Stream Processing of XPath Queries with Predicates

Ashish Kumar Gupta
University of Washington
akgupta@cs.washington.edu

Dan Suciu
University of Washington
suciu@cs.washington.edu

ABSTRACT

We consider the problem of evaluating large numbers of XPath filters, each with many predicates, on a stream of XML documents. The solution we propose is to lazily construct a single deterministic pushdown automata, called *the XPush Machine* from the given XPath filters. We describe a number of optimization techniques to make the lazy XPush machine more efficient, both in terms of space and time. The combination of these optimizations results in high, sustained throughput. For example, if the total number of atomic predicates in the filters is up to 200000, then the throughput is at least 0.5 MB/sec: it increases to 4.5 MB/sec when each filter contains a single predicate.

1. INTRODUCTION

A promising approach to intra- and inter-enterprise integration is through message-oriented middleware servers (MOM), in particular XML message brokers. These systems allow applications to exchange information by sending XML messages, and by subscribing to such messages. The broker's main task is to route the messages from producers to the consumers. It may also perform additional tasks, such as simple message transformations and backups. Major database vendors, like IBM and Oracle, already offer complete message brokers, and a number of startup companies are addressing specifically the XML routing problem[1].

The core technical challenge in such systems is processing a large collection of XPath queries (filters) on an incoming stream of XML packets. We call this the *XML stream processing* problem. Each filter is a boolean expression, so the answer consists, for each XML document, of a set of query IDs that are true on the document. The XML stream processing problem occurs in XML packet routing [20], selective dissemination of information [2], and notification systems [17].

[1]Some of the companies are: Fiorano Software, Sarvega, Elitesecureweb, Knowhow, Xbridgesoft, XmlBlaster.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for pro£t or commercial advantage and that copies bear this notice and the full citation on the £rst page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior speci£c permission and/or a fee.
SIGMOD 2003, June 9-12, 2003, San Diego, CA.
Copyright 2003 ACM 1-58113-634-X/03/06 ...$5.00.

The difficulty in XML stream processing is that the number of XPath queries in the workload is very high. A naive approach to query evaluation, which computes each query separately, obviously doesn't scale. Previous approaches [2, 4, 13, 11] have addressed this problem by identifying and eliminating common subexpression in the *structure navigation* part of the XPath queries. However, no technique exists today for eliminating redundant work in the *predicate evaluation* part of the XPath queries. Unfortunately, the computation time is dominated by the latter when queries have multiple predicates, which is typical in XML messaging systems.

Example 1.1 [Running example] Consider the following two XPath queries:

```
P1 = //a[b/text()=1 and .//a[@c>2]]
P2 = //a[@c>2 and b/text()=1]
```

We will use this workload as our running example throughout the paper. The structure navigation part consists of evaluating paths like `//a, //a/b, //a//a[@c]` etc to find the atomic values that need to be tested, while the predicate evaluation part evaluates the atomic predicates, then combines them with the `and` connectors in the queries. Previous techniques to XML stream processing eliminate common subexpressions in the first part, but cannot exploit, for the example, the fact that the predicate `[b/text()=1]` is common to the two queries. When the workload has many (say tens of thousand) XPath queries, each with several (say 5-20) predicates, such common predicates are frequent, and keeping track of them separately for each query degrades the performance significantly.

One approach to the predicate evaluation problem is to group queries sharing one or more common predicates: this is used, for example, in continuous queries [7, 8]. For each data packet, the common predicates in the group are evaluated first. If they are true, and no query in the group has any additional predicates, then we are done with this group; otherwise we need to fall back to evaluating the remaining predicates separately. This approach works best when the groups have little overlap and most queries in each group have no additional predicates. Otherwise it degenerates to a naive evaluation method. An important limitation of this approach is that it requires direct access to the XML document: the predicate evaluation order is decided by the optimizer (usually starting with the most selective predicate), and

is not the document input order, hence we need to have a DOM representation of the XML packet.

We present in this paper a new approach to processing XPath expressions on streaming XML data that eliminates both common subexpressions in the structure navigation and in the predicates. Moreover, this technique works on a stream representation of the XML document, by using a SAX parser, hence requires no main memory representation of the document. Our technique scales to both large numbers of XPath expressions *and* to large numbers of predicates per query. Predicates are combined with `and`, `or`, and `not`, and can be interleaved arbitrarily with the navigation.

Our goal is to perform a constant amount of work for each SAX event. In particular if the SAX event causes many predicates to become true, these predicates need to be handled like a single group. Processing predicates in XPath expressions leads naturally to a bottom-up evaluation on the XML tree. For example, in order to evaluate `a[b/text()=1 and c/text()=2]` on some `<a>...</a>` element we need to check first the predicates `b/text()=1` and `c/text()=2` on `a`'s children, and only then can we conclude that the entire XPath expression is true on the `<a>...</a>` element. To avoid a main memory representation of the entire XML tree, we process the stream of SAX events with a stack, simulating the bottom-up computation. The information we push on the stack keeps track of which predicates have evaluated to `true`: each stack symbol corresponds to a set of predicates in the XPath workload. For example, when `<a>` is first encountered we push $\emptyset$ on the stack, denoting the fact that no predicates are true yet on this node; when we start processing its first child, the stack is $(\ldots, \emptyset, \emptyset)$. If one of `a`'s children is `<b> 1 </b>`, then after processing it the stack becomes $(\ldots, \emptyset, \{$`b/text()=1`$\})$; that is, still no predicate is true on `<a>`, but the predicate `b/text()=1` has been checked on the level below. Now if we see a child `<c> 2 </c>` of `a`: then the stack becomes $(\ldots, \emptyset, \{$`b/text()=1`$,$ `c/text()=2`$\})$; finally, when the end tag `</a>` is encountered, the the stack becomes $(\ldots, \{$`a[b/text()=1, and c/text()=2]`$\})$, and we can conclude that the entire XPath expression is true on the `<a>...</a>` element.

We translate the entire XPath workload into a single deterministic pushdown automaton [15]. We modify the definition of the pushdown automaton to adapt it to XPath queries and XML data and call the resulting formalism an *XPush Machine*. We show how to translate an entire workload of XPath filters into a single XPush machine. A state in the XPush machine corresponds to a set of predicates in the XPath queries. To avoid the theoretical exponential state blow-up, we compute the XPush machine lazily. There is a relatively high cost in computing an XPush state at runtime, but we recover that cost when the state is reused. Several heuristic-based optimizations are possible on the XPush machine, and we discuss here a few, showing their effectiveness. The goal of the optimizations is three-fold: to reduce the total number of XPush states (thus saving memory), to reduce the number of predicates per state (thus making them easier to compute), and to pre-compute some of the states.

Related Work In a seminal paper by Hoffman and O'Donnell [14], the *tree pattern matching* problem was introduced, in which a subject tree (the data) has to be matched with a set of tree patterns (the queries). The problem was motivated by several applications, and has since spawned a large amount of work [3, 21, 10, 6, 16]. Hoffmann and O'Donnell show that the tree patterns can be preprocessed into a data structure of exponential size, which factors out all common subpatterns, such that every subject tree can subsequently be matched bottom-up in linear time. In our work we share similar goals, but cannot apply those techniques directly because tree patterns are ordered, have no wildcards $(*, //)$, and the exponential size data structure is prohibitive for large workloads of XPath queries. The lazy XPush machine and its associated optimizations can be viewed as a significant generalization and improvement of the tree pattern matching technique for the specific task of evaluating XPath queries.

There is an alternative approach to pattern matching that does not require any kind of preprocessing. To date, the fastest top-down algorithm known is $O(n \log^3 n)$, where n is the combined subject plus pattern size [10]. A good introduction to this literature can be found in [6]. The complexity of the XPath evaluation problem is discussed in [12].

Several research projects have discussed evaluation of XPath expressions on XML streams. The XFilter system [2] was the first to define the problem, and to describe several evaluation techniques. It builds a separate FSM for each query; as a result it does not exploit commonality that exists among the path expressions; XTrie [4] indexes sub-strings of path expressions that only contain parent-child operators, and shares the processing of only these common sub-strings among the queries; YFilter [11] detects all common prefixes, including wildcards and descendant axes; the entire workload is converted into a lazy DFA in [13]. None of these systems detect common predicates. A technique that goes in this direction is the event notification system described in [17], where complex events are defined as conjuncts of atomic events, and common atomic events are identified with a trie structure. Another system that moves in this direction is NiagaraCQ [7], where a set of conjunctive relational queries is continuously evaluated on relational data sources that keep getting updated.

A technique for evaluating XPath expressions using stack machines is described in [18]. In that approach one single XPath expression is translated into multiple pushdown automata that are connected by a network and need to be run in parallel and synchronized. Such a translation is not adequate for our purposes because it does not scale to large numbers of XPath queries. The technique we present here constructs a single XPush machine for all XPath queries.

Paper Outline Background and problem definition is given in Sec. 2. The XPush machine is formally defined in Sec. 3, and its implementation is discussed in Sec. 4. Several heuristic-based optimization techniques are discussed in Sec. 5. A short theoretical study of the number of states is given in Sec. 6 and an experimental evaluation is in Sec. 7. We conclude in Sec. 8

2. PROBLEM DEFINITION

```
P     ::= /E | //E
E     ::= label | text() | * | @* | .  |
          E/E | E//E | E[Q]
Q     ::= E | E Oprel Const |
          Q and Q | Q or Q | not(Q)
Oprel ::= < | ≤ | > | ≥ | = | ≠
```

Figure 1: The XPath fragment considered in this paper.

XPath The XPath fragment that we consider in this paper contains element and attribute labels, wildcards, child and descendant axis, atomic predicates on data values, and the boolean connectors `and`, `or`, and `not`. A complete grammar is given in Fig. 1. Notice that `not` is supported, and this is important in XML messaging, where sometimes packets need to be forwarded when some condition is not true. Recall that `not` introduces a universal quantification in XPath. For example `/a[not(b/text()=1)]` matches an XML document if all the `b` elements are $\neq 1$.

We treat an XPath expression `P` as above as a boolean filter: an XML document *matches* `P` if and only if `P` selects at least one node when evaluated on the document's root.

An Index for Atomic Predicates The set of atomic predicates included in the XPath fragment is important and affects significantly the techniques described in the paper. We support atomic predicates (Fig. 1) that compare an XPath expression with a constant, using one of $=, <, \leq, >, \geq, \neq$; we assume a fixed, ordered domain of data values $\mathcal{V}$, which we will take to be $\mathcal{V} =$ `int` or $\mathcal{V} =$ `string` in the examples in the paper. The basic operation that we need to be able to support is: given a data value $v \in \mathcal{V}$, find which predicates from among a given collection of atomic predicates are true on v. This is done by constructing an index on the atomic predicates: we call it an *atomic predicate index*. A binary search tree can easily offer this functionality for the atomic predicates in Fig. 1. One may extend the set of atomic predicates, provided that we can still build the index. For example it is possible to support the string oriented predicates `starts-with` and `contains` defined in XPath [9], by adapting Aho and Corasick's dictionary search tree [1]. In general, however, such extensions are non trivial.

XML and SAX Parsers We use a modified SAX parser to read the XML document, which generates the following five types of events:

```
startDocument()
startElement(a)
text(s)
endElement(a)
endDocument()
```

Here `a` is a label from an alphabet Σ of labels, and `s` is a data value from $\mathcal{V}$. To simplify the presentation we treat in this paper attributes similarly to elements, thus the label `a` above may refer either to an element label or to an attribute label. For example, for the XML document below:

```
<a c="3"> <b> 4 </b> </a>
```

gets converted by the parser into the following sequence of events:

```
startDocument()      startElement(b)
startElement(a)      text("4")
startElement(@c)     endElement(b)
text("3")            endElement(a)
endElement(@c)       endDocument()
```

An application provides five call-back functions corresponding to the five event types.

The XML stream processing problem Formally, we are given a set $\mathcal{P} = \{P_1, \ldots, P_n\}$ of XPath filters, where each filter has an associated oid from a set $I = \{o_1, \ldots, o_n\}$, and a stream of XML documents. The problem is to compute, for each document D, the set of oid's corresponding to the XPath expressions that match D.

3. THE XPUSH MACHINE

3.1 De£nition

We define here the *XPush Machine*, which is a modified deterministic pushdown automaton (PDA). The purpose of an XPush machine is to simulate the execution of a workload of XPath filters. When it exhausts the input XML document, the XPush machine returns a set of XPath oids, from a given set $I = \{o_1, \ldots, o_n\}$. The main change from a standard PDA is that the states have two components, a top-down and a bottom-up component, and that the transition functions have been carefully decomposed into several functions exploring only that part of the state that they strictly need (top-down or bottom-up): this results in a more complicated definition with more transition functions, but leads to space savings, as we shall see. A second change is that the XPush machine accepts as inputs SAX events, as defined in Sec. 2, with labels from an alphabet Σ and data values from a domain $\mathcal{V}$. Formally:

DEFINITION 3.1. *An* XPush Machine *is a tuple* $(Q^t, Q^b, q_0^t, q_0^b, t_{push}, t_{value}, t_{pop}, t_{add}^t, t_{add}^b, t_{accept})$ *where:*

- Q^t, Q^b *are called the sets of* top-down *and* bottom-up *states respectively. A* state *is* $q = (q^t, q^b)$, $q^t \in Q^t$, $q^b \in Q^b$, *and* $Q = Q^t \times Q^b$ *denotes the set of states.*
- $(q_0^t, q_0^b) \in Q$ *is the* initial state.
- $t_{push}, t_{value}, t_{pop}, t_{add}^b, t_{add}^t, t_{accept}$ *are partial functions of the following types:*

$$\begin{aligned} t_{push} &: Q^t \times \Sigma \to Q^t \\ t_{value} &: Q^t \times \mathcal{V} \to Q^b \\ t_{pop} &: Q^b \times \Sigma \to Q^b \\ t_{add}^b &: Q^b \times Q^b \to Q^b \\ t_{add}^t &: Q^t \times Q^b \to Q^t \\ t_{accept} &: Q^b \to \mathcal{P}(I) \end{aligned}$$

```
procedure startDocument()
   q^t ← q^t_0     q^b ← q^b_0
   s ← empty stack;
procedure startElement(a)
   push(s, (q^t, q^b));
   q^t ← t_push(q^t, a)
   q^b ← q^b_0
procedure text(str)
   q^b ← t_value(q^t, str)
procedure endElement(a)
   q_aux ← t_pop(q^b, a)
   (q^t_s, q^b_s) ← pop(s);
   q^b ← t^b_add(q^b_s, q_aux)
   q^t ← t^t_add(q^t_s, q_aux)
procedure endDocument()
   return t_accept(q^b);
```

Figure 2: SAX call-back functions implementing the XPush Machine. The current state is denoted (q^t, q^b).

The execution of the XPush Machine is defined in Fig. 2. It maintains a current state $q = (q^t, q^b) \in Q$ and a current stack of states s. Initially $q = (q^t_0, q^b_0)$ and s is the empty stack. The machine reads SAX events from the input stream. On a `startElement(a)` event, it pushes the current state, q on the stack and updates the current state to $(t_{push}(q^t, a), q^b_0)$. On a `text(str)`, it updates the current state to $(q^t, t_{value}(q^t, str))$. On an `endElement(a)` it first computes $q_{aux} = t_{pop}(q^b, a)$, then pops the top state $q_s = (q^t_s, q^b_s)$ from the stack, and updates the current state to $(t^t_{add}(q^t_s, q_{aux}), t^b_{add}(q^b_s, q_{aux}))$. When the input document is exhausted, the machine returns the set of identifiers $t_{accept}(q^b)$. Notice that the XPush machine is deterministic, hence each SAX event is processed in $O(1)$ time.

The six transition functions are implemented by six tables, $T_{push}, T_{value}, T_{pop}, T^b_{add}, T^t_{add}, T_{accept}$. Four of the tables, T_{push}, T_{pop}, T^b_{add}, T^t_{add}, are arrays of hash tables, T_{value} is an array of atomic predicate indexes (see Sec. 2), and T_{accept} is an array of lists of oids. T_{push} and T_{pop} may have entries corresponding to the wildcards $*$ and $@*$, in addition to the labels in Σ, and lookup is modified as follows: if $T_{pop}[q^b][a]$ is undefined then we lookup $T_{pop}[q^b][*]$, or $T_{pop}[q^b][@*]$, depending on whether a is an element or attribute label, and similarly for T_{push}.

Example 3.2 Fig. 3 illustrates an XPush machine that computes the workload {`P1, P2`} in Example 1.1. We will describe in Example 3.3 how this XPush machine was derived from the workload; here we describe only its inner structure. There is a single top-down state, q^t_0, and 22 bottom-up states. T_{pop} is an array indexed by Q^b (hence has 22 entries), and each is a hash table indexed by $\Sigma \cup \{*, @*\}$. T^b_{add} is also an array indexed by Q^b whose entries are hash tables indexed by a certain subset of Q^b, namely those that appear in the T_{pop} table: $\{q_0, q_3, q_4, q_6, q_7, q_{14}, q_{15}\}$. The total number of entries in all hash tables in T^b_{add} is $22 \times 7 = 154$. This is a significant space savings over the traditional representation of a pushdown automaton [15]: there the effects of T_{pop} and T^b_{add} are combined into a single transition table, $T[q^b_s][q^b][a] = T^b_{add}[q^b_s][T_{pop}[q^b][a]]$, and would require, in our example, over 22^2 entries. T_{value} is an atomic predicate index that indexes the two atomic predicates $= 1$ and > 2: it is a binary search tree, which we show as a

XPush machine for workload in Example 1.1:

$Q^t = \{q^t_0\}$, $Q^b = \{q_0, q_1, \ldots, q_{21}\}$

$T_{push}[q^t][*] = q^t, \forall q^t \in Q^t$

$T_{pop}[q^b][\sigma] =$

q^b	$\sigma \in \Sigma \cup \{*, @*\}$				
	a	b	$@c$	$*$	$@*$
q_0				q_0	q_0
q_1		q_3		q_0	q_0
q_2			q_4	q_0	q_0
q_3				q_0	q_0
q_4	q_6			q_0	q_0
q_5	q_7			q_0	q_0
q_6				q_6	q_0
q_7				q_7	q_0
q_8	q_{14}			q_6	q_0
q_9	q_{15}			q_7	q_0
q_{10}				q_6	q_0
q_{11}				q_7	q_0
q_{12}	q_{15}			q_6	q_0
q_{13}	q_{15}			q_7	q_0
q_{14}				q_{14}	q_0
q_{15}				q_{15}	q_0
q_{16}				q_{14}	q_0
q_{17}				q_{15}	q_0
q_{18}				q_{14}	q_0
q_{19}				q_{15}	q_0
q_{20}	q_{15}			q_{14}	q_0
q_{21}				q_{15}	q_0

$T^b_{add}[q^b_s][q^b] =$

q^b_s	q^b						
	q_0	q_3	q_4	q_6	q_7	q_{14}	q_{15}
q_0	q_0	q_3	q_4	q_6	q_7	q_{14}	q_{15}
q_1	q_1						
q_2	q_2						
q_3	q_3	q_3	q_5	q_8	q_9	q_{16}	q_{17}
q_4	q_4	q_5	q_4	q_{10}	q_{11}	q_{18}	q_{19}
q_5	q_5	q_5	q_5	q_{12}	q_{13}	q_{20}	q_{21}
q_6	q_6	q_8	q_{10}	q_6	q_7	q_{14}	q_{15}
q_7	q_7	q_9	q_{11}	q_7	q_7	q_{15}	q_{15}
q_8	q_8	q_8	q_{12}	q_8	q_9	q_{16}	q_{17}
q_9	q_9	q_9	q_{13}	q_9	q_9	q_{17}	q_{17}
q_{10}	q_{10}	q_{12}	q_{10}	q_{10}	q_{11}	q_{18}	q_{19}
q_{11}	q_{11}	q_{13}	q_{11}	q_{11}	q_{11}	q_{19}	q_{19}
q_{12}	q_{12}	q_{12}	q_{12}	q_{12}	q_{13}	q_{20}	q_{21}
q_{13}	q_{13}	q_{13}	q_{13}	q_{13}	q_{13}	q_{21}	q_{21}
q_{14}	q_{14}	q_{16}	q_{18}	q_{14}	q_{15}	q_{14}	q_{15}
q_{15}	q_{15}	q_{17}	q_{19}	q_{15}	q_{15}	q_{15}	q_{15}
q_{16}	q_{16}	q_{16}	q_{20}	q_{16}	q_{17}	q_{16}	q_{17}
q_{17}	q_{17}	q_{17}	q_{21}	q_{17}	q_{17}	q_{17}	q_{17}
q_{18}	q_{18}	q_{20}	q_{18}	q_{18}	q_{19}	q_{18}	q_{19}
q_{19}	q_{19}	q_{21}	q_{19}	q_{19}	q_{19}	q_{19}	q_{19}
q_{20}	q_{20}	q_{20}	q_{20}	q_{20}	q_{21}	q_{20}	q_{21}
q_{21}	q_{21}	q_{21}	q_{21}	q_{21}	q_{21}	q_{21}	q_{21}

$T^t_{add}[q^t][q^b] = q^t, \forall q^b \in Q^b$

$T_{value}[q^t][v] =$

$q^t \in Q^t$	$v \in \mathcal{V}$			
	$(-\infty, 1)$	$\{1\}$	$(1, 2]$	$(2, \infty)$
q^t_0		q_1		q_2

$$T_{accept}[q^b] = \begin{cases} \{o_1, o_2\} & \forall q^b \in \{q_{15}, q_{17}, q_{19}, q_{21}\} \\ \{o_1\} & \forall q^b \in \{q_{14}, q_{16}, q_{18}, q_{20}\} \\ \{o_2\} & \forall q^b \in \{q_7, q_9, q_{11}, q_{13}\} \\ \emptyset & \text{otherwise} \end{cases}$$

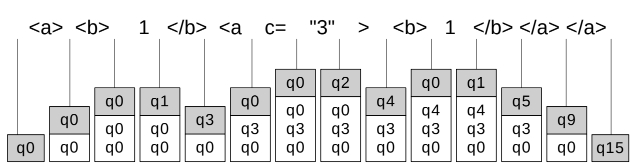


Figure 3: XPush Machine for Ex. 1.1, and a trace of its execution. Missing entries in $T^b_{add}[q^b_s][q^b]$ mean undefined entries. The trace shows only the bottom-up state, since the top-down state is always q^t_0.

table in Fig. 3. The figure also illustrates the execution trace of the XPush machine on the document:

```
<a> <b> 1 </b> <a c="3"> <b> 1 </b> </a> </a>
```

The top-down state component is omitted (it is always q_0^t). The current state is shown at the top, and the stack is shown below. We explain some of the transitions here. The interesting part starts when we encounter the first `text(1)` and the current state becomes $q_1 = T_{value}[q_0^t][1]$; next we see an `endElement(b)` and we compute $T_{pop}[q_1][b] = q_3$, and the current state becomes $T^b_{add}[q_0][q_3] = q_3$; next we see `startElement(a)` followed by `startElement(@c)` (see Sec. 2): each time we push, and set the current state to q_0. Now we see `text(3)` and enter state $q_2 = T_{value}[q_0^t][3]$, followed by `endElement(@c)` when we enter $T^b_{add}[q_0][T_{pop}[q_2][@c]] = T^b_{add}[q_0][q_4] = q_4$. The other transitions should be clear. When the end of the document is encountered the machine is in state q_{15}, and it returns $T_{accept}[q_{15}] = \{o_1, o_2\}$. This is correct, indeed: both `P1` and `P2` match the XML document. Notice that there is no redundant computation in the XPush machine: each SAX event requires only one or two lookups in the hash tables, hence generalizes to $O(1)$ processing time regardless of how many predicates in the workload of XPath expressions it affects.

Bottom-up vs. top-down computation An XPush machine computes bottom-up on the XML tree, listening to `text()` and `endElement()` events: in the example, the top-down phase only builds the stack. The bottom-up style is unavoidable with a deterministic machine, because bottom-up tree automata can be determinized, but top-down automata cannot [19]. We will use, however, the top-down part to express certain optimizations, in Sec. 5.

3.2 Compiling a Set of XPath Filters to an XPush Machine

We show how to compile a set of XPath filters $\mathcal{P} = \{P_1, \ldots, P_n\}$ into a single XPush machine. The method described here is naive, and we will discuss a number of optimizations in the next section: we call the resulting machine the *bottom-up* XPush machine. It is obtained in two steps: (1) convert each of the XPath filters $P_1, \ldots, P_n$ into an Alternating Finite Automaton, AFA, $A_1, \ldots, A_n$, (2) translate the set of all AFAs, $A_1, \ldots, A_n$, to a single XPush machine. We describe each step next.

Step 1: Constructing the Alternating Finite Automata An Alternating Finite Automaton, AFA, [5, 19] is a nondeterministic finite automaton A where each state is labeled with AND, OR, or NOT. Equivalently, the set of states, S, is partitioned into $S = S_{OR} \cup S_{AND} \cup S_{NOT}$. We allow ε transitions and denote $\delta : S \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(S)$ the transition function. A has one initial state, and each terminal state $s \in S$ is labeled with an atomic predicate on data values: we denote with $\pi_s(v)$ the truth value of that predicate on $v \in \mathcal{V}$. For nonterminal states we set $\pi_s(v) = false$. Without loss of generality we impose the following constraints, which help us simplify the presentation: AND and OR states have only ε outgoing transitions, NOT states have a single outgoing transition, and all terminal states are OR states.

Given an XML document tree we say that A *accepts* the document if its initial state *matches* the root node[2]. An OR state, $s \in S_{OR}$ *matches* a node x if x is a data value node and $\pi_s(x)$ is true, or there exists *some* transition $s' \in \delta(s, a)$, and some child y of x labeled a (or $y = x$ when $a = \varepsilon$), such that s' matches y. An AND state, $s \in S_{AND}$, matches x if for *all* transitions $s' \in \delta(s, \varepsilon)$, s' matches x. A NOT state, $s \in S_{NOT}$, matches x if s' does *not* match x, where s' is the unique successor state of s, $\delta(s, \varepsilon) = \{s'\}$.

During the first translation step, we convert every XPath expression $P_1, \ldots, P_n$ into an equivalent AFA, $A_1, \ldots, A_n$. This construction is straightforward: when stripped of the AND, OR, NOT labels the AFAs become precisely the NFAs that have been considered in previous XPath evaluation techniques [11, 13], so it suffices to apply any of those techniques to build the NFAs first, then insert appropriate AND and NOT labels for the `and` and `not` boolean operators in the XPath expressions, and label all other states with OR. If the query does not have a predicate, then we assume a `true` predicate. We will denote with S the union of all states in $A_1, \ldots, A_n$, and $s_1, \ldots, s_n$ the initial state in each of them.

Example 3.3 Fig. 4 illustrates two AFAs, A_1, A_2, corresponding to the two XPath expressions `P1, P2` in our running example. Here $S = \{1, 2, \ldots, 13\}$, $s_1 = 1$, $s_2 = 8$. States 2 and 9 are AND states and each has two ε transitions, and all other states are OR states. Notice that we use the wildcard $*$ in the representation of the AFA (and, similarly, we may use $@*$), and have to take it into consideration when computing δ: e.g. $\delta(5, a) = \{5, 6\}$, $\delta(5, b) = \{5\}$, and $\delta(5, @c) = \emptyset$. To illustrate predicates on data values, we have: $\pi_7(55) = true$, $\pi_7(1) = false$, $\pi_2(v) = false$, $\forall v \in \mathcal{V}$, etc. The states in the AFAs correspond to subqueries in the XPath filters. For example state 3 corresponds to the subquery `[b/text()=1]` of `P1`, while state 2 corresponds to the subquery `[b/text()=1 and .//a[@c>2]]`. One may check that A_1 accepts an XML tree if and only if the XPath filter `P1` is true on that tree, and similarly for A_2 and `P2`.

Step 2: Constructing the bottom-up XPush Machine Finally, in the second step, the bottom-up XPush machine is defined to be $(Q^t, Q^b, q_0^t, q_0^b, t_{push}, t_{value}, t_{pop}, t^t_{add}, t^b_{add}, t_{accept})$ where:

$$
\begin{aligned}
Q^t &= \{q_0^t\} \\
Q^b &= \mathcal{P}(S) \\
q_0^b &= \emptyset \\
t_{push}(q^t, a) &= q^t \\
t_{value}(q^t, v) &= \{s \mid s \in S, \pi_s(v) = true\} \\
t_{pop}(q^b, a) &= \delta^{-1}(eval(q^b), a) \\
t^b_{add}(q^b_s, q^b) &= q^b_s \cup q^b \\
t^t_{add}(q^t, q^b) &= q^t
\end{aligned}
$$

[2]This is one node above the top-most element node, see the formal XPath semantics in [9].

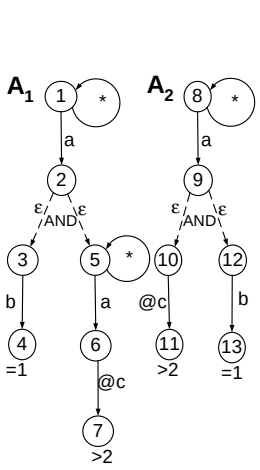
$$t_{accept}(q^b) \quad = \{o_i \mid o_i \in I, s_i \in q^b\}$$

We first explain the two notations introduced in t_{pop}: $\delta^{-1}(q, a)$ denotes $\{s' \mid \delta(s', a) \cap q \neq \emptyset\}$, while $eval(q)$ takes a set of states $q \subseteq S$ and adds to it repeatedly all states that are logically implied by states already present in q. That is: it adds an AND state s to q if *all* its successors $s' \in \delta(s, \varepsilon)$ are in q; it adds an OR state s to q if *some* successor $s' \in \delta(s, \varepsilon)$ is in q; and finally it adds a NOT state s if its successor $s' \in \delta(s, \varepsilon)$ is *not* in q. Multiple iterations are required when boolean connectives are nested in the XPath expressions, and NOT states need to be handled bottom up, in order to process correctly cases like `not(not(Q))`. The details are straightforward and we omit them.

We now explain the bottom-up XPush machine. There is a unique top-down state q_0^t and the bottom-up states are sets of states in A, $\mathcal{P}(S)$. Thus a bottom-up state corresponds to a set of subqueries in the original workload of XPath filters, since each AFA state corresponds precisely to a subquery. The XPush machine keeps track of which AFA states match the current XML node. For a leaf XML node with value v, this set is precisely $t_{value}(q^t, v) = \{s \mid \pi_s(v) = true\}$. To compute these sets after an `endElement(a)`, first find out which AFA states have matched the a node: these are all states in the current q^b, plus all states that are logically implied by it, as computed by $eval(q^b)$; next compute all AFA states that matched a's parent based on these matches: this is $t_{pop}(q^b, a)$; finally union these with the previous states that matched a's parent, retrieved from the stack. Obviously, $t_{accept}(q^b)$ returns the oids of those XPath expressions whose initial states are in q^b.

Pruning the XPush Machines A top-down or bottom-up state in an XPush machine is called *accessible* if there exists some XML document such that the XPush machine will reach that state when run on that document. This definition depends on the class of XML documents considered, e.g. whether a DTD is assumed or not. Assuming for the moment that there is no DTD, one may simply compute the set of accessible states by starting from the initial states and repeatedly applying the transition functions $t_{value}, t_{pop}, t^b_{add}$ as defined above We only retain the accessible states in the bottom-up XPush machine. To make pruning more effective, we will always assume that the XML document has no mixed content. This means that leaf AFA states (which only match atomic values) should not occur in the same set with non-leaf AFA states that correspond to elements (and not attributes): in particular we will not compute $t^b_{add}(q^b_s, q^b)$ if this is violated. This would prohibit the XPush machine to process `<a> 1 <b> 2 </b> </a>`, but will still process `<a c="2"> 1 </a>`.

Example 3.4 Figure 4 illustrates the construction of the bottom-up XPush machine from the two AFAs for `P1` and `P2` in our running example: the transition tables are in Fig. 3. Only accessible states are constructed, by repeatedly applying the definitions of the bottom-up XPush machine. We start by applying the definitions for t_{value}, and obtain[3]: $t_{value}(q_0^t, 1) = \{4, 13\} = q_1$,



q_0	$\emptyset$
q_1	$\{4, 13\}$
q_2	$\{7, 11\}$
q_3	$\{3, 12\}$
q_4	$\{6, 10\}$
q_5	$\{3, 6, 10, 12\}$
q_6	$\{5\}$
q_7	$\{5, 8\}$
q_8	$\{3, 5, 12\}$
q_9	$\{3, 5, 8, 12\}$
q_{10}	$\{5, 6, 10\}$
q_{11}	$\{5, 6, 8, 10\}$
q_{12}	$\{3, 5, 6, 10, 12\}$
q_{13}	$\{3, 5, 6, 8, 10, 12\}$
q_{14}	$\{1, 5\}$
q_{15}	$\{1, 5, 8\}$
q_{16}	$\{1, 3, 5, 12\}$
q_{17}	$\{1, 3, 5, 8, 12\}$
q_{18}	$\{1, 5, 6, 10\}$
q_{19}	$\{1, 5, 6, 8, 10\}$
q_{20}	$\{1, 3, 5, 6, 10, 12\}$
q_{21}	$\{1, 3, 5, 6, 8, 10, 12\}$

Figure 4: AFAs for `P1`, `P2` in Example 1.1, and the states of the corresponding bottom-up XPush machine. The transition tables are shown in Fig. 3.

$t_{value}(q_0^t, x) = \{7, 11\} = q_2$ for $x > 2$, and $t_{value}(q_0^t, x) = \emptyset = q_0$ for all other values of x; in practice we obtain t_{value} by computing the atomic predicate index. Next we apply the function t_{pop}: $t_{pop}(q_1, b) = \{3, 12\} = q_3$, because $eval(q_1) = q_1 = \{4, 13\}$ and, following a b transition backwards, one reaches the states $3, 12$. Similarly $t_{pop}(q_2, @c) = \{6, 10\} = q_4$, and $t_{pop}(q_4, a) = q_6 = \{5\}$. To illustrate addition, we have $t^b_{add}(q_3, q_6) = \{3, 12\} \cup \{5\} = \{3, 5, 12\} = q_8$. To understand how AND states are handled (and similarly NOT, OR states) consider $t_{pop}(q_8, a)$. We first compute $eval(q_8) = eval(\{3, 5, 12\}) = \{2, 3, 5, 12\}$. The meaning is that if states 3 and 5 have matched, then so has state 2. Next we follow a transitions backwards from these states and obtain $t_{pop}(q_8, a) = \{1, 5\} = q_{14}$. All states and transitions in Fig. 3 are obtained this way. Notice that in the T^b_{add} table the entries for q_1 and q_2 are left blank: this is because both q_1 and q_2 contain leaf AFA states, hence, assuming no mixed data in the XML documents, these states cannot match together with any other states. It is also interesting to see how the execution trace in Fig. 3 keeps track of the set of matching AFA states. For example, after reading the first `endElement(b)` the current state is $q_3 = \{3, 12\}$, meaning that the AFA states 3 and 12 have matched so far, corresponding to the common subquery `[b/text()=1]` in both `P1` and `P2`. After reading the second `endElement(b)` the current state is $q_5 = \{3, 6, 10, 12\}$ which means that the following subqueries have matched: `[b/text()=1]`, `[@c > 2 and b/text()=1]`, and `[@c > 2]`. In other words, the states in the bottom-up XPush machine eliminate common subexpressions between filters.

4. IMPLEMENTATION

The XPush machine needs to be computed lazily. We explain here why, and describe the runtime data struc-

[3]The states $q_0, \ldots, q_{21}$ are numbered in no particular order.

tures that we used.

The Lazy XPush Machine We cannot eagerly compute the entire bottom-up XPush machine for a large workload of XPath expressions because it results in exponentially many states. Instead we compute it lazily, at runtime, expanding only those states that are accessible for the given input XML data instance. There is a high penalty associated with computing a state, when it is discovered for the first time. However, we recover this cost later, when the state is reused. We discuss here how the lazy computation helps avoid the exponential state blowup.

By computing the XPush machine lazily we reduce the number of states in three ways. First, we do not construct states that are inconsistent with the DTD. For example, consider n different XPath expressions of the form:

```
/person[name/text()="John"]
/person[name/text()="Smith"]
. . .
```

each looking for a different value for `name`. The eager XPush machine needs 2^n states, one for each subset of names that a person might have. Suppose, however, that the DTD restricts a `person` to have only one `name`: then at most $n+1$ states will be created by the lazy XPush machine. We could have achieved the same effect by pruning the states in the eager XPush machine more carefully, taking the DTD into account, but with the lazy XPush machine this comes for free.

Second, lazy evaluation exploits regularities in the data that are not captured by the DTD. For example, consider several XPath expressions of the form `/person/[phone/text()="v"]`, with different values of the phone number `v`, and assume that the DTD allows a person to have multiple phones. The eager XPush machine needs 2^n states to keep track of all possible sets of phone numbers that a person might have, and clearly the DTD would not help here. But in practice most persons have only one phone, occasionally two, hence the lazy XPush constructs at most $n(n-1)/2$ states, and quite likely only slightly more than n states.

Third, the lazy XPush Machine may simply avoid constructing states that are both allowed by the DTD and consistent with the application domain, but which simply don't occur in a given data set. This idea will be exploited in Theorem 6.2.

Data Structures We have carefully coded the state management to reduce the cost of a state computation. An XPush state is represented as an sorted array of AFA states, plus a 32 bit signature (hash value) of these AFA states. All the XPush states that have been discovered, are stored in a hash table indexed by their signature. All operations in the definition of the XPush machine are implemented such that the arrays of AFA states are never required to be sorted explicitly. For example to compute $\delta^{-1}(q,a)$ needed in $t_{pop}(q^b,a)$ we maintain backpointers for each AFA state[4] and simply traverse the sorted array q once, follow the back pointers, and obtain $\delta^{-1}(q,a)$ already in sorted order (because the sort keys in the AFA states are generated based on depth-first traversal). For $q = eval(q^b)$, we do a number of iterations equal to the deepest nesting of boolean operators in the XPath workload. Each iteration requires one complete traversal of q^b plus a merge between the sorted q^b and the sorted set of new states that need to be inserted in q^b. We omit the details. Finally, $t^b_{add}(q^b_s, q^b)$ implies a merge-join of two sorted arrays.

State Precomputation To speed up the lazy XPush machine at runtime we precompute eagerly some of its states and transition table entries. In the bottom-up XPush machine discussed so far, we only compute the atomic predicate index and all the XPush states of the form $t_{value}(q^t_0, v)$.

5. OPTIMIZATIONS

The heuristic-based optimizations described here have three goals: reduce the number of states in the XPush machine (thus saving space), reduce the number of AFA states per XPush state (thus speeding up runtime state construction), and precompute some XPush state before processing any XML inputs.

Top-down Pruning The bottom-up XPush machine may follow false leads that will be invalidated only later, and this ultimately leads to unnecessary states. To illustrate this point, assume that one or more `<c>` element may occur in any of `<e1>`, `<e2>`, ..., `<en>`. Let us also assume that the workload consists of queries of the form `/ei[c/text()="ci"]`, $i = 1, \ldots, n$. After reading the XML fragment:

`<e1>` ... `<c> ci1 </c>` ... `<c> cij </c>` ... `</e1>`

an XPush state will be created containing AFA states from all the filters that look for a `c` element with text value c_{i_k}, where $k \in \{1, \ldots, j\}$, ignoring the fact that in the document, the `c` elements occur under an e_1 element. Clearly, those predicates that do not occur under an e_1 are false leads and will be invalidated later, but they can create an exponential increase in the number of XPush states because any subset of the predicates `c/text()="ci"` can be true, depending upon which of the c_i's appear in the document. The top-down pruning optimization eliminates the wrong XPush states, by keeping track of the enabled branches in the top-down component of the state, and starting the bottom-up computations only at the enabled branches. The changes to the definitions in Sec. 3.2 are:

$$
\begin{aligned}
Q^t &= \mathcal{P}(S) \\
q^t_0 &= \{s_1, \ldots, s_n\} \\
t_{push}(q^t, a) &= close(\{\delta(s,a) \mid s \in q^t\}) \\
t_{value}(q^t, v) &= \{s \mid s \in q^t, \pi_s(v) = true\} \\
t^t_{add}(q^t, q^b) &= q^t
\end{aligned}
$$

$$
\begin{aligned}
close(q^t) &= \texttt{repeat } q^t := q^t \cup \{\delta(s,\varepsilon) \mid s \in q^t\} \\
&\quad\ \texttt{until no-more-change} \\
&\quad\ \texttt{return } q^t
\end{aligned}
$$

Order Optimization This optimization is based on order information between elements extracted from the DTD. To illustrate consider the XPath expression

```
/person[name/text()="Smith" and age/text()="33"
        and phone/text()="5551234"]
```

[4]Each state s has either one or two incoming transitions.

and assume that, according to the DTD, `name`, `age`, and `phone` must appear in this order in XML data. The lazy XPush machine still has 2^3 states, corresponding to all subsets of the predicates: for example the XML document `<person> <name> John </name> <age> 33 </age> ...` activates the `age` predicate but not the `name` predicate. Similarly, each of the 2^3 subsets of predicates can be activated by some XML document. To prevent that, we use the DTD to define a partial order on elements and attributes: $a \prec b$ if a must precede b whenever a and b are siblings. Every attribute always precedes every element, and additional order information between elements can be extracted from the DTD, when available. Next, we extend this order relation to AFA states: $s \prec s'$ if s and s' are both children of the same AND state, and every outgoing label from s precedes every outgoing label from s': if either s or s' have $*$ transition, then $s \not\prec s'$ and $s' \not\prec s$. Using this relation we make the following changes in the definition of the XPush machine, where $prec(s) = \{s' \mid s' \prec s\}$:

$$t^b_{add}(q^b_s, q^b) \quad = \quad q^b_s \cup \{s \mid s \in q^b, prec(s) \subseteq q^b_s\}$$

Early Notification Optimization Let s be the first branching AFA state in some alternating automaton A: for example in Fig. 4 the first branching state in A_1 is 2, and in A_2 is 9. In early notification we stop the evaluation of the AFA early, once this state has matched some node in the XML document. For this technique to be correct we must turn on *top-down pruning*. This ensures correctness for workloads that do not contain //: to handle // correctly we need to intersect the bottom-up state with the top-down state after every pop operation (formally, this requires a minor change in the definition of the XPush machine). This optimization can be extremely effective when s occurs deeply: for example in the case of a linear XPath expression, s is the (unique) leaf state. It follows that during the entire bottom-up phase of the evaluation, A's states are no longer included in the XPush states. In an extreme case, when all XPath expressions are linear, the XPush machine with this optimization degenerates to a top-down automaton, which has been shown in [13] to be very effective for processing linear XPath expressions.

Training the XPush Machine Given a workload of XPath queries we generate the *training data* for that workload as follows. We generate one XML document tree D for every XPath query tree P: atomic predicates are replaced with values that satisfy them, and label constants are replaced with elements or attributes. Wildcards $*$ and // are expanded using the DTD, and boolean connectors are simply ignored. For example, the query:

```
/a[(b/text()=3 and @c=4) or d/text()=5]
```

will result into the following training document:

```
<a c="4"> <b> 3 </b> <d> 5 </d> </a>
```

The DTD is also consulted to generate the elements in the right order: in the example above, `b` and `d` may be swapped if the DTD requires `d` to occur before `b`. All such generated documents are concatenated and the result is called *training data*. The lazy XPush machine is run on the XML training data first, which determines it to compute some of its states; then, the "warmed-up" machine is run on the real XML data. Now, the states that have been already computed by the training data can be reused, which results in increased throughput.

6. A THEORETICAL ANALYSIS

We have observed empirically (Sec. 7) that the number of states in the lazy XPush machine is not exponential. Here we try to justify this observation analytically, and give two explanations. The first is that there are relationships between AFA states that make certain sets of AFA states inaccessible, and the second is that low selectivities of the atomic predicates reduces the number of expected states in the lazy XPush machine.

To explain the first we borrow techniques developed for tree pattern matching in [14]. Given two AFA states $s, s' \in S$, we say that s *subsumes* s' if, for every node in an XML document, if s matches that node then so does s': we denote $s \Rightarrow s'$. We say that s and s' are *inconsistent* if they never match the same node in an XML document; we denote $s \mid s'$. Finally we say that s, s' are *independent* if neither $s \Rightarrow s'$, nor $s' \Rightarrow s$, nor $s \mid s'$. The *independence graph* is defined to have all AFA states as vertices, and as edges all pairs (s, s') of independent states. We have the following result, an adaptation of [14]:

THEOREM 6.1. *The number of accessible states in the XPush machine is no larger than the number of cliques in the independence graph.*

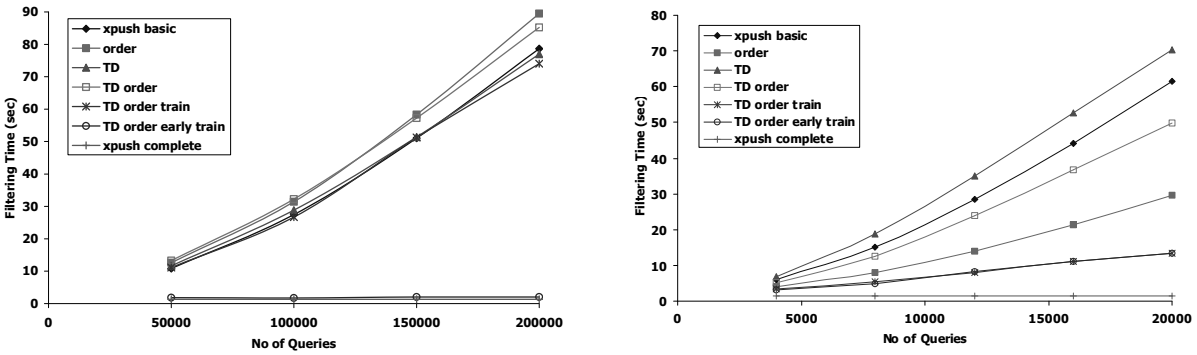
PROOF. Given an accessible state q^b, we associate to it a clique by removing all AFA states s s.t. there exists $s' \in q^b$ with $s' \Rightarrow s$. It is easy to see that two distinct accessible XPush states will result in two distinct cliques, proving the theorem. □

For example, in Fig. 4, we have $8 \Rightarrow 5$: as a consequence the set $\{1, 8\}$ is not a state in the XPush machine. Also, $4 \Leftrightarrow 13$, and $4 \mid s$ for every state $s \neq 13$, since we assume that the XML documents have no mixed content. As a consequence the only XPush state containing 4 is $q_1 = \{4, 13\}$.

The second factor is determined by data value predicates with low selectivities. The intuition is that, in order for k AFA states to form a state in the XPush machine, all their predicates must be true on the *same* input XML document. The probability of this happening in a given set of XML documents is a function of those predicates' selectivities, and decreases exponentially with k. To make this argument formal, we consider flat workloads. Define a *flat XPath workload* to be a set of n XPath queries where each query is of the form:

$$/a[b_1/text() = v_1 \texttt{ and } \ldots \texttt{ and } b_k/text() = v_k]$$

That is, each query starts with $/a$ (the same a in all queries), and has some number of atomic predicates of the form $[b_i/text() = v_i]$; predicates may be shared between XPath expressions, and a given tag b_i may occur in different predicates with different constants. Assume that we run the lazy XPush machine on a stream of N XML documents, and want to analyze the expected number of states created. We consider both the case without order optimization, and with order optimization. To simplify the problem, assume that every atomic



(a) 1.15 Predicates/Query

(b) 10.45 Predicates/Query

Figure 5: Filtering Time

predicate has the same probability σ of being true on a given XML document; σ is the predicate's selectivity.

In the case without order optimization, let us denote with m the total number of distinct atomic predicates in the workload: hence there are at most 2^m possible states in the XPush machine. For the second case, we will assume that there is a total order imposed by the DTD, $b_1 \prec b_2 \prec \ldots$, and, furthermore, that every query has exactly k atomic predicates. We have:

THEOREM 6.2. *Consider the execution of a lazy XPush machine for flat XPath workload with n queries over an XML stream of N documents. Assume that all atomic predicates have the same selectivity $\sigma \in (0,1)$, and $\sigma \ll 1/N$. Then: (1) if the XPush machine does not implement order optimization then the expected number of states is at most $1 + Nm\sigma$, where m is the total number of atomic predicates in the workload. (2) if the XPush machine does order optimization, then the expected number of states is at most:*

$$N(\frac{1-\sigma^{k+1}}{1-\sigma})^n$$

where k the number of atomic predicates per query and assumed to be fixed.

PROOF. (1) Fix an XML document D and a set of k atomic predicates. The probability that exactly these predicates are satisfied by D is $\sigma^k(1-\sigma)^{m-k}$; if this happens, then D contributes with at most k states in the lazy XPush machine, as the k atomic predicates are satisfied in some order (we will count the empty set separately). Thus, the expected number of non-empty sets of predicates that will become states in the lazy XPush machine while processing one XML document is $\sum_{k=0,m}\binom{m}{k}k\sigma^k(1-\sigma)^{m-k} = \sigma m$. The theorem follows by adding up over all N XML documents, then accounting for the empty state. (2) A state in the lazy XPush machine with order optimization is uniquely determined by n numbers, $r_1, \ldots, r_n \in \{0, 1, \ldots, k\}$. The number r_i indicates that the first r_i predicates in query i are true and the remaining are false; clearly the probability of this happening is σ^{r_i}. Thus, the expected number of states in the XPush machine is:

$$N \sum_{0 \le r_1, \ldots, r_n \le k} \sigma^{r_1} \ldots \sigma^{r_k} = N(\frac{1-\sigma^{k+1}}{1-\sigma})^n$$

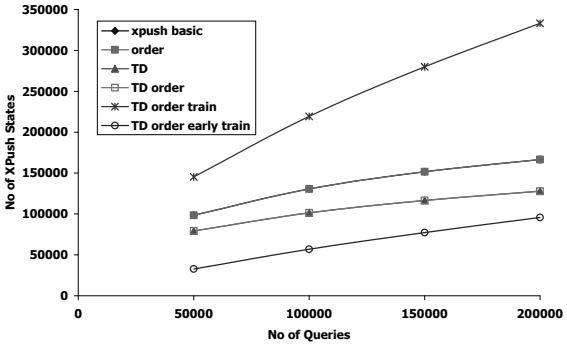
□

This analysis reveals three things. First, as the selectivity σ decreases, there are fewer expected states. Second the number of states increases linearly with the number of XML documents N: we need some form of memory management in order to process infinite streams. Third, assuming that the queries consist of conjunction of predicates, when we apply the order optimization, then the number of states decreases if we increase the number of branches (i.e. atomic predicates) per query, k, while keeping the total number of branches kn constant, i.e. XPush machine with order optimization will have fewer states for workloads with more branches per query.

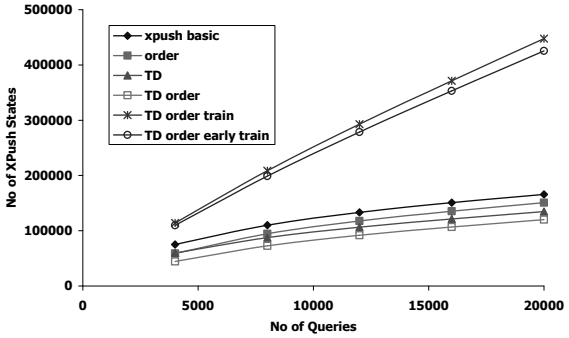
7. EXPERIMENTS

We evaluated the XPush machine addressing the following questions. How effective can the XPush machine be ? What are the memory requirements of the lazy XPush machine ? How close is the performance of the lazy XPush machine to its ideal performance, when it doesn't have to compute any more states at runtime ? And, finally, how effective are the optimization techniques ?

Experimental setting We run experiments on two real data sets: Protein (`http://pir.georgetown.edu` and NASA (`http://xml.gsfc.nasa.gov`), but report results only for the Protein dataset, for lack of space (the results for NASA were similar). All the experiments used a 9.12 MB XML fragment of the Protein dataset, unless stated otherwise. Protein dataset has a non-recursive DTD and the maximum depth of the document is 7. NASA dataset has a recursive DTD, with maximum document depth equal to 8. We generated synthetic XPath queries using a modified version of the generator in [11]: we modified it to generate bushy query trees, rather than left-linear trees, and modified it to generate atomic predicates using data values from the given data instance, ensuring that each predicate is true on at least some XML document. Thus the selectivity of the atomic predicates depends on the data set for which we generated the queries, and is not the same in all experiments. The probability of wildcard and descendant axis were both set to zero for the set of experiments for which we report the numbers here.

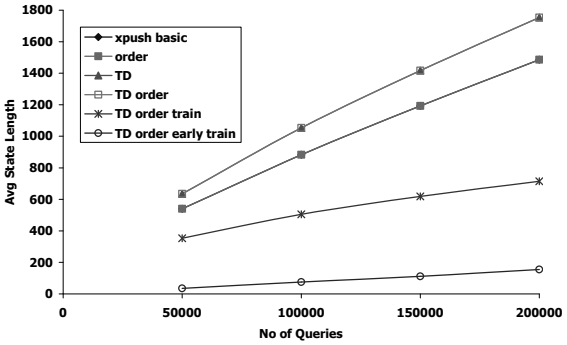


(a) 1.15 Predicates/Query

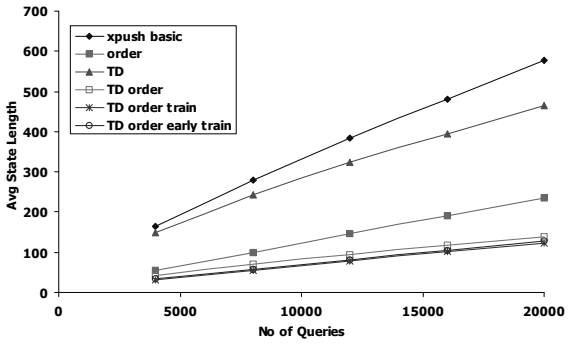


(b) 10.45 Predicates/Query

Figure 6: Number of XPush States



(a) 1.15 Predicates/Query



(b) 10.45 Predicates/Query

Figure 7: Average XPush State Size

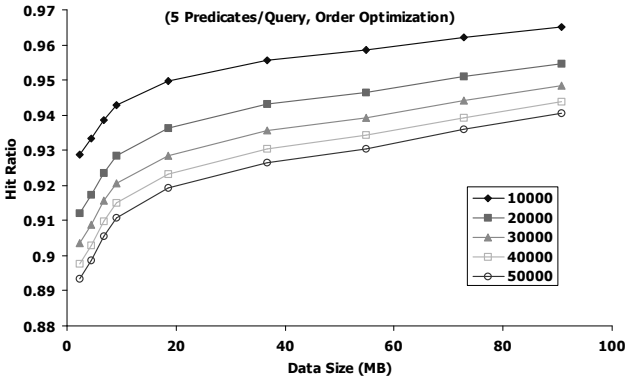
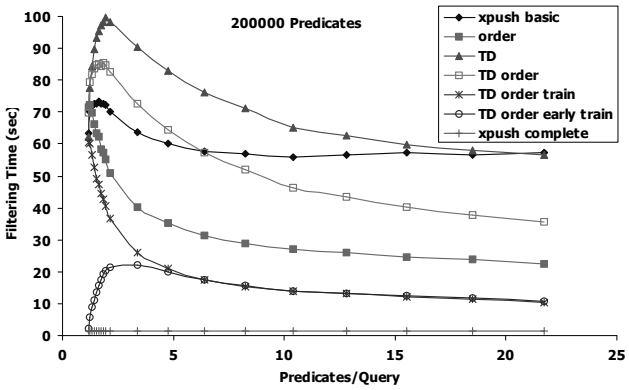


Figure 8: Hit Ratio

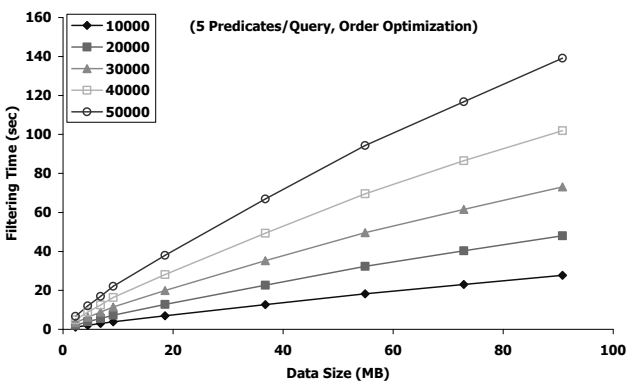
All experiments are on a Pentium III, 700MHz machine, with 2GB of memory, running RedHat Linux 7.1.

Effectiveness of the XPush machine In Fig. 5(a) we show the filtering time (which includes the parsing time also) for workloads with 50000 to 200000 queries with about 1.15 predicates per query. Combination of the four optimization techniques results in filtering time of around 2.1 seconds, no matter how many queries are there in the workload. To measure the effectiveness of the "completed" XPush machine we ran it twice over that data, and report only the time to process it the second time: this took 1.2s including parsing time, and should be compared with the time taken by Apache's parser to parse that data set, 2.53s (we used a faster parser in the XPush machine, which took 1s to parse 9.12MB data). This confirms that the XPush machine can be very efficient, and the only significant cost is that associated to lazy state computation. In Fig. 5(b), we report the same numbers for workloads with 10.45 predicates per query on an average. Here the combination of top down, order and training optimizations gives the best results. Early notification does not result in any further reduction in filtering time. This can be further seen in Fig. 9(a) where we see that for the workloads containing more than 5 predicates per query, the plot for TD-order-train coincides with the plot for TD-order-early-train.

Runtime memory requirements Fig. 6(a) and 7(a) show the number of XPush states and the average size of each state. In Fig. 6(a), for a workload of 200000 queries, the number of states for the basic XPush machine was around 150000, far from the worst case, which is exponential in the number of atomic predicates. This graph also shows the effect of the various optimizations discussed. All the optimizations result in decrease in number of states. The only exception is TD-order-train, where the number of states actually increases as compared to basic XPush. This is because of the additional states created during the training phase, which are never used later. The effect of the optimizations is even more dramatic in Fig. 7(a), where we show the average size of each state. Combining these two results in a slightly above linear increase of the total memory requirement

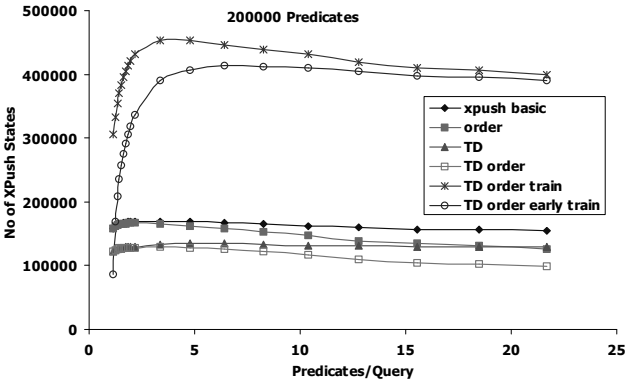


(a) Varying No of Predicates/Query

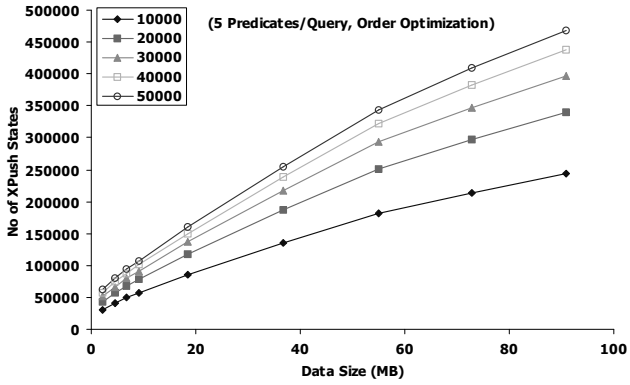


(b) Varying Data Size

Figure 9: Filtering Time



(a) Varying No of Predicates/Query



(b) Varying Data Size

Figure 10: Number of XPush States

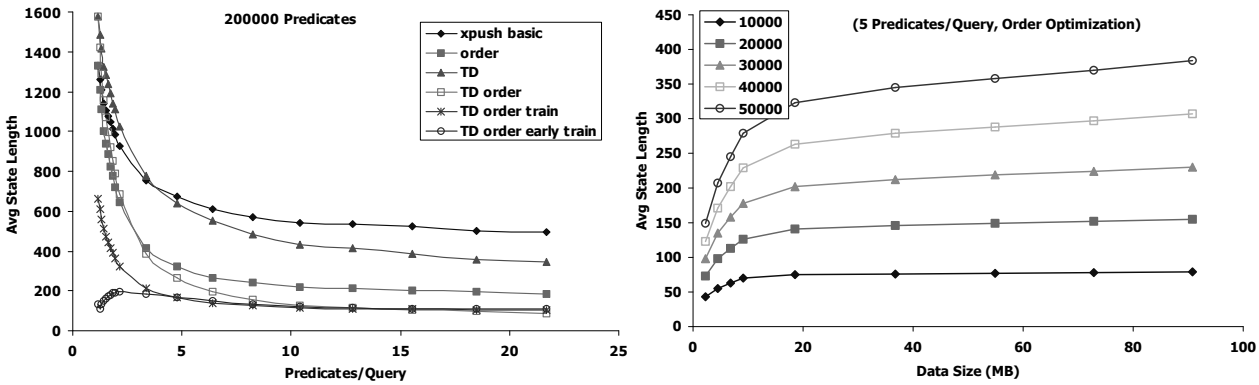
as a function of the workload. Fig. 6(b) and 7(b) show the same results for the case of 10.45 predicates per query. The graphs in Fig. 10(a) and 11(a) show the same measures, but now we increased the number of predicates per query while keeping the total number of atomic predicates fixed at 200000. As we predicted in Theorem 6.2, the number of states decreased. As a consequence, the running time for these queries, shown in Fig. 9(a) decreases too. Finally, Fig. 10(b) and 11(b) show the same measurements as a function of the data size, showing a slightly sub-linear increase.

Hit ratio One can think of the XPush machine as a cache: states remember configurations that we have seen before, and can be deleted when we run out of memory and recomputed later. In Fig. 8, we show the hit ratio, i.e. the number of successful lookups in the XPush tables versus the total number of lookups. We see that, after 20MB of data has been processed the hit ratio is well above 90%, then increases to over 93%.

Effectiveness of the optimizations This can be best seen in Fig. 5(a) and (b). Each optimization improves performance, by essentially reducing the number of states and their size. In Fig. 5(a), the exceptions to this are order optimization and top-down with order optimization. This is because with only 1.15 predicates per query, very few queries have more than one predicate. So, the benefit obtained from order optimization is very little, and it doesn't offset its overhead. In Fig. 5(b), the only exception is the top-down optimization in isolation: the explanation here is that we can no longer precompute the atomic predicate index, and doing it at runtime affects performance. However, when coupled with training, the top-down optimization is very effective: this is because the training data generates all predicate indexes in the XPush machine.

8. CONCLUSION AND FUTURE WORK

Our goal is to process efficiently large numbers of XPath expressions with many predicates per query, on a stream of XML data. We have described a new push-down machine, called XPush, that can express such workloads. If fully computed, the XPush machine runs extremely fast on the XML stream, since it processes each SAX event in $O(1)$ time, independent of the query workload: in our experiments it ran twice as fast as the Apache parser. However, in most practical applications the XPush machine cannot be precomputed but needs to be computed lazily, at runtime. We have shown experimentally that by computing it lazily the memory requirements of the XPush machine are manageable. We have also shown that the cost paid for computing the states at runtime is recovered later: the hit ratio in our experiments was well over 90%, even over 93% after processing large amounts of data. Finally, we have shown



(a) Varying No of Predicates/Query (b) Varying Data Size

Figure 11: Average XPush State Size

that a combination of optimizations improved significantly the runtime performance of the XPush machine.

Currently, we do not support updates to the XPath workload, but they can be supported in one of the two ways. The first is brute force: reset the lazy XPush machine periodically and re-start it on the new XPath workload, with an initially empty set of states and tables. This is equivalent to flushing an entire cache. The second method applies to insertions of new XPath filters only. To insert a new XPath filter, build a new XPush machine on top of the old XPush machine and the new XPath expression. The states in the new XPush machine are very small: they contain at most one state from the old XPush machine and a few AFA states from the new XPath filter.

Acknowledgments

We thank Pradeep Shenoy, Tami Tamir and the anonymous reviewers for their comments. Suciu was partially supported by the NSF CAREER Grant 0092955, NSF Grant IIS-0140493, a gift from Microsoft, and a Sloan Fellowship. Gupta was partially supported by the NSF CAREER Grant 0092955.

9. REFERENCES

[1] A. Aho and M. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18:333–340, 1975.

[2] M. Altinel and M. Franklin. Efficient filtering of XML documents for selective dissemination. In *Proceedings of VLDB*, 2000.

[3] J. Cai, R. Paige, and R. Tarjan. More efficient bottom-up multi-pattern matching in trees. *TCS*, 106(1):21–60, 1992.

[4] C. Chan, P. Felber, M. Garofalakis, and R. Rastogi. Efficient filtering of XML documents with XPath expressions. In *Proceedings of ICDE*, 2002.

[5] A. Chandra, D. Kozen, and L. Stockmeyer. Alternation. In *Journal of the ACM*, pages 115–133, January 1981.

[6] C. Chauve. Tree pattern matching for linear static terms. In *Proceedings of the International Symposium on String Processing and Information Retrieval*, volume 2476 of *Lecture Notes in Computer Science*, pages 160–169. Springer, 2002.

[7] J. Chen, D. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: a scalable continuous query system for internet databases. In *Proceedings of SIGMOD*, 2000.

[8] J. Chen, D. J. DeWitt, and J. F. Naughton. Design and evaluation of alternative selection placement strategies in optimizing continuous queries. In *Proceedings of ICDE*, 2002.

[9] J. Clark. XML path language (XPath), 1999. `http://www.w3.org/TR/xpath`.

[10] R. Cole, R. Hariharan, and P. Indyk. Tree pattern matching and subset matching in deterministic O(n log3 n)-time. In *SODA*, pages 245–254, 1999.

[11] Y. Diao, P. Fischer, M. Franklin, and R. To. Yfilter: Efficient and scalable filtering of XML documents. In *Proceedings of ICDE*, 2002.

[12] G. Gottlob, C. Koch, and R. Pichler. Efficient algorithm for processing XPath queries. In *Proceedings of VLDB*, 2002.

[13] T. J. Green, G. Miklau, M. Onizuka, and D. Suciu. Processing XML streams with deterministic automata. In *Proceedings of ICDT*, 2003.

[14] C. M. Hoffmann and M. J. O'Donnell. Pattern matching in trees. *JACM*, 29(1):68–95, 1982.

[15] J. Hopcroft and J. Ullman. *Introduction to automata theory, languages, and computation*. Addison-Wesley, 1979.

[16] P. Kilpelainen and H. Mannila. Ordered and unordered tree inclusion. *SIAM Journal of Computing*, 24(2):340–356, 1995.

[17] B. Nguyen, S. Abiteboul, G. Cobena, and M. Preda. Monitoring XML data on the web. In *Proceedings of SIGMOD*, 2001.

[18] D. Olteanu, T. Kiesling, and F. Bry. An evaluation of regular path expressions with qualifiers against XML streams. In *Proceedings of ICDE*, 2003.

[19] G. Rozenberg and A. Salomaa. *Handbook of Formal Languages*. Springer Verlag, 1997.

[20] A. Snoeren, K. Conley, and D. Gifford. Mesh-based content routing using XML. In *Proceedings of the 18th Symposium on Operating Systems Principles*, 2001.

[21] M. Thorup. Efficient preprocessing of simple binary pattern forests. *Journal of Algorithms*, 20(3):602–612, 1996.