1. INTRODUCTION

We address the following XML stream processing problem. Given a large collection of XPath queries, evaluate the queries on an incoming stream of XML documents. Each query 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 several applications, such as XML packet routing [19], selective dissemination of information [2], and notification systems [16]. There is an industry trend towards the development of XML messaging systems for business applications, which has already spawned a flurry of startup companies developing XML routing systems\(^1\). All such systems have, at their core, an XML stream processing engine.

The main challenge 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, 10] have addressed this problem by identifying and eliminating common subexpressions 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 this is precisely the most computationally intensive part for workloads where queries consists of multiple predicates, which are typical in XML routing systems [19].

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

\[
P_1 = //a[b/text()=1 \text{ and } @c>2]\\n_P_2 = //a[@c>2 \text{ and } b/text()=1]
\]

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 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 many (say 5-20) predicates, such common predicates are frequent, and keeping track of them separately for each query degrades the performance significantly.

We present in this paper a new approach to processing XPath expressions on streaming XML data that eliminates all common subexpressions in both the structure navigation and the predicate evaluation part of a workload of XPath queries. 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 approach consists of translating the entire 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 to save space in the the transition tables, and call the resulting formalism an XPush Machine. The first contribution of this paper consists of defining the XPush machine, and describing a method for translating a workload of XPath filters into a single XPush machine. The number of states in the XPush machine is exponential in the size of the workload, hence it is not possible to precompute it: instead, we compute it lazily, that is, its states are expanded at runtime, and only those states are created that are necessary to process the given data instance. We pay a relatively high cost in computing a state at runtime, but recover that cost when the state is reused. Our second contribution consists in analyzing both empirically and theoretically the number of states in the lazy XPush machine: we found it to be of about the same order of magnitude as the total number of atomic predicates in the workload, much less than the worst case exponential number. This is significant because it makes it possible to deploy an XPush machine run-
ning tens of thousands of complex XPath queries on today’s typical systems. Our third contribution consists of several optimization techniques for the XPush machine that reduce the memory requirements and/or speed up the lazy computation of its states. One class of optimizations exploits XML specific constraints: for example the fact that all attributes precede all subelements, that certain elements may be ordered by the DTD, etc. The other type of optimization consists of training the XPush machine before running it on the actual data: this pre-computes some of the states in the lazy XPush machine that are likely to be needed at runtime and this speeds up the machine at runtime. The training data set is derived automatically from the XPath workload, by generating one XML document for each XPath expression. Finally, our fourth contribution consists of an experimental evaluation of the XPush machine, using real XML data and synthetic XPath queries.

Related Work In a seminal paper Hoffman and O’Donnell [14] introduce the tree pattern matching problem, 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, 20, 9, 6]. 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 XML stream processing we also want to identify all common subpatterns (predicates), but their technique, and its subsequent improvements, do not apply directly, for several reasons: the tree patterns are ordered, have no wildcards (*, //), and the exponential size data structure is prohibitive for large workloads of XPath queries. Our 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 \( O(n \log^2 n) \), where \( n \) is the combined subject plus pattern size [9]. 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; XTrie [4] uses a trie to detect common paths in the queries; YFilter [10] 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 [16], 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

### Figure 1: The XPath fragment considered in this paper.

A technique for evaluating XPath expressions using stack machines is described in [17]. 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.

### 2. PROBLEM DEFINITION

**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 packet routing applications, where packets need sometimes 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 \( \neq 1 \). More interestingly, /a[not(b/text()=1)] matches a document if all the \( b \) elements are \( = 1 \).

We treat an XPath expression \( P \) as above as a boolean filter: an XML document matches \( P \) 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 \( =, \neq, <, \leq, >, \geq \); we assume a fixed, ordered domain of data values \( V \), which we will take to be \( V = \text{int} \) or \( V = \text{string} \) in the examples in the paper. The basic operation that we need to be able to support is: given a data value \( v \in V \), 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

\[
\begin{align*}
P & ::= /E \mid //E \\
E & ::= \text{label} \mid \text{text()} \mid * \mid \Omega \mid . \\
& \mid E / E \mid E / \Omega \\
Q & ::= E \mid E \ \text{Oprel} \ \text{Const} \\
& \mid Q \land Q \mid Q \lor Q \mid \text{not}(Q) \\
\text{Oprel} & ::= \langle \mid \leq \mid \rangle \mid \geq \mid = \mid \neq
\end{align*}
\]
in XPath [8], 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 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 `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 b="3"> <c> 4 </c> </a>
```

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

- `startDocument()`
- `startElement(a)`
- `startElement(\@b)`
- `text("3")`
- `endElement(\@b)`
- `startElement(c)`
- `text("4")`
- `endElement(c)`
- `endElement(a)`
- `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 `P = \{P_1, \ldots, P_n\}` of XPath filters, where each filter has an associated oid from a set `I = \{\alpha_1, \ldots, \alpha_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 Definition

The machine we introduce here, called the **XPush Machine**, is a modified deterministic pushdown automaton (PDA). It performs a `push` on every `startElement`, a `pop` on every `endElement`, and keeps the stack constant on a `text` event: in the formal language terminology it is input driven [11]. We modify the standard definition of a PDA [15] as follows. The inputs to the XPush machine are SAX events with labels from an alphabet `Σ` and data values from a domain `V`; a state has two parts, the top-down and the bottom-up component; and for a pop the machine needs to lookup three tables instead of one. When the XML input ends, the machine returns a set of XPath oids from a set `I = \{\alpha_1, \ldots, \alpha_n\}`. The formal definition is:

**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}^t, t_{add}^b, t_{accept}` are partial functions of the following types:
  - `t_{push} : Q^t \times \Sigma \rightarrow Q^t`
  - `t_{value} : Q^t \times V \rightarrow Q^b`
  - `t_{pop} : Q^b \rightarrow Q^b`
  - `t_{add}^t : Q^b \times \Sigma \rightarrow Q^b`
  - `t_{add}^b : Q^t \times Q^b \rightarrow Q^b`
  - `t_{accept} : Q^b \rightarrow P(I)`

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_0^t, q_0^b)` and `s` is the empty stack. The machine reads SAX events from the input stream and processes them as follows. 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_0^b)`. On a `text(s)` event, it updates the current state to `(q^t, t_{value}(q^t, s))`. On an `endElement(a)` it first computes `q_{aux} = t_{pop}(q^t, a)`, then pops the top state `q_s = (q_s^t, q_s^b)` from the stack, and updates the current state to `(t_{add}^t(q_s^t, q_{aux}), t_{add}^b(q_s^t, q_{aux}))`. When the input document is exhausted, the machine returns the set of identifiers `t_{accept}(q_b)`, which are used to notify the application. In summary, the top-down state `q^t` is updated when the machine moves both top-down and bottom-up in the XML tree, while the bottom up state is updated only when it moves bottom-up. 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_{add}^t, T_{add}^b, T_{accept}`. Four of the tables `T_{push}, T_{pop}, T_{add}, T_{add}^b, T_{accept}` 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^t][a][@*]` is undefined then we lookup `T_{pop}[q^t][a]`, or `T_{pop}[q^t][@*]`, 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 `\{P_1, P_2\}` in Example 1.1. There
### Procedure startDocument

- $q' \leftarrow q^0 \quad q^0 \leftarrow q_0$
- $s \leftarrow$ empty stack

### Procedure startElement

- \( \text{push}(s, (q', q^b)) \);
- \( q' \leftarrow t_{\text{push}}(q', a) \)
- \( q^b \leftarrow q_0 \)

### Procedure text

- \( q^b \leftarrow t_{\text{value}}(q', s) \)

### Procedure endElement

- \( q_{\text{aux}} \leftarrow t_{\text{pop}}(q^b, a) \)
- \( (q^b, q^{\text{pop}}) \leftarrow \text{pop}(s) \)
- \( q' \leftarrow t_{\text{add}}(q^b, q_{\text{aux}}) \)
- \( q' \leftarrow t_{\text{add}}(q^b, q_{\text{aux}}) \)

### Procedure endDocument

- return \( t_{\text{accept}}(q^b) \)

---

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

is a single top-down state, \(q^a\), and 26 bottom-up states. \(T_{\text{pop}}\) is an array indexed by \(Q^b\) (hence has 26 entries), and each is a hash table indexed by \(\Sigma \cup \{\ast\}\). Notice that most entries are undefined, meaning that the hash tables are small: the total number of entries in all hash tables is 33. \(T_{\text{add}}\) is also an array indexed by \(Q^b\) whose entries are hash tables indexed by a certain subset of \(Q^a\), namely on those sets that appear in the \(T_{\text{pop}}\) table: \(\{q_1, q_2, q_3, q_5, q_7, q_8, q_9, q_{10}\}\). All entries are defined: to reduce clutter we omit in Fig. 3 those entries for which \(T_{\text{add}}[q_0][q^b] = q^a\). Thus, the total number of entries in all hash tables in \(T_{\text{add}}\) is 26 \(\times\) 7 = 182. This is a significant space savings over the traditional representation of a pushdown automaton [15]: there the effects of \(T_{\text{pop}}\) and \(T_{\text{add}}\) are combined into a single transition table, \(T_{q_0}[q^a][a] = T_{\text{add}}[q_0][T_{\text{pop}}[q^b][a]]\), and would require, in our example, over 26^2 entries. \(T_{\text{value}}\) 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 table in Fig. 3. The figure also illustrates the execution trace of the XPush Machine on the document:

\[\text{<a} \text{<b} \text{1</b> <a c="3"> <b 1</b> </a> </a}\]

The top-down state component is omitted (it is always \(q^a\)). 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{text} and the current state becomes \(q_0 \leftarrow t_{\text{value}}(q')[1]\); next we see an \text{endElement} and we compute \(T_{\text{pop}}[q_0][b] = q_5\), and the current state becomes \(T_{\text{add}}[q_0][q_5] = q_5\); next we see \text{startElement} followed by \text{startElement} (Sec. 2): each time we push, and set the current state to \(q_0\). Now we see

---

**Figure 3: An XPush Machine for the running example, Ex. 1.1, and a trace of its execution.** Missing entries in \(T_{\text{add}}[q_0][q^b]\) mean \(q_0\). The trace shows only the bottom-up state, since the top-down state is always \(q^a\).
text(3) and enter state \( q_4 = T_{value}[q_1][3] \), followed by
\( \text{endElement}(\text{c}) \) when we enter \( T_{add}[q_0][T_{pop}[q_4][\text{c}]] = T_{add}[q_0][q_4] = q_8 \). The other transitions should be clear.
When the end of the document is encountered the machine is in state \( q_{19} \), and it returns \( T_{accept}[q_{19}] = \{o_2\} \). This is correct; indeed, \( P_2 \) matches the XML document, while \( P_1 \) does not. We will show in the next section that this machine implements correctly the workload \( \{P_1, P_2\} \). It should be obvious, however, 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** The bottom up part of the computation (i.e. \( Q^b \) and its associated tables) is critical for the XPush machine, since this allows us to compute both the structure navigation and the boolean predicates in the XPath queries. The top-down part is redundant (it may express the structure navigation only), but we introduced it in order to make possible certain optimizations described 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. Notice that this method differs significantly from that in [17], which translates a single XPath expression into multiple push-down transducers.

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, 18] 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 \( \varepsilon \) transitions and denote \( \delta : S \times (\Sigma \cup \{\varepsilon\}) \rightarrow \mathcal{P}(S) \) 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 \( \pi_v(v) \) the truth value of that predicate on \( v \in \mathcal{V} \). For nonterminal states we set \( \pi_v(v) = false \). Without loss of generality we impose the following constraints, which help us simplify the presentation: AND and NOT states have only \( \varepsilon \) 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 either \( x \) is a data value node and \( \pi_v(x) \) is true, or there exists some transition \( s' \in \delta(s, a) \), and some child \( y \) of \( \pi_v(x) \) 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 \( s \).

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 [10, 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. 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 \( P_1, P_2 \) in our running example. Here \( S = \{0, 1, 2, \ldots, 12\} \), \( s_1 = 0 \), \( s_2 = 7 \). States 1 and 8 are AND states and each has two \( \varepsilon \) 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 \( \delta \): e.g. \( \delta(4, a) = \{4, 5\} \), \( \delta(4, b) = \{4\} \), and \( \delta(4, @c) = \emptyset \). To illustrate predicates on data values, we have: \( \pi_6(s_5) = true \), \( \pi_6(1) = false \), \( \pi_1(v) = false \), \( \forall v \in \mathcal{V} \), etc. The states in the AFAs correspond to subqueries in the XPath filters. For example state 2 corresponds to the subquery [\( b/text()=1 \)] of \( P_1 \), while state 1 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 \( P_1 \) is true on that tree, and similarly for \( A_2 \) and \( P_2 \).

**Step 2: Constructing the bottom-up XPush Machine** Finally, in the second step, the bottom-up XPush machine is defined to be
\[
(Q^f, Q^b, q_0^f, t_{value}, t_{pop}, t_{add}, t_{accept})
\]
where:
\[
Q^f = \{q^f\}
\]
\[
Q^b = \mathcal{P}(S)
\]
\[
q_0^b = \emptyset
\]
\[
t_{push}(q^f, a) = q^f
\]
\[
t_{value}(q^f, v) = \{s \mid s \in S, \pi_v(v) = true\}
\]
\[
t_{pop}(q^b, a) = \delta^{-1}(\text{eval}(q^b), a)
\]
\[
t_{add}(q^b, q^f) = q^b \cup q^f
\]
\[
t_{add}(q^f, q^b) = q^f
\]
\[
t_{accept}(q^b) = \{o_i \mid o_i \in I, s_i \in b\}
\]

We first explain the two notations introduced in \( t_{pop} \):

\(^2\)This is one node above the top-most element node, see the formal XPath semantics in [8].
\( \delta^{-1}(q, a) \) denotes \( \{ s' \mid \delta(s', a) \cap q \neq \emptyset \} \), while \( \text{eval}(q) \) 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 \( \text{not} \) 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 \( \text{not}(\text{not}(Q)) \). The details are straightforward and we omit them.

We now explain the bottom-up XPush machine. There is a unique top-down state \( q^b \) and the bottom-up states are sets of states in \( A, P(S) \). The idea behind this construct is that the machine keeps track of which AFA states match the current XML node. For a leaf XML node with value \( v \), this set is precisely \( t_{\text{value}}(q^b, v) = \{ s \mid \pi_v(s) = \text{true} \} \). To compute these sets after an \( \text{EndElement}(a) \), 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 \( \text{eval}(q^b) \); next compute all AFA states that matched \( a \)'s parent based on these matches: this is \( t_{\text{top}}(q^b, a) \); finally union these with the previous states that matched \( a \)'s parent, retrieved from the stack. Obviously, \( t_{\text{accept}}(q^b) \) returns the oids of those XPath expressions whose initial states are in \( q^b \).

\textbf{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 consider, 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 \( t_{\text{value}}, t_{\text{top}}, t_{\text{add}} \) as defined above. We only retain the accessible states in the bottom-up XPush machine.

\textbf{Example 3.4} Figure 4 illustrates the construction of the bottom-up XPush machine from the two AFAs for \( P_1 \) and \( P_2 \) 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 \( t_{\text{value}} \), and obtain\(^3\): \( t_{\text{value}}(q^b, 1) = \{ 3, 10 \} = q_6 \), \( t_{\text{value}}(q^b, x) = \{ 6, 12 \} = q_4 \) for \( x > 2 \), and \( t_{\text{value}}(q^b, x) = \emptyset = q_0 \) for all other values of \( x \); in practice we obtain \( t_{\text{value}} \) by computing the atomic predicate index. Next we apply the function \( t_{\text{top}} \): \( t_{\text{top}}(q_0, b) = \{ 2, 9 \} = q_3 \), because \( \text{eval}(q_0) = q_0 = \{ 3, 10 \} \) and, following a \( b \) transition backwards, one reaches the states 2, 9. Similarly \( t_{\text{top}}(q_4, \@c) = \{ 5, 11 \} = q_8 \), and \( t_{\text{top}}(q_8, a) = \{ 4 \} = q_7 \). To illustrate addition, we have \( t_{\text{add}}(q_8, q_7) = \{ 2, 9 \} \cup \{ 4 \} = \{ 2, 4, 9 \} = q_11 \). To understand how AND states are handled (and similarly NOT, OR states) consider \( \delta^{-1}(q, a) \)

\( t_{\text{top}}(q_{11}, a) \). We first compute \( \text{eval}(q_{11}) = \text{eval}(\{2, 4, 9\}) = \{1, 2, 4, 9\} \). The meaning is that if states 2 and 4 have matched, then so has state 1. Next we follow \( a \) transitions backwards from these states and obtain \( t_{\text{top}}(q_{11}, a) = \{0, 4\} = q_4 \). All states and transitions in Fig. 3 are obtained this way. 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 \( \text{EndElement}(b) \) the current state is \( q_6 = \{2, 9\} \), meaning that the AFA states 2 and 9 have matched so far, corresponding to the common subquery \( [b/\text{text()}=1] \) in both \( P_1 \) and \( P_2 \). After reading the second \( \text{EndElement}(b) \) the current state is \( q_{12} = \{2, 5, 9, 11\} \) which means that the following subqueries have matched: \( [b/\text{text()}=1], [\@c > 2 \land b/\text{text()}=1] \), and \( [\@c > 2] \). In other words, the states in the bottom-up XPush machine eliminate common subexpressions between filters.

\section{4. IMPLEMENTATION}

The XPush machine needs to be computed lazily. We explain here why, and describe the runtime data structures that we used.

\subsection{4.1 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 com-

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure4.png}
\caption{AFAs for \textit{P1}, \textit{P2} in Example 1.1, and the states of the corresponding bottom-up XPush machine. The transition tables are shown in Fig. 3.}
\end{figure}
puting 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:

\[
\text{/person[name/text()="John"]}
\]

\[
\text{/person[name/text()="Smith"]}
\]

\ldots

each looking for a different value for \text{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 \text{person} to have only one \text{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

\[
\text{/person[phone/text()="v"]}
\]

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 explained in Theorem 6.2.

### 4.2 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 \( t_{pop}(q', a) \) 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 \( q = \text{eval}(q') \), 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 \( q^s \) and the sorted set of new states that need to be inserted in \( q^b \). We omit the details. Finally, \( t_{add}(q', q^b) \) is imply a merge-join of two sorted arrays.

### 4.3 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, \( t_{value} \), and all the XPush states that result from that.

### 5. OPTIMIZATIONS

The goal of the optimizations is to improve the performance of the lazy XPush machine during state construction: if all states were already computed there were no need for optimizations. We describe two classes of optimizations: the first prunes states, the second trains the XPush machine with synthetic data (training data) before running it on real data.

#### 5.1 State Pruning

**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 \(<\text{color}>\) may occur both in a \(<\text{product}>\) element and in a \(<\text{building}>\). After reading the XML fragment:

\[
<\text{product}>\ldots <\text{color}>\text{green} </\text{color}> \ldots
\]

an XPush state will be created containing AFA states from all the filters that contain the predicate \([\text{color/text()="green"]\), irrespective of whether this is under a \text{product} or a \text{building}. Clearly, those predicates that occur under a \text{building} are false leads and will be invalidated later, but they will unnecessarily double the number of states corresponding to the \text{building} element. The top-down pruning optimization eliminates the wrong AFA 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:

\[
Q^t = P(S)
\]

\[
t^q_0 = \{s_1, \ldots, s_n\}
\]

\[
t_{push}(q^t, a) = \text{close}(\delta(s, a) \mid s \in q^t)
\]

\[
t_{value}(q^t, v) = \{s \mid s \in q^t, \pi_s(v) = \text{true}\}
\]

\[
t_{add}(q^t, q^b) = q^t
\]

\[
close(q^t) = \text{repeat} q^t := \{\delta(s, \varepsilon) \mid s \in q^t\}
\]

until no-more-change

\[
\text{return } q^t
\]

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

\[
\text{/person[name="Smith" and age="33" and phone="5551234"]}
\]
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 $2^2$ subsets of predicates can be activated by some XML document. To prevent this, we define a partial order relation on all element and attribute labels: $a < b$ 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: $s < s'$ 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 $s \neq s'$ and $s' \neq s$. Using this relation we make the following changes in the definition of the XPush machine, where $\text{prec}(s) = \{ s' | s' < s \}$:

$$t'_\text{add}(q_s^a, q_s^b) = q_s^c \cup \{ s | s \in q_s^d, \text{prec}(s) \subseteq q_s^e \}$$

Other Pruning Optimizations We have considered two additional optimizations: deep order optimization and guard optimization. The first applies the order optimization higher in the tree, by moving the optimization logic to the top-down component of XPush machine. The second purges states when an element is encountered that guarantees that a predicate will never be satisfied. However we observed no significant improvements due to these optimizations in our experiments, so we do not discuss them any further.

5.2 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. Wildcards * and // are expanded using the DTD, and boolean connectors are simply ignored. For example, the query: `/a[(b/text())=3 and c=a=4] or d/text()=5` will result into the following training document:

```
<a c=4> <b> 3 </b> </a> <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 the 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 significant savings.

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 which makes 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 $s \perp s'$. Finally we say that $s, s'$ are independent if neither $s \Rightarrow s'$, nor $s' \Rightarrow s$, nor $s \perp s'$. 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 $q^s$, we associate to it a clique by removing all AFA states $s$ s.t. there exists $s' \in q^s$ 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 $7 \Rightarrow 4$: as a consequence the set $\{0, 7\}$ is not a state in the XPush machine. Also, $3 \Rightarrow 10$, and $3 \perp s$ for every state $s \neq 10$, since we assume that the XML documents have no mixed content. As a consequence the only XPush state containing 3 is $q_8 = \{3, 10\}$.

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 dramatically as $k$ increases. 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_i/text() = v_1 and ... 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 predicate has the same probability $\sigma$ of being true on a given XML document; $\sigma$ 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, \(b_1 < b_2 < \ldots\), 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:

\[
N\left(1 - \frac{\sigma^{k+1}}{1 - \sigma}\right)^n
\]

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 \(\sigma^k(1 - \sigma)^{n-k}\); 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 \(\sum_{k=0,n} \binom{n}{k}k\sigma^k(1-\sigma)^{n-k} = \sigma n\). 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 \(\sigma^{r_i}\). Thus, the expected number of states in the XPush machine is:

\[
N \sum_{0 \leq r_1 \ldots r_n \leq k} \sigma^{r_1} \cdots \sigma^{r_k} = N\left(1 - \frac{\sigma^{k+1}}{1 - \sigma}\right)^n
\]

\(\square\)

This analysis reveals two things. First, as the selectivity \(\sigma\) decreases, there are fewer expected states. Second, 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. That is, XPush machines 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 requirement 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](http://pir.georgetown.edu)) and NASA ([xml.gsfc.nasa.gov](xml.gsfc.nasa.gov)), but report only the results for Protein, for lack of space (the results for NASA were similar). We generated synthetic XPath queries using a modified version of the generator in [10]: 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 query is true on at least some XML documents. 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. All experiments are on a Pentium III, 700MHz, dual processor, with 2GB of memory, running RedHat 7.1.

**Effectiveness of the XPush** In Fig. 5 (a) we show the overall running time on a 9.12MB fragment for workloads between 5000 and 20000 queries with about 10,4 branches per query: hence the total number of atomic predicates is up to 200000. 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 1.2 seconds including parsing time, and should be compared with the time taken by Apache’s parser to parse that data set, 2.5s (we used a faster parser in the XPush machine). This confirms that the XPush machine can be very efficient, and the only significant cost is that associated to lazy state computation.

**Runtime memory requirements** Fig. 5 (c), (e), (g) show the total number of states, the average size of each state, and the total amount of memory used (their product, multiplied by 4). For a workload of 20000 queries, each with 10 branches, the number of states in (c) was below 165000. Clearly, this is 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: each indeed prunes some states in the lazy XPush machine. The effect of the optimizations is even more dramatic in (e), where we show the average size of each state: the order optimization, with or without the top-down optimization, reduces the size of states by a factor of 2-3. Still, combined, the two components result in a slightly above linear increase of the total memory requirement as a function of the workload, (g). The graphs in Fig. 5 (d), (f), (h) show the same measures, but now we increased the number of predicates per query while keeping the total number of atomic predicates fixed. As we predicted in Theorem 6.2, the number of states decreased. As a consequence, the running time for these queries, shown in Fig. 5 (b) decreases too. This shows that the XPush machine is most effective when the queries have many branches. Finally, Fig. 6 (c)-(h) shows 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 just remember configurations that we have seen before, and can be deleted when we run out of memory and recomputed later. In Fig. 6 (a) and (b) 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 or so has been processed the hit ratio is well above 90%, then increases to over 94%.

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. The only exception is the top-down optimization in isolation: the explanation here is that here 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.

Other experiments As mentioned, similar experiments on a different data set (NASA) yielded similar results. We ran the same experiments on query workloads with *’s and //’s and observed a slight degradation in performance, because the order optimization is much less effective there. We omit them for lack of space.

8. CONCLUSIONS
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 pushdown 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 on 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 state at runtime is recovered later: the hit ratio in our experiments was well over 90%, even over 96% after processing large amounts of data. Finally, we have shown that a combination of optimizations improved significantly the runtime performance of the XPush machine.

9. REFERENCES
Figure 5: Experiments
Figure 6: Experiments