function, due to the lack of upward-jumping functions in this index, it performs its upward part using only parent moves (instead of jumping to ancestors with particular labels). It however remains an effective strategy when one of the labels in the query has a low count (our index provides the global count of a label in constant time).

Documents and Queries We used the XMark [19], document generator for our tests. We report our results for a document of size 116MB. The tree oriented queries we used are given in Figure 2. Q01 to Q09 are realistic queries for XMark documents, taken from the XPathMark benchmark [4]. Q10 to Q15 allow us to illustrate in more details the behaviour of our ASTAs.

Impact of Jumping and Memoization We report in Figure 4 the query answering time of our engine for each query (note the logarithmic scale for the times). The"Naive Eval." series represents a straightforward execution of Algorithm 4.1. As we can see, a naive evaluation where the $|Q|$ factor has to be paid for each node, and which potentially visits every node in D is not satisfactory. For queries where a "//" occurs at top-level, the full document needs to be traversed, yielding an evaluation time from 1s to 10s. The "Jumping Eval." series represents a run where the evaluation function computes the top-down approximation of relevant nodes on-the-fly and jumps only to these nodes. No memoization occurs therefore the $|Q|$ factor is paid for each visited node. As expected, this is a huge improvement compared to the naive case. With this optimization alone, all the tested queries require less than 150ms to evaluate, an improvement of ten to hundred-folds. The "Memo. Eval." series represents runs where on-the-fly computations are memoized. For these runs, the $|D|$ factor is paid in full (unless the automaton can skip whole subtrees as in Q01) while the $|Q|$ factor is amortized. This technique also improve query answering time considerably: a full traversal takes no more than 450ms. The fact that only firstChild and nextSibling moves are used also demonstrate that alternating automata are a framework of choice, even over pointer-based data-structures. Lastly, the "Opt. Eval" series represents runs where both optimizations are enabled. We can see that they are complementary: with the exception of Q01 and Q12, the "Opt. Eval" time is always better (at least twice as fast) as either optimization taken individually. Q01 and Q12 are a very particular case where the query only touches two nodes therefore the transitions memoized in the look-up table are never re-used and their insertions only constitute an overhead.

Approximation of Top-Down Relevance, Automata Logic and Memoization: the table in Figure 3 gives the number of selected nodes (Line **(1)**). These numbers are to be contrasted with Line **(2)**, which represents the number of nodes visited by a jumping function (that is, the size of the approximated set of relevant nodes). For realistic queries (Q01-Q09 with the exception of Q08), the number of selected nodes is more than 10% of the number of visited nodes (this ratio is given at Line **(5)**). Of particular interest is Q05. For such a query, and while the automaton is given in an alternating and non-deterministic way, we end up touching exactly the number of relevant nodes (the top-most `listitems` and the `keywords` below them). This number can be contrasted with the total number of nodes (more than 5 millions), most of which are completely ignored by the evaluation function.

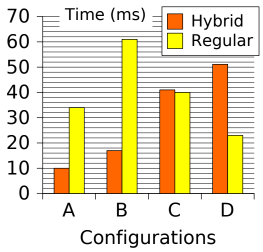
Line **(3)** shows also that for a non-jumping algorithm, our evaluation function skips, when possible, a large number of subtrees. Of course it is necessary to traverse the whole document as soon as a top-level "//" is present.

The automata logic is better highlighted by looking at Line **(2)** for query Q10 to Q15. Here, it is clear that predicates are efficiently checked. For Q11, Q12 and Q13, the predicate check is done together with the accumulation of `keyword` nodes, and no extra relevant node is touched. For query Q14 and Q15, only a small number of nodes (1 and 2 respectively) are touched in order to satisfy the predicate. Of course, the predicate need not be applied to root node, such optimizations are performed for any kind of conditions, regardless of their position in the query (it is easier to illustrate them on the single root element).

Lastly, Line **(4)** represents the number of entries added to the memoization table, or equivalently the number of nodes for which the evaluation function paid a $|Q|$ factor (whereas all the others consisted of a constant-time look-up). For practical queries, the size of such tables is very small and the speed-up they generate is worth the small memory overhead (a few kilo-bytes at most).

Hybrid Traversal Figure 5 describes the behaviour of the hybrid evaluation function for four particular configurations of XMark documents that we manually created.
We consider the query //`listitem`//`keyword`//`emph` and change the proportion and placement of the `listitem`, `keyword` and `emph` elements. For each such configurations (**A** to **D**), we report the query evaluation time for a hybrid run and for a regular

[1] Our implementation is written in OCaml (for ASTA/XPath query part) and C++ (for the indexes). Our test machine is described in Appendix D.



	A	B	C	D
(1)	4	4	65831	15074
(2)	9	11	74302	33041
(3)	70028	134247	74302	35045

(1) number of selected nodes
number of nodes visited by:
(2) a hybrid run
(3) a top-down+bottom-up run

A : 75021 `listitem`, 3 `keyword` below `listitems` (3 in total) and 4 `emphs` below those 3 `keywords`;

B : 75021 `listitem`, 60234 `keyword` below `listitems` (60234 in total) and 4 `emphs` below those `keywords`;

C : 9083 `listitem`, one `keyword` below `listitems` (40493 in total) and 65831 `emphs` below one of the `keyword` below a listitem;

D : 20304 `listitem`, 10209 `keyword` below one `listitem` (10209 in total) and 15074 `emphs` below one of those `keyword`.

Figure 5: Selected and visited nodes for the hybrid and top-down evaluation procedures, for query `//listitem//keyword//emph`

top-down+bottom-up run. We also report the number of nodes selected by the query and the number of nodes visited by both strategies. Configuration **A** and **B** represent the best cases for the hybrid traversal: one of the label in the query has a very low global count. In **A**, the count of `keyword` nodes is small, the evaluation starts at these nodes, checks in a pure bottom-up fashion that they have a `listitem` ancestor and collect their `emph` descendants. For configuration **B**, the hybrid run actually performs a pure bottom-up run of the query, starting at `emph` nodes. Both visit very few nodes compared to the relevant nodes approximated by the top-down+bottom-up evaluation (Line **(3)**). Configuration **C** represents a case where the hybrid behaves like the top-down+bottom-up run, since the global count of `keyword` elements is low. Lastly, Configuration **D** is the worst-case scenario, where `keyword` as the lowest global count, but which is close to the number of `listitem` elements. Even though the top-down+bottom-up visits more nodes, it is twice as fast thanks to its use of jumping primitives. While this particular experiment seems artificial, configuration **A** and **B** actually simulate the behaviour of text-oriented queries, where the text predicate is often very selective. Such queries where investigated in [1], where the same hybrid procedures yields significant improvement over state of the art text-aware XPath engines.

6. CONCLUSION

We have presented an effective way to reduce the number of nodes traversed during the evaluation of a navigational XPath query, using the novel notion of *relevant nodes* for an automaton. We have shown that this notion, coupled with a wide range of implementation techniques made alternating selecting tree automata a compilation target of choice for XPath queries, yielding execution speed on par with the best XPath engines available. While we have only focused our presentation on forward Core XPath, our prototype actually implements backward axes (by adding "up-moves" to formulas of the ASTA which are rewritten into down moves on-the-fly) and XPath 1.0 functions. Unfortunately "up-moves" are not part of the theory and present two problems. The first one is that we do not have yet a sound approximation of relevant nodes in the presence of up-move (therefore we cannot jump). The second, more troublesome one is that with the presence of up-moves, a single top-down followed by a bottom-up pass is not sufficient in general, one needs an extra top-down pass (as observed by Koch in [10]), or require more book-keeping operations in the result sets. XPath 1.0 functions are naively treated as black-boxes which are called during formula evaluation. This defeats some of the automata optimizations since a query "//`a`[count(.//`b`)]//c" gets compiled into three separate automata.

As future work, we plan to generalizes the top-down approximation to backward axes (it seems possible since ASTAs are known to not gain any expressive power with the addition of up-moves), extend the work in [1] to not only handle efficiently text predicates but also numeral predicates, context dependent functions (e.g. "`position()`") and data joins.

7. REFERENCES

[1] D. Arroyuelo, F. Claude, S. Maneth, V. Mäkinen, G. Navarro, K. Nguyen, J. Sirén, and N. Välimäki. Fast in-memory XPath search using compressed indexes. In *ICDE*, 2010. To appear.

[2] Ricardo A. Baeza-Yates and Gaston H. Gonnet. Fast text searching for regular expressions or automaton searching on tries. *J. ACM*, 43(6):915–936, 1996.

[3] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Online publication.

[4] M. Franceschet. XPathMark - An XPath benchmark for XMark generated data. In *XSym 2005, 3rd Int. XML Database Symposium*, LNCS n. 3671, 2005.

[5] A. Frisch. Regular tree language recognition with static information. In *TCS*. Kleuwer, 2004.

[6] G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. In *VLDB*, pages 95–106, 2002.

[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] T. J. Green, A. Gupta, G. Miklau, M. Onizuka, and D. Suciu. Processing xml streams with deterministic automata and stream indexes. *ACM Trans. Database Syst.*, 29(4), 2004.

[9] T. Grust, M. van Keulen, and J. Teubner. Staircase join: Teach a relational dbms to watch its (axis) steps. In *VLDB*, pages 524–525, 2003.

[10] C. Koch. Efficient processing of expressive node-selecting queries on XML data in secondary storage: A tree automata-based approach. In *VLDB*, pages 249–260, 2003.

[11] Michael Y. Levin. Compiling regular patterns. In *ICFP '03*, pages 65–77, New York, NY, USA, 2003. ACM.

[12] A. Neumann and H. Seidl. Locating matches of tree patterns in forests. In *FSTTCS*, pages 134–145, 1998.

[13] F. Neven and T. Schwentick. Query automata over finite trees. *Theor. Comput. Sci.*, 275(1-2):633–674, 2002.

[14] J. Niehren, L. Planque, J.-M. Talbot, and S. Tison. N-ary queries by tree automata. In *DBPL*, pages 217–231, 2005.

[15] Jukka Paakki. Attribute grammar paradigms - a high-level methodology in language implementation. *ACM Comput. Surv.*, 27(2):196–255, 1995.

[16] P. M. Pettovello and F. Fotouhi. Efficient XPath query processing. In *CASCON*, page 2, 2008.

[17] P. Mark Pettovello and F. Fotouhi. Mtree: an XML XPath graph index. In *SAC*, pages 474–481, 2006.

[18] K. Sadakane and G. Navarro. Fully-functional static and dynamic succinct trees. *CoRR*, abs/0905.0768, 2009.

[19] A. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A benchmark for XML data management. In *VLDB '02*, pages 974–985, 2002.

APPENDIX

A. SELECTING TREE AUTOMATA

We consider an example of a BDSTA for which there is no equivalent top-down deterministic STA.

Example A.1 Let $\mathcal{A}_{//\mathtt{a}[.//\mathtt{b}]} = (\Sigma, Q, \mathcal{T}, \mathcal{B}, \mathcal{S}, \delta)$ where $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, $Q = \{q_0, q_1\}$, $\mathcal{T} = \{q_0, q_1\}$, $\mathcal{B} = \{q_0\}$, $\mathcal{S} = \{(q_1, \mathtt{a})\}$, and δ consists of the following eight transitions. We now write a transition $(q, x, q', q'') \in \delta$ as $q \leftarrow x, (q', q'')$. Let $\_$ denote any state in $\{q_0, q_1\}$.

$$\begin{array}{ll} q_1 \leftarrow \quad \{\mathtt{b}\}, (q_0, \_) & q_1 \Leftarrow \quad \{\mathtt{a}\}, (q_1, \_) \\ q_0 \leftarrow \Sigma \setminus \{\mathtt{b}\}, (q_0, \_) & q_1 \leftarrow \Sigma \setminus \{\mathtt{a}\}, (q_1, \_) \end{array}$$

The automaton $\mathcal{A}_{//\mathtt{a}[.//\mathtt{b}]}$ accepts the set of all trees: $\mathcal{L}(\mathcal{A}_{//\mathtt{a}[.//\mathtt{b}]}) = T(\Sigma)$. Moreover, $\mathcal{A}_{//\mathtt{a}[.//\mathtt{b}]}$ is a bottom-up complete BDSTA. It selects all the a-nodes that have a b-node in their left subtree. In terms of XML, this automaton realizes the XPath query //a[.//b]. We claim that there is no top-down deterministic STA equivalent to $\mathcal{A}_{//\mathtt{a}[.//\mathtt{b}]}$ of Example 2.1. Intuitively, the top-down automaton does not know whether or not to select an a-node, because this depends on the left subtree of that node, which has not yet been processed by the automaton.

Definition A.1 (Reachable state) If a state q' appears in the right-hand side of a rule with q in its left-hand side, then we say that q *one-step reaches* q', denoted by $q \rightarrow_{\mathcal{A}} q'$. We denote by $\rightarrow^*_{\mathcal{A}}$ the reflexive transitive closure of $\rightarrow_{\mathcal{A}}$, and say that q *reaches* q' if $q \rightarrow^*_{\mathcal{A}} q'$.

Definition A.2 (Restriction of STA to a set of state) Let $\mathcal{A} = (\Sigma, Q, \mathcal{T}, \mathcal{B}, \mathcal{S}, \delta)$ and $\{q_1, \ldots, q_n\} \subseteq Q$, the *restriction of $\mathcal{A}$ to* $\{q_1, \ldots, q_k\}$ is the STA

$$\mathcal{A}[q_1, \ldots, q_n] = (\Sigma, Q', \mathcal{T}', \mathcal{B}', \mathcal{S}', \delta')$$

where $\mathcal{T}' = \{q_1, \ldots, q_n\}$, Q' is the set of the states reachable from $\mathcal{T}'$, i.e., $Q' = \{q' \in Q \mid \exists q \in \mathcal{T}',\ q \rightarrow^*_{\mathcal{A}} q'\}$, and $\mathcal{B}'$, $\mathcal{S}'$, and δ' are the restrictions to the states in Q' of $\mathcal{B}$, $\mathcal{S}$, and δ, i.e., $\mathcal{B}' = \mathcal{B} \cap Q'$, $\mathcal{S}' = \{(q, l) \in \mathcal{S} \mid q \in Q'\}$, and $\delta' = \{(q, L, q_1, q_2) \in \delta \mid q \in Q'\}$.

A.1 Relating STAs to Ordinary Tree Automata

It is well-known that for every ordinary deterministic tree automaton (TA) there is an equivalent unique minimal one, and that it can be computed in quadratic time. Instead of inventing and proving a new minimization procedure for STAs we prefer to encode them into ordinary tree automata in such a way that the encoding allows us to obtain a minimal STA from the minimal encoded automaton. Thus, we reduce minimization for STAs to minimization for ordinary tree automata.

We require that the STA $\mathcal{A}$ is either top-down or bottom-up complete. To encode an STA into a TA, we simply encode the selection of a node through special labels. We define the alphabet $\widehat{\Sigma} = \{\hat{\sigma} \mid \sigma \in \Sigma\}$. Now, if $\mathcal{A}$ selects a node in a given tree (with label l), then the TA $\widehat{\mathcal{A}}$ *associated to* $\mathcal{A}$ accepts a tree that has the label $\hat{l}$ at that node. Formally,

$$\widehat{\mathcal{A}} = (\Sigma \cup \widehat{\Sigma}, Q, \mathcal{T}, \mathcal{B}, \varnothing, \widehat{\delta})$$

where $\hat{\delta}$ is defined as follows. Every transition $(q, L, q_1, q_2) \in \delta$ such that there exists an $l \in L$ with $(q, l) \in \mathcal{S}$ is changed into the new transition (q, L', q_1, q_2) of $\widehat{\mathcal{A}}$ where $L' = \{l \in L \mid (q, l) \notin \mathcal{S}\}$ (if $L' = \varnothing$ then the transition is removed), and additionally we add the new transition $(q, \widehat{L}, q_1, q_2)$ to $\hat{\delta}$ where $\widehat{L} = \{\hat{L} \mid l \in L, (q, l) \in \mathcal{S}\}$. Finally, we make the automaton obtained so far complete: for every $q \in Q$ let $L(q) = \{\sigma \in \Sigma \cup \widehat{\Sigma} \mid \hat{\delta}(q) \neq \emptyset\}$ and, if $L(q) \neq \varnothing$ then add the transition $(q, L(q), q_\perp, q_\perp)$ to $\hat{\delta}$. For the new sink state $q_\perp$ we add the transition $(\hat{q}_\perp, \Sigma \cup \widehat{\Sigma}, \hat{q}_\perp, \hat{q}_\perp)$ to $\hat{\delta}$. It should be clear that

(1) for every $t \in \mathcal{L}(\mathcal{A})$ there exists a tree $t' \in \widehat{\mathcal{A}}$ which is obtained from t by changing the label of every $\pi \in \mathcal{A}(t)$ into $\hat{l}$, where $l = t(\pi)$.

(2) for every $t' \in \mathcal{L}(\widehat{\mathcal{A}})$ there exists a tree $t \in \widehat{\mathcal{A}}$ obtained by removing all hats, and, every node π in t' that has a hat, π is in $\mathcal{A}(t)$

If (1) and (2) hold for two automata $\mathcal{A}$ and $\hat{\mathcal{A}}$ then we say that they are *equivalent*, denoted by $\mathcal{A} \equiv \hat{\mathcal{A}}$.

Example A.2 The recognizer associated with the STA defined in Example 2.1 is:

$$\widehat{\mathcal{A}} = (\Sigma \cup \widehat{\Sigma}, \{\hat{q_0}, \hat{q_1}, \hat{q_\perp}\}, \{\hat{q_0}\}, \{\hat{q_0}, \hat{q_1}\}, \varnothing, \hat{\delta})$$

where $\hat{\delta}$ is defined as:

$$\begin{array}{ll} \hat{q_0}, \{\mathtt{a}\} \quad \rightarrow (\hat{q_1}, \hat{q_0}) & \hat{q_1}, \{\hat{\mathtt{b}}\} \cup \Sigma \setminus \{\mathtt{b}\} \rightarrow (\hat{q_1}, \hat{q_1}) \\ \hat{q_0}, \Sigma \setminus \{\mathtt{a}\} \rightarrow (\hat{q_0}, \hat{q_0}) & \hat{q_1}, \{\mathtt{b}\} \cup \widehat{\Sigma} \setminus \{\hat{\mathtt{b}}\} \rightarrow (\hat{q_\perp}, \hat{q_\perp}) \\ \hat{q_0}, \widehat{\Sigma} \quad \rightarrow (\hat{q_\perp}, \hat{q_\perp}) & \hat{q_\perp}, \Sigma \cup \widehat{\Sigma} \quad \rightarrow (\hat{q_\perp}, \hat{q_\perp}) \end{array}$$

The connection between an STA and its associated recognizer is quite strong, as we state in the following lemma.

Lemma A.1 *Let $\mathcal{A}$ and $\mathcal{A}'$ be two STAs, defined over the same alphabet Σ. Then $\mathcal{A} \equiv \mathcal{A}'$ if and only if $\mathcal{L}(\widehat{\mathcal{A}}) = \mathcal{L}(\widehat{\mathcal{A}'})$.*

We have seen how to translate an STA into an ordinary tree automaton. It should be clear that this translation preserves determinism. The translation is invertible: for any $\widehat{\mathcal{A}}$ automaton, one can build an equivalent (in the sense of Lemma A.1) ordinary tree automaton $\mathcal{A}$. However, this inverse translation *does not* preserve determinism. Indeed, while both formalisms are equally expressive, they do not have the same behaviour. The automaton $\widehat{\mathcal{A}}$ only needs to verify that a tree in $T(\Sigma \cup \widehat{\Sigma})$ is in its language. This can always be done in a bottom-up deterministic way (it is folklore that bottom-up tree automata can be determinized, see [3]).

For our purpose, it is enough to observe that if a deterministic automaton $\widehat{\mathcal{A}}$ is "selecting-unambiguous", then it can be transformed into a deterministic SA. Formally, the tree automaton $\mathcal{A} = (\Sigma \cup \widehat{\Sigma}, Q, \mathcal{T}, \mathcal{B}, \varnothing, \delta)$ is *selecting-unambiguous* if and only if for every $q \in Q$, and for every $t \in \mathcal{L}(\mathcal{A}[q])$:

- if $t(\epsilon) = \sigma \in \Sigma$, then $t[\epsilon \leftarrow \hat{\sigma}] \notin \mathcal{L}(\mathcal{A}[q])$
- if $t(\epsilon) = \hat{\sigma} \in \widehat{\Sigma}$, then $t[\epsilon \leftarrow \sigma] \notin \mathcal{L}(\mathcal{A}[q])$

Lemma A.2 *Let $\mathcal{A}$ be a complete TA. Then the automaton $\widehat{\mathcal{A}}$ is selecting-unambiguous.*

Lemma A.3 *Let $\mathcal{A}'$ be a complete selecting-unambiguous TDTA (resp. BDTA). There exists a complete TDSTA (resp. BDSTA) $\mathcal{A}$ such that $\hat{\mathcal{A}} \equiv \mathcal{A}'$.*

PROOF. (sketch) The proof builds the automaton $\mathcal{A}$ as such. For each transition $(q, L, q_1, q_2) \in \delta'$. We split the transition in two, $(q, L', q_1, q_2) \in \delta'$ and $(q, L'', q_1, q_2) \in \delta'$ where $L' = L \cap \Sigma$ and $L'' = L \cap \widehat{\Sigma}$ (if L' or L'' is empty, we just skip it). Since $\mathcal{A}'$ is marking-unambiguous, if $\sigma \in L'$, then $\hat{\sigma} \notin L''$ (and vice versa). If neither q_1 nor q_2 is a sink state, then we add $(q, L', q_1, q_2) \in \delta'$ as a transition to δ and if $L'' = \{\hat{\sigma_1}, \ldots, \hat{\sigma_k}\}$ we add $(q, \{\sigma_1, \ldots, \sigma_k\}, q_1, q_2)$ to δ and (q, σ_i) to $\mathcal{S}$. Once this is done for all transitions, we remove all unreachable states and we obtain $\mathcal{A}$.

A.2 Minimization

As mentioned before, minimal here means, the smallest number of states. Given a BDTA $\mathcal{A} = (\Sigma, Q, \mathcal{T}, \mathcal{B}, \delta)$, the standard algorithm for minimization (see, e.g., [3]) builds the set of equivalence classes for every state in Q. Two states q and q' are in the same equivalence class if and only if $\mathcal{L}(\mathcal{A}[q]) = \mathcal{L}(\mathcal{A}[q'])$. The algorithm initializes the set of equivalence classes with $E_0 = \{Q \setminus \mathcal{T}, \mathcal{T}\}$. The intuition is that final and non-final states are not in the same equivalence classes (indeed, if q is a final state and q' not a final state, then $\mathcal{A}[q]$ accepts the null tree $\#$ while $\mathcal{A}[q']$ does not, hence $\mathcal{L}(\mathcal{A}[q]) \neq \mathcal{L}(\mathcal{A}[q'])$). The algorithm proceeds then to refine the equivalence relation. We note $q\ E_n\ q'$ the fact that q and q' are equivalent in the equivalence relation E_n, that is there exists $S \in E_n$ such that $q \in S$ and $q' \in S$. From E_n the algorithm computes a finer equivalence relation E_{n+1} such that $q\ E_{n+1}\ q'$ if:

- $q\ E_n\ q'$;
- $\forall \sigma \in \Sigma, \forall q_1, q_2 \in Q$: $\delta(q1, q, l) = \delta(q1, q', l)$ and $\delta(q, q_2, l) = \delta(q', q2, l)$.

The procedures stops when $E_n = E_{n+1}$. The case of TDTA is similar.

Of course we would like, given a selecting automaton $\mathcal{A}$, to compute is associated recognizer $\widehat{\mathcal{A}}$, minimize it using the standard procedure and translate it back into a selecting automaton. However, as we have seen, translating a recognizer into a selecting automaton does not always preserve determinism. Fortunately, we can show that the property of selecting unambiguousness is preserved by the minimization procedure.

Lemma A.4 *Let $\widehat{\mathcal{A}}$ be a complete TDTA (resp. BDTA) over the alphabet $\Sigma \cup \widehat{\Sigma}$. Let $\widehat{\mathcal{A}}_{min}$ be the minimal automaton such that $\mathcal{L}(\widehat{\mathcal{A}}_{min}) = \mathcal{L}(\widehat{\mathcal{A}})$. If $\widehat{\mathcal{A}}$ is selecting-unambiguous, then so is $\widehat{\mathcal{A}}_{min}$.*

PROOF. Since $\widehat{\mathcal{A}}$ is selecting-unambiguous it holds that $\forall q \in \widehat{Q}, \forall t \in \mathcal{L}(\widehat{\mathcal{A}}[q])$, if $t(\epsilon) = \sigma \in \Sigma$ then $t[\epsilon \leftarrow \hat{\sigma}] \notin \mathcal{L}(\widehat{\mathcal{A}}[q])$ and if $t(\epsilon) = \hat{\sigma} \in \widehat{\Sigma}$ then $t[\epsilon \leftarrow \sigma] \notin \mathcal{L}(\widehat{\mathcal{A}}[q])$.

Now suppose that there are two states $q_1, q_2 \in \widehat{Q}$ such that $\exists \sigma(t_1, t_2) \in \mathcal{L}(\widehat{\mathcal{A}}[q_1])$ and $\exists \hat{\sigma}(t_1, t_2) \in \mathcal{L}(\widehat{\mathcal{A}}[q_2])$. It holds that $\widehat{\mathcal{A}}_{min}$ is selecting-unambiguous if and only if q_1 and q_2 are not in the same equivalence class (if they where, then there would be a state in $q' \in Q_{min}$ for which the selecting-unambiguous property do not hold, the state representing the equivalence class of q_1 and q_2). We must therefore show that $\mathcal{L}(\widehat{\mathcal{A}}[q_1]) \neq \mathcal{L}(\widehat{\mathcal{A}}[q_2])$ This is immediate: since $\mathcal{A}$ is selecting unambiguous, and since $\sigma(t_1, t_2) \in \mathcal{L}(\widehat{\mathcal{A}}[q_1])$, then $\hat{\sigma}(t_1, t_2) \notin \mathcal{L}(\widehat{\mathcal{A}}[q_1])$. However $\hat{\sigma}(t_1, t_2) \in \mathcal{L}(\widehat{\mathcal{A}}[q_2])$ and therefore $\mathcal{L}(\widehat{\mathcal{A}}[q_1]) \neq \mathcal{L}(\widehat{\mathcal{A}}[q_2])$.

Using this lemma, we can state the existence of a minimal selecting tree automaton.

Theorem A.1 *Let $\mathcal{A}$ be a complete TDSTA (resp. BDSTA). There exists a complete TDSTA (resp. BDSTA) $\mathcal{A}_{min}$ which is equivalent to $\mathcal{A}$ and no other equivalent TDSTA (resp. BDSTA) has less states than $\mathcal{A}_{min}$.*

Theorem A.1 states the existence of a minimal selecting automaton and also give a way to compute it. Indeed, it is sufficient to translate a selecting automaton into a recognizer, minimize the latter and transform it back into a selecting automaton. However, the proof of Lemma A.4 hints us toward a more direct method. Indeed in a recognizer, if a state $\hat{q}_1$ accepts some tree $\sigma(t_1, t_2)$ and a state $\hat{q}_2$ accepts the tree $\hat{\sigma}(t_1, t_2)$, then $\hat{q}_1$ and $\hat{q}_2$ are in different equivalence classes. In the transformation from recognizer to selecting automaton, q_2, σ becomes a selecting configuration. Therefore, if two states q_1 and q_2 are such that $q_1, \sigma \notin \mathcal{S}$ and $q_2, \sigma \in \mathcal{S}$ then these two states are not in the same equivalence class. Minimizing an selecting automaton can therefore be achieved by using the standard algorithm, but where the initial relation E_0 is:

$$E_0 = \begin{array}{l} \{\{q \in Q \mid q \in \mathcal{F}, q \in \mathcal{S}\}, \{q \in Q \mid q \in \mathcal{F}, q \notin \mathcal{S}\}, \\ \{q \in Q \mid q \notin \mathcal{F}, q \in \mathcal{S}\}, \{q \in Q \mid q \notin \mathcal{F}, q \in \mathcal{S}\}\}. \end{array}$$

Here $\mathcal{F}$ stands for the set of final states, that is $\mathcal{T}$ for BDTAs and $\mathcal{B}$ for TDTAs.

B. RELEVANCE

B.1 Top-Down Relevance

Algorithm B.1 (Top-down traversal with jumping)

Input: Minimal TDTA $\mathcal{A} = (\Sigma, Q, \mathcal{F}, \mathcal{I}, \mathcal{S}, \delta)$ and a tree t

Output: (possibly empty) Mapping from nodes of t to states of $\mathcal{A}$.

```
let following(π, L, π0)=
  if π = Ω then return ∅
  else return {π}∪following(f_t(π, L, π0),L,π0);

let relevant_nodes (t, π, q) =
   if ∃L ⊂ Σ, (q, L, q, q) ∈ δ and ¬is_marking(q)
   then { L′ := Σ \ L;
      if t(π) ∈ L′ then return {π};
      π′ := d_t(π, L′);
      return {π′}∪follow(π′,L′, π)
   } else
   if ∃L ⊂ Σ, (q, L, q, q_⊤) ∈ δ
      and is_universal (q_⊤) and ¬is_marking(q)
   then { L′ := Σ \ L;
      if t(π) ∈ L′ then return {π};
      π′ := l_t(π, L′);
      if π′ = Ω then return ∅ else return {π′}
   } else
   if ∃L ⊂ Σ, (q, L, q_⊤, q) ∈ δ
      and is_universal (q_⊤) and ¬is_marking(q)
   then { L′ := Σ \ L;
      if t(π) ∈ L′ then return {π};
      π′ := l_t(π, L′);
      if π′ = Ω then return ∅ else return {π′}
   } else
   return {π};

let td_jump_rec (π, q) =
 l: = t(π);
 if l = # then
   if q ∈ B then return {π ↦ q}
   else throw Failure
 else {
 {q1, q2} := δ(q, l);
 if is_sink (q1) or is_sink(q2) then throw Failure;
 lnodes := relevant_nodes (t, π · 1, q1);
 rnodes := relevant_nodes (t, π · 2, q2);
 return {π ↦ q} ∪   ⋃_{π1 ∈ lnodes}  topdown_jump_rec(π1, q1)
                 ∪   ⋃_{π2 ∈ rnodes}  topdown_jump_rec(π2, q2);
 }

let topdown_jump(t,(Σ, Q, F, {q}, S, δ)) =
 try {
  nodes := relevant_nodes (t, ϵ, q);
  return ⋃_{π ∈ nodes} topdown_jump_rec(π, q);
 } catch (Failure) { return ∅ }
```

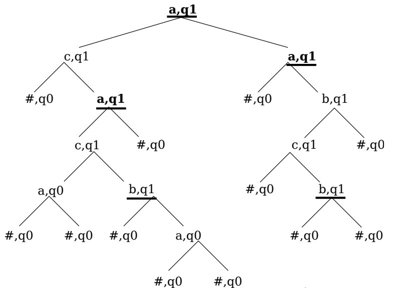


Figure 6: Bottom-up run of automaton $\mathcal{A}$ from Example B.1

B.2 Bottom-Up Relevance

Algorithm B.2 (Bottom-up evaluation)
Input : A BDTA $\mathcal{A} = \{\Sigma, Q, \mathcal{T}, \{q_0\}, \mathcal{S}, \delta\}$ a tree t a sequence

$$S_0 = (\pi_0, q_0),\ \ (\pi_1, q_0),\ \ \ldots,\ \ (\pi_n, q_0)$$

where the π_i are the leaves of t in pre-order.

Output : A mapping from nodes of t to states of $\mathcal{A}$

```
let bottom_up_rec (S, t, R) =
 switch S {
 case (π, q):
  if π = ε and q ∈ T
  then return {ε ↦ q} ∪ R,();
  else throw Failure;

 case (π1, q1),(π2, q2),S':
  if siblings (π1, π2)
  then {
     π := parent π1;
     {q} := δ(q1, q2, t(π));
  return bottom_up_rec(((π, q), S'), t, {π ↦ q} ∪ R)
  } else {
     R',S'' := bottom_up_rec(((π2, q2), S'), t, R);
     return bottom_up_rec (((π1, q1), S''), t, R');
 case ():
     return R,();
  }

let bottom_up (t, S0, A) =
  try {
   R,_ := bottom_up_rec(S0, t, {π ↦ q | (π, q) ∈ S0});
   return R;
  } catch (Failure)
      return ∅;
```

$$\mathcal{A}_{//\mathtt{a[.//b]}} = (\underbrace{\{\mathtt{a},\mathtt{b},\mathtt{c}\}}_{\Sigma}, \underbrace{\{q_0, q_1\}}_{Q}, \underbrace{\{q_0\}}_{\mathcal{T}}, \underbrace{\{q_0, q_1\}}_{\mathcal{B}}, \underbrace{\{(q_1, \mathtt{a})\}}_{\mathcal{S}}, \delta)$$

Example B.1 A transition $(q, L, q', q'') \in \delta$ is written in the form $q \leftarrow L(q', q'')$ and the wildcard $\_$ denotes any state in $\{q_0, q_1\}$. δ is defined by:

$$\begin{array}{llll} q_1 \leftarrow & \{\mathtt{b}\}, (q_0, \_) & q_1 \Leftarrow & \{\mathtt{a}\}, (q_1, \_) \\ q_0 \leftarrow \Sigma \setminus & \{\mathtt{b}\}, (q_0, \_) & q_1 \leftarrow \Sigma \setminus & \{\mathtt{a}\}, (q_1, \_) \end{array}$$

A run of this automaton on an input tree is given in Figure 6. This automaton selects all the a-labelled node which are above a b-labelled node. The selected nodes are circled and the relevant nodes are underlined. As in the general case and the TDSTA case, selected nodes are relevant. Otherwise, we can remark that any subtree whose root is in state q_0 contain only non relevant nodes. In the case of minimal BDSTAs, the state q_0 allows to skip subtrees (as $q_\top$ for TDSTAs). Indeed in a minimal BDSTA, q_0 is the only state which accepts a null-tree $\#$. But conversely, any subtree which is recognized in q_0 could be replaced by a null-tree without changing the semantics of the query. Thus, skipped subtrees are those whose root is in state q_0. For skipping nodes along a path, the same conditions as previously apply: either the automaton remains in the same state for a node and both its children, or the root and one of its children are in the same state and the other children can be skipped, that is, is in state q_0.

C. AUTOMATA FOR XPATH

Definition C.1 (XPath fragment) An *XPath expression* is a finite production of the following grammar, with start symbol Core:

Core	::=	LocationPath | '/' LocationPath
LocationPath	::=	LocationStep ('/' LocationStep)*
LocationStep	::=	Axis '::' NodeTest
		| Axis '::' NodeTest '[' Pred ']'
Pred	::=	Pred 'and' Pred | Pred 'or' Pred
		| 'not' '(' Pred ')' | Core | '(' Pred ')'
Axis	::=	`descendant` | `child`
		| `following-sibling` | `attribute`
NodeTest	::=	*tag* | $\star$ | `node()` | `text()`

The following example clearly shows why using normal STAs would cause an exponential blow-up:

Example C.1 Consider the XPath query:
`//x[ (a1 or a2) and ... and (a2n-1 or a2n) ]`
where the $\mathtt{a}_i$ are pairwise distinct labels. The ASTA for this query is:

$$\begin{array}{ll} q_{\mathtt{x}}, \{x\} & \Rightarrow (\downarrow_1 q_{\mathtt{a}_1} \vee \downarrow_1 q_{\mathtt{a}_2}) \wedge \ldots \wedge (\downarrow_1 q_{\mathtt{a}_{2n-1}} \vee \downarrow_1 q_{\mathtt{a}_{2n}}) \\ q_{\mathtt{x}}, \Sigma & \rightarrow \downarrow_1 q_{\mathtt{x}} \vee \downarrow_2 q_{\mathtt{x}} \\ q_{\mathtt{a}_i}, \{q_{\mathtt{a}_i}\} & \rightarrow \top \\ q_{\mathtt{a}_i}, \Sigma & \rightarrow \downarrow_2 q_{\mathtt{a}_i} \end{array}$$

This ASTA has: $2 \cdot n + 1$ states, $4 \cdot n + 2$ transitions, one of length $2 \cdot n$ and the other of fixed length (less than 3). It is well known that converting this ASTA into an STA yield an exponential blow-up (since one has to compute the disjunctive normal form of the formulas; for the first transition, the DNF has size 2^n).
Evaluation of formulas and node selection: We define the notion of result sets an the semantics of the evaluation of formulas, which also handles node selection.

Definition C.2 (Result set) Let $\mathcal{A} = (\Sigma, \mathcal{Q}, \mathcal{T}, \delta)$ be an ASTA and $t \in T(\Sigma)$. A *result set* is a mapping from states in Q to sets of nodes in $\mathcal{D}om(t)$. Given a mapping Γ, we denote by $\Gamma(q)$ the set of states associated with q (the empty set if q is not in $\mathcal{D}om(\Gamma)$) and we define the union of two mappings as:

$$(\Gamma_1 \cup \Gamma_2)(q) = \Gamma_1(q) \cup \Gamma_2(q)$$

We can now define the evaluation of a set of transitions for an automaton.

Definition C.3 (Evaluation of a set of transitions) Let

$$\mathcal{A} = (\Sigma, \mathcal{Q}, \mathcal{T}, \delta)$$

be an ASTA, $t \in T(\Sigma)$ a tree and *Trs* $\subseteq \delta$ a set of transitions. The evaluation of *Trs* for a node $\pi \in \mathcal{D}om(t)$ is a result set given by the function:

$$\begin{array}{l} \mathtt{eval\_trans}(\Gamma_1, \Gamma_2, \pi, \mathit{Trs}) = \\ \qquad \displaystyle\bigcup_{(q,L,\rightarrow,\phi)\in \mathit{Trs}} \{q \mapsto S \mid \Gamma_1, \Gamma_2 \vdash_{\mathcal{A}} \phi = (\top, S)\} \\ \cup \displaystyle\bigcup_{(q,L,\Rightarrow,\phi)\in \mathit{Trs}} \{q \mapsto \{\pi\} \cup S \mid \Gamma_1, \Gamma_2 \vdash_{\mathcal{A}} \phi = (\top, S)\} \end{array}$$

where Γ_1 and Γ_2 are result sets, and $\Gamma_1, \Gamma_2 \vdash_{\mathcal{A}} \phi = (b, S)$ is the judgement derived by the rules in Figure 7.

These rules are pretty straightforward and combine the rules for a classical alternating automaton, with the rules of a marking automaton. Rule **(or)** and **(and)** implements the Boolean connective

$$\frac{}{\Gamma_1, \Gamma_2 \vdash_{\mathcal{A}} \top = (\top, \emptyset)}\textbf{(true)} \quad \frac{\Gamma_1, \Gamma_2 \vdash_{\mathcal{A}} \phi = (b, R)}{\Gamma_1, \Gamma_2 \vdash_{\mathcal{A}} \neg\phi = (\bar{b}, \emptyset)}\textbf{(not)}$$

$$\frac{\Gamma_1, \Gamma_2 \vdash_{\mathcal{A}} \phi_1 = (b_1, \Gamma_1') \quad \Gamma_1, \Gamma_2 \vdash_{\mathcal{A}} \phi_2 = (b_2, \Gamma_2')}{\Gamma_1, \Gamma_2 \vdash_{\mathcal{A}} \phi_1 \vee \phi_2 = (b_1, \Gamma_1') \otimes (b_2, \Gamma_2')}\textbf{(or)}$$

$$\frac{\Gamma_1, \Gamma_2 \vdash_{\mathcal{A}} \phi_1 = (b_1, \Gamma_1') \quad \Gamma_1, \Gamma_2 \vdash_{\mathcal{A}} \phi_2 = (b_2, \Gamma_2')}{\Gamma_1, \Gamma_2 \vdash_{\mathcal{A}} \phi_1 \wedge \phi_2 = (b_1, \Gamma_1') \oslash (b_2, \Gamma_2')}\textbf{(and)}$$

$$\frac{q \in \mathcal{D}om(\Gamma_i)}{\Gamma_1, \Gamma_2 \vdash_{\mathcal{A}} \downarrow_i q = (\top, \Gamma(q))} \text{for } i \in \{1,2\} \textbf{ (left,right)}$$

$$\frac{\text{when no other rule applies}}{\Gamma_1, \Gamma_2 \vdash_{\mathcal{A}} \phi = (\bot, \emptyset)}$$

where:

$\overline{\top} = \bot \; \overline{\bot} = \top$

$$(b_1, \Gamma_1) \otimes (b_2, \Gamma_2) = \begin{cases} \top, \Gamma_1 & \text{if } b_1 = \top, b_2 = \bot \\ \top, \Gamma_2 & \text{if } b_2 = \top, b_1 = \bot \\ \top, \Gamma_1 \cup \Gamma_2 & \text{if } b_1 = \top, b_2 = \top \\ \bot, \emptyset & \text{otherwise} \end{cases}$$

$$(b_1, \Gamma_1) \oslash (b_2, \Gamma_2) = \begin{cases} \top, \Gamma_1 \cup \Gamma_2 & \text{if } b_1 = \top, b_2 = \top \\ \bot, \emptyset & \text{otherwise} \end{cases}$$

Figure 7: Inference rules defining the evaluation of a formula

of the formula and collect the marking found in their true sub-formulas. Rules **(left)** and **(right)** (written as a rule schema for concision) evaluate to true if the state q is in the corresponding set. Intuitively, states in Γ_1 (resp. Γ_2) are those accepted in the left (resp. right) subtree of the input tree. To handle selection, we proceed as follows. Assuming the left subtree returned a result set Γ_1 and the right subtree a result set Γ_2:

(1) For each $q, L \Rightarrow \phi$ such that ϕ evaluates to $\top$ ($\downarrow_i q'$ evaluates to $\top$ if $q' \in \mathcal{D}om(\Gamma_i)$), add the mapping $q \mapsto \{\pi\}$ to Γ;

(2) For each $q, L \rightarrow \phi$ or $q, L \Rightarrow \phi$, for which ϕ evaluates to $\top$, if $\downarrow_i q' \in \phi$ evaluates to $\top$, add the mapping $q \mapsto \Gamma_i(q')$ to Γ.

This is done by the function `eval_trans` Informally we remember each node which was selected by a particular transition (1) and for each selected node in state q' we propagate it to q if it contributes to the truth of a formula proving q. The selected nodes which gets propagated to a state in $\mathcal{T}$ are therefore part of an accepting run and constitute the result of the query. If we take the example run given in Figure 1 of Section 4, node selection is performed as follows. Consider the rightmost c node in the figure ($\star$). This node was entered in state $\{q_0, q_1, q_2\}$, therefore the active transitions for it are:

$$\{q_0, \Sigma \rightarrow \downarrow_1 q_0 \vee \downarrow_2 q_0;\;\; q_1, \Sigma \rightarrow \downarrow_1 q_1 \vee \downarrow_2 q_1;\;\; q_2, \{c\} \rightarrow \top;\;\; q_2, \Sigma \rightarrow \downarrow_2 q_2\}$$

and the result sets for its left and right subtrees are $\varnothing$ (since the calls to both left and right move failed). In this environment only the third transition is satisfied, the result set returned is therefore $\Gamma_1 = \{q_2 \mapsto \varnothing\}$. Returning from the recursive calls, we arrive on the b node above it, for which the active transitions are:

$$\{q_0, \Sigma \rightarrow \downarrow_1 q_0 \vee \downarrow_2 q_0;\; q_1, \{b\} \Rightarrow \downarrow_1 q_2;\; q_1, \Sigma \rightarrow \downarrow_1 q_1 \vee \downarrow_2 q_1;\}$$

Evaluated under the results $(\Gamma_1, \varnothing)$ for the left and right subtrees, only the second transition is satisfied. Furthermore, this transition is a selecting one, it therefore returns result set $\Gamma_2 = \{q_1 \mapsto \{\pi_b\}\}$ where π_b is the identifier of this node. The parent of this b node is again a b node where the same transitions are active. However the result sets for the left and right subtrees are $(\varnothing, \Gamma_2)$. Under these hypothesis only the third transition can be satisfied (and it is a not a selecting one). The current b node is therefore not selected, but the result set is $\Gamma_3 = \{q_1 \mapsto \Gamma_2(q_1)\}$ (since $\downarrow_2 q_1$ evaluated to $\top$ during the evaluation of the third transition). We have $\Gamma_3 = \{q_1 \mapsto \{\pi_b\}\}$. We now move onto the a parent of this b node, where the active transitions are:

$$\{q_0, \{a\} \rightarrow \downarrow_1 q_1;\; q_0, \Sigma \rightarrow \downarrow_1 q_0 \vee \downarrow_2 q_0;\}$$

evaluated under the assumptions $(\Gamma_3, \varnothing)$. Here the first formula evaluates to $\top$, yielding the result set $\Gamma_4 = \{q_0 \mapsto \Gamma_3(q_1)\} = \{q_0 \mapsto \{\pi_b\}\}$. We now see that the node $\{\pi_b\}$ has been "promoted" to state q_0. Using this technique we can ensure that nodes selected non-deterministically during the bottom-up run are kept only if they propagate up to the starting state q_0, in which case they are part of the result. We illustrate how these rules work by explaining the evaluation of the automaton in Figure 1.

Detailed explanation of Figure 1: With respect to node selection, consider the rightmost c node in the figure ($\star$). This node was entered in state $\{q_0, q_1, q_2\}$, therefore the active transitions for it are:

$$\{q_0, \Sigma \rightarrow \downarrow_1 q_0 \vee \downarrow_2 q_0;\;\; q_1, \Sigma \rightarrow \downarrow_1 q_1 \vee \downarrow_2 q_1;\;\; q_2, \{c\} \rightarrow \top;\;\; q_2, \Sigma \rightarrow \downarrow_2 q_2\}$$

and the result sets for its left and right subtrees are $\varnothing$ (since the calls to both left and right move failed). In this environment only the third transition is satisfied, the result set returned for this node is therefore $\Gamma_1 = \{q_2 \mapsto \varnothing\}$. Returning from the recursive calls, we arrive on the b node above it, for which the active transitions are:

$$\{q_0, \Sigma \rightarrow \downarrow_1 q_0 \vee \downarrow_2 q_0;\; q_1, \{b\} \Rightarrow \downarrow_1 q_2;\; q_1, \Sigma \rightarrow \downarrow_1 q_1 \vee \downarrow_2 q_1;\}$$

Evaluated under the results $(\Gamma_1, \varnothing)$ for the left and right subtrees, only the second transition is satisfied. Furthermore, this transition is a selecting one, it therefore returns result set $\Gamma_2 = \{q_1 \mapsto \{\pi_b\}\}$ where π_b is the identifier of this node. The parent of this b node is again a b node where the same transitions are active. However the result sets for the left and right subtrees are $(\varnothing, \Gamma_2)$. Under these hypothesis only the third transition can be satisfied (and it is a not a selecting one). The current b node is therefore not selected, but the result set is $\Gamma_3 = \{q_1 \mapsto \Gamma_2(q_1)\}$ (since $\downarrow_2 q_1$ evaluated to $\top$ during the evaluation of the third transition). We have $\Gamma_3 = \{q_1 \mapsto \{\pi_b\}\}$. We now move onto the a parent of this b node, where the active transitions are:

$$\{q_0, \{a\} \rightarrow \downarrow_1 q_1;\; q_0, \Sigma \rightarrow \downarrow_1 q_0 \vee \downarrow_2 q_0;\}$$

evaluated under the assumptions $(\Gamma_3, \varnothing)$. Here the first formula evaluates to $\top$, yielding the result set $\Gamma_4 = \{q_0 \mapsto \Gamma_3(q_1)\} = \{q_0 \mapsto \{\pi_b\}\}$. We now see that the node $\{\pi_b\}$ has been "promoted" to state q_0. Using this technique we can ensure that nodes selected non-deterministically during the bottom-up run are kept only if they propagate up to the starting state q_0, in which case they are part of the result.

D. EXPERIMENTS

Experimental Setup tests were executed on an Intel Xeon Core 2 Duo, 3 Ghz, with 4GB of RAM. We used Ubuntu Linux 9.10 distribution, with kernel 2.6.32 and 64 bits userland. Our implementation was compiled using g++ 4.4.1 and OCaml 3.11.1. We used version v4.34.0 of the MonetDB Server, with 32 bits OIDs. Experimental results for query Q01 to Q15 are given in Figure 8. For both engines, the results was materialized in memory but not serialized. We took the best of 5 consecutive runs for each query.

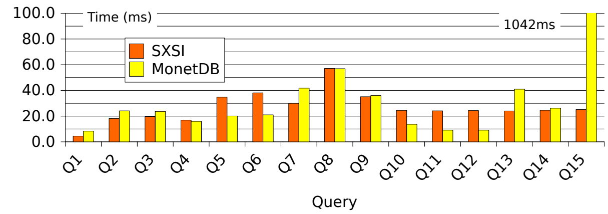


Figure 8: Query answering time for the SXSI and MonetDB