Streaming XPath Processing with Forward and Backward Axes

Charles Barton, Philippe Charles
Deepak Goyal, Mukund Raghavachari
IBM T.J. Watson Research Center

Marcus Fontoura, Vanja Josifovski
IBM Almaden Research Center

Abstract

We present a novel streaming algorithm for evaluating XPath expressions that use backward axes (*parent* and *ancestor*) and forward axes in a single document-order traversal of an XML document. Other streaming XPath processors, such as YFilter, XTrie, and TurboXPath handle only forward axes. We show through experiments that our algorithm significantly outperforms (by more than a factor of two) a traditional non-streaming XPath engine. Furthermore, since our algorithm only retains relevant portions of the input document in memory, it scales better than traditional XPath engines. It can process large documents; we have successfully tested documents over 1GB in size. On the other hand, the traditional XPath engine degrades considerably in performance for documents over 100 MB in size and fails to complete for documents of size over 200 MB.

1 Introduction

XPath 1.0 [8], a language for addressing parts of XML [4] documents, is an integral component of languages for XML processing such as SQLX [13], XSLT [7] and XQuery [10]. The performance of implementations of these languages depends on the efficiency of the underlying XPath engine. XPath expressions have also been used as a general-purpose mechanism for accessing portions from XML documents, for example, an XPath-based API is provided in DOM 3 [16] for traversing DOM [11] trees. XPath expressions have found use in publish-subscribe systems as a mechanism for specifying content-based subscriptions [1, 5]. Given the central role that XPath plays in the XML stack, algorithms for improving the performance of evaluating common XPath expressions are essential.

In many environments, it is natural to treat the data source as a stream, processing queries on the data source as it is parsed. Examples include XML filtering systems [1, 9, 5], Continuous Query Systems [6],

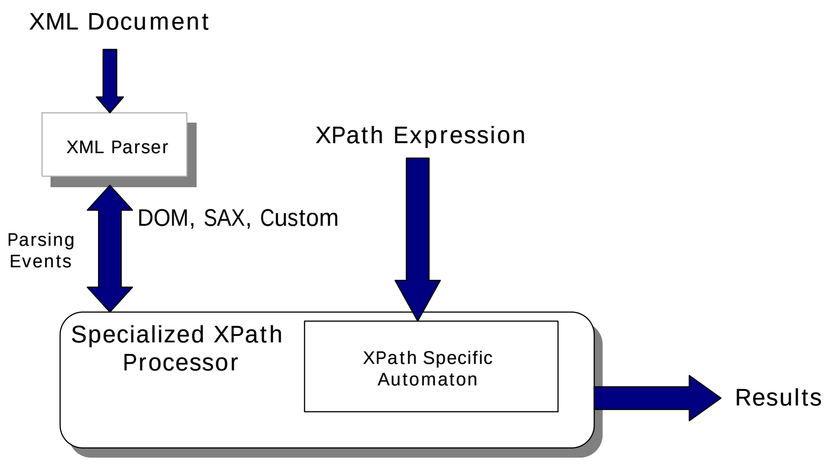


Figure 1: Structure of a streaming XPath processor.

and systems where high-volume XML data sources are integrated in a federated manner [12]. An XPath engine that operates on a streaming data source is structured as shown in Figure 1. An XPath expression is analyzed and represented as an automaton. The XPath engine consumes events (for example, SAX events) produced by a parser. For each event, the automaton may make state transitions, and if necessary, store the element. At the end of processing the stream (or document), the XPath engine returns the list of elements that is the result of the evaluation of the XPath expression.

Most current XPath engines, for example, the one provided with Xalan [2], require that an entire document be in memory before evaluating an XPath expression. For large documents, this approach may result in unacceptable overhead. Furthermore, the XPath engine in Xalan evaluates XPath expressions in a naive manner, and may perform unnecessary traversals of the input document. For example, consider an expression such as /descendant::x/ancestor::y, which selects all y ancestors of x elements in the document. The Xalan XPath engine evaluates this expression by using one traversal over the entire document to find all the x elements, and for each x element, a visit to each of its ancestors to find appropriate y elements. As a result, some elements in the document may be visited more than once.

The premise of streaming XPath is that in many instances XPath expressions can be evaluated in one depth-first, document-order traversal of an XML document. The benefits of streaming XPath are twofold. First, rather than storing the entire document in memory, only the portion of the document relevant to the evaluation of the XPath is stored. Second, the algorithm visits each node in the document exactly once, avoiding unnecessary traversals.

In this paper, we present the $\chi\alpha o\varsigma$[1] algorithm, which can evaluate XPath expressions containing both backward (such as parent and ancestor) and forward axes (such as child and descendant) in a streaming fashion in time linear in the size of the document. Other streaming XPath processors, such as YFilter [9], XTrie [5], and TurboXPath [12] handle only forward axes.

Central to the $\chi\alpha o\varsigma$ algorithm is the conversion of an XPath expression into a representation, called the *x-dag*, in which all uses of backward axes are converted into forward constraints — a key step in making streaming XPath processing possible. This representation is used to perform the two major operations of the algorithm: filtering incoming events to select those that affect the evaluation of the XPath expression, and recording information in a data structure called the *matching structure* when parts of the input XPath expression have been satisfied. These two operations are interleaved so that at the end of processing the document, the matching structure contains the solution of the evaluation of the XPath expression. This paper makes the following significant contributions:

1. A novel streaming algorithm for handling both backward and forward axes, which can be extended to handle all XPath axes. The $\chi\alpha o\varsigma$ algorithm can handle both recursive and non-recursive documents.

2. A concise representation of an XPath expression, called x-dag (Section 3.2) in which all backward constraints are converted into forward constraints. The x-dag is also a convenient representation for intersections and joins of XPath expressions, which we shall discuss only briefly in this paper.

3. A data structure called the matching structure (Section 4.2) that compactly represents all matchings (Section 3.3) of an XPath expression in a document. The result of the evaluation of the XPath expression can be computed easily and efficiently from this data structure.

This paper is structured as follows. We first present background on XPath expressions in Section 2. Then, in Section 3, we introduce a convenient tree-based representation of XPath expressions, called the *x-tree*, and explain how it can be converted into an x-dag. The x-dag is the central data structure of our algorithm. We also define the semantics of the evaluation of XPath expressions in terms of the notion of *matchings* on the x-tree and the x-dag. In Section 4, we present an overview of our algorithm and describe extensions to our algorithm in Section 5. Our experimental results are discussed in Section 6, and finally, we conclude in Section 7.

[1] $\chi\alpha o\varsigma$ (Xaos, pronounced Chaos) is an acronym for XML Analysis, Optimization, and Stuff

1.1 Related Work

Our work is most closely related to the *XFilter* [1], *YFilter* [9], *XTrie* [5], and *TurboXPath* [12] systems, all of which involve evaluation of XPath/XQuery-based queries on streaming XML documents. XFilter, YFilter, and XTrie are XML filtering systems where documents are routed and filtered based on subscriptions that are expressed as queries. The TurboXPath system has been used for XML-enabled data integration where user queries can operate over a mixture of locally stored data in a relational database and data streamed from external sources. The XFilter system handles simple XPath location path expressions (straight-line path expressions without any branching and predicates) by transforming them into a Deterministic Finite Automaton. The YFilter system is an extension of XFilter in which a group of simple XPath location path expressions are combined into a single Nondeterministic Finite Automaton (NFA), which corresponds to the union of these path expressions. Both XTrie and TurboXPath can handle tree-shaped path expressions involving predicates (which are internally represented as trees called the XTrie and ParseTree respectively). In addition, TurboXPath can also handle multiple output nodes. However, all of these systems are limited to handling location path expressions that only contain forward axes (e.g. child, descendant, forward-sibling). The $\chi\alpha o\varsigma$ system improves upon these systems by adding the ability to handle both backward (e.g. parent, ancestor) and forward axes in the context of streaming XML. Our approach also handles multiple output nodes but we will discuss it only briefly in this paper.

Tozawa and Murata [15] describe a method for converting an XPath expression into modal logic formulas with past modalities. They present an algorithm for converting such formulas into tree automata, which can be used to evaluate XPath expressions on an input document. Their paper describes a theoretical approach that can handle all XPath axes. The current status of the implementation of their algorithm is unclear. It would be interesting to compare the performance of their implementation with that of $\chi\alpha o\varsigma$.

The *NiagaraCQ* [6] system is a continuous query system that supports querying of distributed XML datasets using an XML query language. Continuous queries allow users to receive new results as they become available. The focus of the NiagaraCQ project is on exploiting similarities in structure of queries to share computation across groups of queries, and use of incremental group optimization and incremental evaluation techniques. However, the queries that they focus on involve simple structural pattern matching rather than XPath/XQuery-based queries that we deal with in this paper.

2 Background

We describe the tree model of XML documents that is the basis of the definition of XPath. We then describe the event stream that drives the $\chi\alpha o\varsigma$ algorithm. Finally, we present the subset of XPath that we focus on in this paper.

2.1 Tree Model for XML Documents

An XML document can be represented as a tree whose nodes represent the structural components of the document — elements, text, attributes, comments, and processing instructions. Parent-child edges in the tree represent the inclusion of the child component in its parent element, where the scope of an element is bounded by its start and end tags. The tree corresponding to an XML document is rooted at a virtual element, Root, which contains the document element. We will, henceforth, discuss XML documents in terms of their tree representation; $\mathcal{D}$ represents an XML document, and $V_{\mathcal{D}}$ and $E_{\mathcal{D}}$ denote its nodes and edges respectively. Figure 2 illustrates the tree representation of an XML document.

For simplicity of exposition, we focus on elements in this paper, and ignore attributes, text nodes, etc. The tree, therefore, consists of the virtual root and the elements of the document. To avoid confusion between the XML document tree and the tree representation of the XPath (described later), we use *elements* to refer to the nodes of the XML tree. We assume that the following functions are defined on the elements of an XML document:

- $id : V_{\mathcal{D}} \rightarrow Integer$: Returns a unique identifier for each element in a document.
- $tag : V_{\mathcal{D}} \rightarrow String$: Returns the tag name of the element.
- $level : V_{\mathcal{D}} \rightarrow Integer$: Returns the distance of the element from the root, where $level(\mathsf{Root}) = 0$.

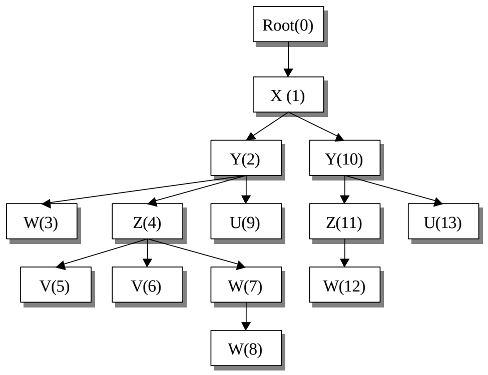
We use $\mathsf{x}_{i,l}$ to denote an element with $tag = \mathsf{x}, id = i, level = l$.

2.2 Event-Based Parsing

An event-based parser, for example, a SAX parser, scans an XML document, producing events as it recognizes element tags and other components of the document. We register functions that are invoked by the parser on start and end element events. Each event conveys the name and level of the corresponding element. The production of events is equivalent to that of a depth-first, pre-order traversal of the document

```
<X>
  <Y>
    <W/>
    <Z>
      <V/>
      <V/>
      <W>
        <W/>
      </W>
    </Z>
    <U/>
  </Y>
  <Y>
    <Z>
      <W/>
    </Z>
    <U/>
  </Y>
</X>
```

(a)



(b)

Figure 2: (a) An XML Document (b) Tree representation of the same document. The number in parentheses next to the tag of each element is the id of the element.

Table 1: XPath subset addressed in paper.

$AbsLocationPath$	:=	$'/'\ RelLocationPath$
$RelLocationPath$	:=	$Step\,'/'\ RelLocationPath \mid Step$
$Step$	:=	$Axis :: NodeTest \mid Step\,'['\ PredicateExpr\,']'$
$PredicateExpr$	:=	$RelLocationPath\ and\ PredicateExpr \mid AbsLocationPath\ and\ PredicateExpr \mid$ $RelLocationPath \mid AbsLocationPath$
$Axis$	:=	$ancestor \mid parent \mid child \mid descendant$
$NodeTest$	:=	$String$

tree, where for each element, a start element event is generated, then its subtree is processed in depth-first order, and finally, an end element event is generated.

2.3 XPath

The XPath language defines expressions for addressing parts of an XML document. We focus on *location path* expressions which evaluate to a set of elements in the document. A location path is a structural pattern composed of sub-expressions called *Step*, joined by the '/' character. Each step consists of an *axis specifier*, a *nodetest*, and zero or more predicates. Location paths are *absolute* if they begin with a '/'; otherwise they are *relative*. Table 1 provides the BNF for the XPath subset that we shall use in this paper (we refer to expressions satisfying this grammar as Restricted XPaths – $\mathcal{R}xp$).[2]

XPath expressions are evaluated relative to a context node in the document tree. The context node for an

[2]For simplicity, we do not include abbreviated XPath expressions in the grammar.

absolute location path is always the root element. To evaluate a relative location path, *Step / RelLocationPath*, with respect to a context node, c, one first computes *Step* relative to c, yielding a set of elements, $\mathcal{N}$. The meaning of *Step / RelLocationPath* is the union of the sets of elements obtained by evaluating *RelLocationPath* in context d, where d ranges over $\mathcal{N}$. The rightmost *NodeTest* not contained in a *PredicateExpr* determines the output of the XPath expression.

The set of elements searched in the evaluation of a *Step* at a context node, c, depends on its axis specifier. For example, the result of evaluating descendant::section is the subset of the proper descendants of the context node that have the tag section. While the $\chi\alpha o\varsigma$ algorithm is extensible to handle all thirteen axis specifiers in XPath 1.0, in this paper we only focus on four: child, descendant, parent, and ancestor.

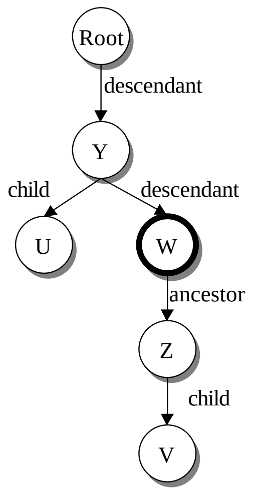
Steps may contain predicates, which restrict the set of elements selected. For example, the expression descendant::chapter[ancestor::book and child::table] selects all chapter descendants of the context node that have a book element as an ancestor and a table element as a child. Note that each chapter element is used as a context node in evaluating the subexpressions, ancestor::book and child::table.

3 X-tree, X-dag, and Matchings

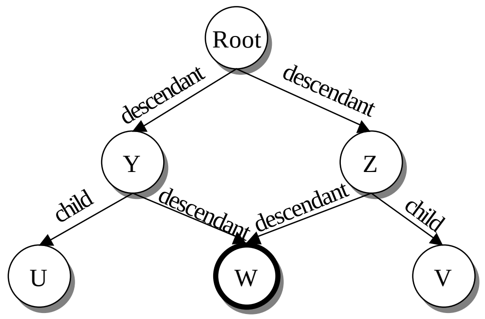
The $\chi\alpha o\varsigma$ algorithm operates on two representations of an XPath expression called x-tree and x-dag. The x-dag is a key construct in our algorithm since it converts backward constraints, such as parent, into forward constraints, thus making streaming processing possible. We use an alternate semantics of XPath expressions defined on x-trees and x-dags based on the notion of *matchings*. It can be shown that our semantics is equivalent to the semantics provided in the XPath 1.0 specification.

3.1 X-tree

We represent an $\mathcal{R}$*xp* expression as a rooted tree $\mathcal{T} = (V_{\mathcal{T}}, E_{\mathcal{T}})$, called x-tree, which has labels on both vertices and edges. The root of the tree is labeled Root. We use the term x-node to refer to the vertices of an x-tree. For every *NodeTest* in the expression, there is an x-node in the x-tree labeled with the nodetest. Each x-node (with the exception of Root) has a unique incoming edge, which is labeled with the *Axis* specified before the *NodeTest*. The x-node corresponding to the rightmost *NodeTest* which is not contained in a *PredicateExpr* is designated to be the output x-node. There are functions, $label : V_{\mathcal{T}} \rightarrow String$, and $axis : E_{\mathcal{T}} \rightarrow \{ancestor, parent, child, descendant\}$ that return the labels associated with the x-nodes



(a)



(b)

Figure 3: (a) X-tree representation of /descendant::Y[child::U]/descendant::W[ancestor::Z/child::V] (b) X-dag representation of the same XPath expression. The circles corresponding to W has a thick edge to represent the fact that it is the output node.

and edges respectively. The x-tree data structure is similar in spirit to XPE trees [5], and the *parse tree* of TurboXPath [12]. We provide rules for building an x-tree from an $\mathcal{R}xp$ in Appendix A for the interested reader. Figure 3a provides an example of an x-tree.

3.2 X-dag

We also use a directed, acyclic graph representation of an $\mathcal{R}xp$ called an x-dag. The x-dag is obtained from the x-tree by reformulating the ancestor and parent constraints in the tree as descendant and child constraints. More precisely, it is a directed, labeled graph, $\mathcal{G} = (V_\mathcal{G}, E_\mathcal{G})$, with the same set of vertices as $\mathcal{T}$, and edges defined as follows:

1. Edges in $\mathcal{T}$ labeled child or descendant are also edges of $\mathcal{G}$.
2. For each edge in $\mathcal{T}$ labeled parent, there is an edge joining the same nodes but with direction reversed and label changed to child. Similarly, ancestor edges are reversed and relabeled as descendant edges.
3. For any non-root x-node $v \in \mathcal{G}$ that has no incoming edges, a descendant edge is added from Root to v.

Figure 3b gives the x-dag corresponding to the x-tree in Figure 3a.

3.3 Matchings

Let v_1 and v_2 be two x-nodes in an x-tree $\mathcal{T}$ connected by an edge e, and let d_1 and d_2 be two elements in a document $\mathcal{D}$, where $tag(d_1) = label(v_1)$ and $tag(d_2) = label(v_2)$. We say that the pair (v_1, d_1) is *consistent* with (v_2, d_2) (relative to x-tree $\mathcal{T}$ and document $\mathcal{D}$) if d_1 and d_2 satisfy the relation $axis(e)$. For example, if v_1 and v_2 are connected by an edge labeled ancestor, then d_2 must be an ancestor of d_1 in $\mathcal{D}$. A matching for $\mathcal{T}$, $m : V_{\mathcal{T}} \rightarrow V_{\mathcal{D}}$, is a partial mapping from x-nodes of x-tree $\mathcal{T}$ to elements of document $\mathcal{D}$ such that the following conditions hold.

1. For all x-nodes $v \in domain(m)$, $label(v) = tag(m(v))$, *i.e.* all mapped vertices satisfy the nodetest.

2. For all x-nodes v_1 and v_2 connected by an edge in $\mathcal{T}$ such that $v_1, v_2 \in domain(m)$, $(v_1, m(v_1))$ is consistent with $(v_2, m(v_2))$.

A matching is *at an x-node* v if and only if its domain is contained in the sub-tree rooted at v. A matching at v is total if its domain contains all the vertices of the subtree rooted at v. Let $\mathcal{T}$ denote the x-tree corresponding to an *$\mathcal{R}$xp* r. It is easy to show that a document element n is in the result of the evaluation of *$\mathcal{R}$xp* r, if and only if, there exists a total matching for $\mathcal{T}$ at Root in which the output x-node of $\mathcal{T}$ is mapped to n. $\chi\alpha o\varsigma$ evaluates an *$\mathcal{R}$xp* r precisely in this manner. It finds all total matchings for $\mathcal{T}$ at Root, and emits the document elements that correspond to the output x-node.

The notion of a matching can be analogously extended to an x-dag. A matching is *at an x-node* v *of x-dag* $\mathcal{G}$ if and only if its domain is contained in the *sub-dag* rooted at v. Once again, it is easy to show that a total matching at the root of a x-tree $\mathcal{T}$ is also a total matching at the corresponding x-dag $\mathcal{G}$, and vice-versa.

4 The $\chi\alpha o\varsigma$ Algorithm

Central to the $\chi\alpha o\varsigma$ algorithm is the observation that a total matching at an x-node, v, is composed of total matchings at each of the children of v in $\mathcal{T}$. Let $w_1, w_2, \ldots, w_n$ denote the children of v in $\mathcal{T}$ (in an arbitrary, but fixed order) and let $m_1, m_2, \ldots, m_n$ be total matchings at $w_1, w_2, \ldots, w_n$ respectively. Let e be an element in a document $\mathcal{D}$ such that 1) $tag(e)$ is the same as $label(v)$, and 2) for each child w_i of v, (v, e) is consistent with $(w_i, m_i(w_i))$. Then, a total matching at v can be obtained trivially by taking a disjoint union of all the maps m_i and the singleton map $[v \mapsto e]$[3]. For example, looking at the x-tree in

[3]The singleton map $[v \mapsto e]$ refers to a partial map which maps v to e and is undefined everywhere else.

Figure 3a and document in Figure 2, a total matching at Z *i.e.* $[\mathsf{Z} \mapsto Z_{4,3}, \mathsf{V} \mapsto V_{5,4}]$, together with the fact that $(\mathsf{W}, W_{7,4})$ is consistent with $(\mathsf{Z}, Z_{4,3})$, yields the total mapping $[\mathsf{W} \mapsto W_{7,4}, \mathsf{Z} \mapsto Z_{4,3}, \mathsf{V} \mapsto V_{5,4}]$ at W.

We can make a similar observation about a total matching at x-node v of an x-dag $\mathcal{G}$, but it is more complex than the case of an x-tree. In particular, the existence of a total matching at v implies the existence of total matchings at each of its children in $\mathcal{G}$ but the converse is not true. The complication arises from the fact that the set of x-nodes in the sub-dags rooted at each of the children of v are not necessarily disjoint. We refer to these x-nodes that are shared by more than one sub-dag as *join* points. For example, consider the x-dag in Figure 3b. The sub-dags at Y and Z share a common x-node W, which is therefore a join point. Consider total matchings at Y and at Z in our example. The existence of these matchings do not necessarily imply the existence of a total matching at Root. For there to be a total matching at Root, the two total matchings at Y and Z must agree on W, *i.e.*, must map W to the same element in the document. For example, the total mappings $[\mathsf{Y} \mapsto Y_{10,2}, \mathsf{W} \mapsto W_{12,4}, \mathsf{U} \mapsto U_{13,3}]$ and $[\mathsf{Z} \mapsto Z_{4,3}, \mathsf{W} \mapsto W_{7,4}, \mathsf{V} \mapsto V_{5,4}]$ at Y and Z respectively cannot be combined to form a total matching at Root.

In general, an algorithm that constructs a total matching at an x-node v of x-dag $\mathcal{G}$, from total matchings at each of the children of v in $\mathcal{G}$ must ensure that the total matchings at the children agree on the join points. This verification can be expensive computationally. Furthermore, total matchings at each of the children must be retained until they are composed, at which time it may be determined that these matchings do not agree on the join points, and therefore, cannot contribute to a total matching at v. To minimize storage, we would like to be able to discard such matchings as early as possible.

Consider the special situation when there are no join points in the x-dag, that is, when the x-dag is a tree (the $\mathcal{R}$*xp* does not use the parent or ancestor axis). In this case, there is a relatively straightforward algorithm for constructing total matchings by composition. For an x-node v, the algorithm starts looking for a total matching at a child v' of v once it finds elements e and e' that match v and v' respectively such that (v, e) is consistent with (v', e'). It can be easily verified that when the event corresponding to the end of element e is seen, if one has found at least one total matching at each child of v, then there must exist at least one total matching at v in which v is mapped to e. Conversely, if one has not found a total matching for one or more children of v, then there does not exist a total matching at v in which v is mapped to e.

One can extend this algorithm to handle general x-dags as long as we ensure that when the algorithm constructs total matchings from children total matchings, the consistency of the assignments to join points is checked. To avoid this verification step, we use the x-tree rather than the x-dag as the basis for constructing total matchings from subordinate total matchings. The algorithm cannot, however, be applied to x-trees directly — it is not always possible to determine at the end of an element e that matches x-node $v \in \mathcal{T}$ whether there exists a total matching at v, where v is mapped to e. For example, consider the x-tree in Figure 3a. Let e and e' be elements in the document that match W and Z respectively such that e' is an ancestor of e. When the event corresponding to the end of element e is seen, the absence of a total matching at Z does not imply the non-existence of a total matching at W in which W is mapped to e. It is possible that an element e'' (which is a child of element e') matching V will be seen later, which will contribute to a total matching at Z, and consequently, to a total matching at W.

Thus, our final algorithm uses a subtle combination of both the x-tree and the x-dag to compute total matchings at the root. The x-dag is used to filter out the relevant events from the input event stream and determine when matchings stored at the children of an x-node v can be safely discarded, *i.e.* are guaranteed not to contribute to a total matching at v. The x-tree is used as the basis for determining when and what to compose to avoid the expensive verification of join point assignments in the x-dag. In this section, we shall describe these two components of our algorithm and a data structure called the *matching-structure*, which represents a set of matchings, in greater detail. The first component deals with determining when to start looking for a total matching at a x-node. The second component is regarding the composition of matchings to construct a matching-structure that represents the set of all total matchings at the root.

In Table 2, we have provided a walk through of the execution of the algorithm on the $\mathcal{R}$*xp* of Figure 3 and the document of Figure 2.

4.1 First Component: Looking For Total Matchings

At any point during execution, $\chi\alpha o\varsigma$ has processed a prefix of the input document. An infinite number of XML documents share the same prefix, and $\chi\alpha o\varsigma$ cannot predict the future sequence of events that will be generated by the parser. An element, e, is *relevant* if there exists some document completion where e participates in a total matching at Root. All relevant elements must be processed. As events are processed, new relevant elements may be seen, or elements that were earlier deemed relevant may no longer be relevant. The x-dag representation of the $\mathcal{R}$*xp* is used to determine if an element is relevant.

An element that does not match any x-node is not relevant trivially since it cannot participate in any matching. Moreover, even some elements that match an x-node can be discarded. Consider the start element event for $W_{3,3}$. This element matches the W x-node in the x-dag, but is not relevant because it has no Z

Table 2: Walk through of evaluation of XPath of Figure 3 on document of Figure 2. **S (E)**: $A_{x,y}$ denotes the start (end) element event for an element, $A_{x,y}$. The Looking-for set column shows $\mathcal{L}$ at the end of processing the event.

	Event	Matches	Comments	Looking-for Set
1	**S**: $\mathsf{Root}_{0,0}$	$(\mathsf{Root}, 0)$	Add $(Y, *)$ and $(Z, *)$ to $\mathcal{L}$, since Root matches their ancestors in the x-dag.	$\{(Y, *), (Z, *)\}$
2	**S**: $\mathsf{X}_{1,1}$		Discarded.	$\{(Y, *), (Z, *)\}$
3	**S**: $\mathsf{Y}_{2,2}$	$(Y, *)$	Add $(U, 3)$ to $\mathcal{L}$ because U is connected to Y by a child edge in the x-dag, and Y is matched at level 2. Do not add W to $\mathcal{L}$ because there is no element that matches its Z parent in the x-dag. Continue looking for $(Y, *)$ because any element with tag Y in the subtree of this element will also be a candidate for matching Y.	$\{(Y, *), (Z, *), (U, 3)\}$
4	**S**: $\mathsf{W}_{3,3}$		Discarded. This W is not relevant because it has no match in $\mathcal{L}$.	$\{(Y, *), (Z, *)\}$
5	**E**: $\mathsf{W}_{3,3}$		Discarded.	$\{(Y, *), (Z, *)\}$
6	**S**: $\mathsf{Z}_{4,3}$	$(Z, *)$	Start looking for $(V, 4)$ since we have relevant elements matching Z and Root in the x-dag. Look for it at level 4 because the (Z, V) edge is labeled child.	$\{(Y, *), (Z, *), (W, *), (V, 4)\}$
7	**S**: $\mathsf{V}_{5,4}$	$(V, 4)$	Stop looking for $(V, 4)$ because until the end of this element, $level > 4$.	$\{(Y, *), (Z, *), (W, *)\}$.
8	**E**: $\mathsf{V}_{5,4}$	$(V, 4)$	There is a total matching at V, $\mathcal{M}_{V,5}$. This matching-structure is propagated to the appropriate submatching of $\mathcal{M}_{Z,4}$, the only parent-matching of $\mathcal{M}_{V,5}$.	$\{(Y, *), (Z, *), (W, *), (V, 4)\}$.
9	**S**: $\mathsf{V}_{6,4}$	$(V, 4)$		$\{(Y, *), (Z, *), (W, *)\}$
10	**E**: $\mathsf{V}_{6,4}$	$(V, 4)$	Again, $\mathcal{M}_{V,5}$ is added to the appropriate submatching of $\mathcal{M}_{Z,4}$.	$\{(Y, *), (Z, *), (W, *), (V, 4)\}$
11	**S**: $\mathsf{W}_{7,4}$	$(W, *)$		$\{(Y, *), (Z, *), (W, *)\}$
12	**S**: $\mathsf{W}_{8,5}$	$(W, *)$		$\{(Y, *), (Z, *), (W, *)\}$
13	**E**: $\mathsf{W}_{8,5}$	$(W, *)$	W in the x-dag has an outgoing $\mathsf{ancestor}$ edge. All child-matchings of $\mathcal{M}_{W,8}$, in this case, $\mathcal{M}_{Z,4}$, are propagated into the appropriate submatching of $\mathcal{M}_{W,8}$. All submatchings of $\mathcal{M}_{W,7}$ are now non-empty. $\mathcal{M}_{W,8}$ is propagated to $\mathcal{M}_{Y,2}$	$\{(Y, *), (Z, *), (W, *)\}$
14	**E**: $\mathsf{W}_{7,4}$	$(W, *)$	As above, $\mathcal{M}_{W,7}$ is propagated to $\mathcal{M}_{Y,2}$.	$\{(Y, *), (Z, *), (W, *)(V, 4)\}$
15	**E**: $\mathsf{Z}_{4,3}$	$(Z, *)$	Z has an incoming edge labeled $\mathsf{ancestor}$. Since $\mathcal{M}_{Z,4}$ is satisfied, no clean up is necessary.	$\{(Y, *), (Z, *)(U, 3)\}$
16	**S**: $\mathsf{U}_{9,3}$	$(U, 3)$		$\{(Y, *), (Z, *)\}$
17	**E**: $\mathsf{U}_{9,3}$	$(U, 3)$	The total matching at U, $\mathcal{M}_{U,9}$ is propagated to $\mathcal{M}_{Y,2}$.	$\{(Y, *), (Z, *), (U, 3)\}$
18	**E**: $\mathsf{Y}_{2,2}$	$(Y, *)$	$\mathcal{M}_{Y,2}$ is satisfied since both submatchings, corresponding to U and W are non-empty. Propagate $\mathcal{M}_{Y,2}$, and we have a total matching at Root.	$\{(Y, *), (Z, *)\}$
19	**S**: $\mathsf{Y}_{10,2}$	$(Z, *)$		$\{(Y, *), (Z, *), (U, 3)\}$
20	**S**: $\mathsf{Z}_{11,3}$	$(Z, *)$		$\{(Y, *), (Z, *), (V, 4), (W, *)\}$
21	**S**: $\mathsf{W}_{12,4}$	$(W, *)$		$\{(Y, *), (Z, *), (W, *)\}$
22	**E**: $\mathsf{W}_{12,4}$	$(W, *)$	Since W has an outgoing edge labeled $\mathsf{ancestor}$, add $\mathcal{M}_{Z,11}$ optimistically to the appropriate submatching of $\mathcal{M}_{W,12}$. Since this matching is now satisifed, it is propagated to $\mathcal{M}_{Y,10}$.	$\{(Y, *), (Z, *), (W, *), (V, 4)\}$
23	**E**: $\mathsf{Z}_{11,3}$	$(Z, *)$	$\mathcal{M}_{Z,11}$ is not satisfied — the submatching for V is empty. Undo the propagation of $\mathcal{M}_{Z,11}$ to $\mathcal{M}_{W,12}$. Since $\mathcal{M}_{W,12}$ now is no longer satisfied, undo the propagation from $\mathcal{M}_{W,12}$ to $\mathcal{M}_{Y,10}$.	$\{(Y, *), (Z, *), (U, 3)\}$
24	**S**: $\mathsf{U}_{13,3}$	$(U, 3)$		$\{(Y, *), (Z, *)\}$
25	**E**: $\mathsf{U}_{13,3}$	$(U, 3)$	The total matching, $\mathcal{M}_{U,13}$ is propagated to $\mathcal{M}_{Y,10}$.	$\{(Y, *), (Z, *)\}$
26	**E**: $\mathsf{Y}_{9,2}$	$(Y, *)$	$\mathcal{M}_{Y,10}$ is not satisfied. The submatching for W is empty. Nothing is propagated.	$\{(Y, *), (Z, *)\}$
27	**E**: $\mathsf{X}_{1,1}$		Discarded.	$\{(Y, *), (Z, *)\}$
28	**E**: $\mathsf{Root}_{0,0}$	$(\mathsf{Root}, 0)$	There is one entry in the submatching corresponding to Y, $\mathcal{M}_{Y,2}$. $\mathcal{M}_{\mathsf{Root},0}$ is satisfied.	$\{(\mathsf{Root}, 0)\}$

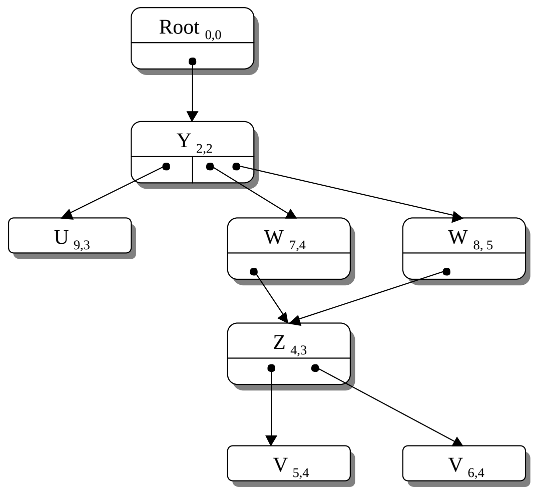
ancestor element in the document; there is no total matching in which $W_{3,3}$ participates that assigns an element z to Z such that z is an ancestor of $W_{3,3}$. Since the input document is processed in a depth-first manner, by the time the start element event for $W_{3,3}$ is processed, the algorithm has already received start element events for all ancestors of $W_{3,3}$ in the input document. It can, therefore, determine if the W element has Y and Z ancestors in the document, and discard the W element if does not satisfy these constraints.

The precise definition of when an element is relevant is as follows. An element e that matches a x-node v is relevant if and only if there exists a matching, $m : V' \rightarrow E'$, where V' is the set of x-nodes containing v and all ancestors of v in the x-dag, and E' is the set of document elements containing e and all ancestors of e in the document, such that if $v_1, v_2 \in V'$ are connected by an edge, then $(v_1, m(v_1))$ is consistent with $(v_2, m(v_2))$. For efficient determination of whether the element associated with a start element event is relevant, we maintain a *looking-for* set, $\mathcal{L}$. The members of $\mathcal{L}$ are $(v \in V_{\mathcal{P}}, level)$ pairs, where level may be an integer or $*$. The looking-for set $\mathcal{L}$ is maintained in such a way that if the element e associated with a start element event is relevant, then, and only then, there exists $(v, level) \in \mathcal{L}$ such that $label(v) = tag(e)$, and either $level = level(e)$ or $level = *$. Integer levels are used to enforce the constraint that if (v_1, e_1) and (v_2, e_2) are consistent and if $axis(v_1, v_2)$ = child, then $level(e_2) = 1 + level(e_1)$. $\mathcal{L}$ is initially set to $\{(\mathsf{Root}, 0)\}$.

For example, at the end of Step 3 in the execution of the algorithm on the XPath expression in Figure 3 on the document of Figure 2 (See Table 2), the looking for set is $\{(Y, *), (Z, *), (U, 3)\}$. $(Y, *)$ is in the looking for set because if the next start event were for an element, e, with tag Y, there would exist a matching $m : V' \rightarrow E' = \{\mathsf{Root} \mapsto \mathsf{Root}, Y \mapsto e\}$. $(Z, *)$ is in the looking for set for a similar reason. $(U, 3)$ is in the looking for set because if the next start element event for element, e, matched it, we would have a matching $m : V' \rightarrow E' = \{\mathsf{Root} \mapsto \mathsf{Root}, Y \mapsto Y_{2,2}, U \mapsto e\}$. e would have to be at level 3 for this matching to be consistent, because there is an edge labeled child between Y and U in the x-dag. We do not, however, have entries for W or V in the looking for set, because if the next start element event matched either of them, we could not construct an appropriate matching (we would not have an appropriate assignment to the Z x-node).

4.2 Matching-Structure

The second part of the algorithm constructs a data structure called a *matching-structure* which is a compact representation of all total matchings at Root of the $\mathcal{R}$*xp* relative to the input document. A matching-



Total Matchings at Root

$[\textsf{Root} \mapsto 0, Z \mapsto 4, Y \mapsto 2, U \mapsto 9, V \mapsto 5, W \mapsto 7]$
$[\textsf{Root} \mapsto 0, Z \mapsto 4, Y \mapsto 2, U \mapsto 9, V \mapsto 5, W \mapsto 8]$
$[\textsf{Root} \mapsto 0, Z \mapsto 4, Y \mapsto 2, U \mapsto 9, V \mapsto 6, W \mapsto 7]$
$[\textsf{Root} \mapsto 0, Z \mapsto 4, Y \mapsto 2, U \mapsto 9, V \mapsto 6, W \mapsto 8]$

Solution: $\{W_{7,4}, W_{8,5}\}$

Figure 4: Matching Structure at the end of processing the XPath of Figure 3. The boxes represent matching-structures. For a matching-structure, $\mathcal{M}_{v,e}$, the top half of the box shows the element that matches v. Each slot in the bottom half of the box corresponds to a submatching, which is represented as a list of pointers to the child matchings.

structure, $\mathcal{M}_{v,e}$, is associated with x-node v, and represents a set of matchings at v in which v is mapped to the document element e. The matching-structure $\mathcal{M}_{v,e}$ additionally contains a submatching for every child of v in the x-tree. A submatching at child w of v is a (possibly empty) set of matching-structures at w. For any matching-structure $\mathcal{M}_{w,e'}$ in the submatching of $\mathcal{M}_{v,e}$ at w, we require that (v, e) be consistent with (w, e'). A matching-structure $\mathcal{M}_{v,e}$ is said to be a *parent-matching* of a matching-structure $\mathcal{M}_{w,e'}$ if v is a parent of w in x-tree $\mathcal{T}$ and (v, e) is consistent with (w, e'). If $\mathcal{M}_{v,e}$ is a parent-matching of $\mathcal{M}_{w,e'}$, then we say also that $\mathcal{M}_{w,e'}$ is a *child-matching* of $\mathcal{M}_{v,e}$.

Figure 4 shows the matching structure at the end of processing the XPath of Figure 3 on the document in Figure 2, and the four total matchings at Root. The result is obtained by taking the W projection, that is $\{W_{7,4}, W_{8,5}\}$.

4.3 Second Component: Composition of Matchings

We assume from now on that all events corresponding to elements that are not relevant have been discarded. When $\chi\alpha o\varsigma$ processes a start element event for an element e that matches a x-node, v, it creates a matching-structure, $\mathcal{M}_{v,e}$, to represent the match. Note that e may match more than one x-node in the x-tree; a matching-structure is created for each such match. The submatchings for these matching-structures are initially empty. As $\chi\alpha o\varsigma$ processes events, it stitches together these matching-structures, so that when the end of the document is seen, $\mathcal{M}_{\mathsf{Root},\mathsf{Root}}$ encodes all total matchings at Root in the document.

The key step in this process is *propagation*. At an end element event for an element e that matches x-node v, we attempt to determine if $\mathcal{M}_{v,e}$ represents a total matching at v. If there is a total matching, we insert $\mathcal{M}_{v,e}$ into the appropriate submatching of its parent-matchings. This propagation may be optimistic in that one may have to undo the propagation as more events are processed. Let us first, however, consider the simpler situation where no cleanup of propagation is necessary, when the x-tree does not contain any edges labeled ancestor or parent. This corresponds to $\mathcal{R}$*xp*'s that use only the child and descendant axes.

When the x-tree contains only child and descendant constraints, any total matching m at v, where $m(v) = e$ maps all x-nodes in the subtree of v to elements that lie in the document subtree of e. Since the total matching is contained within the subtree of e, by the time the end element event for e is seen, we can determine conclusively if $\mathcal{M}_{v,e}$ represents a total matching at v. This leads naturally to an inductive approach to building matchings. For an end element event e, where $\mathcal{M}_{v,e}$ is a matching-structure:

1. If v is a leaf in the x-tree, $\mathcal{M}_{v,e}$ represents a total matching at v by definition (v has no subtrees). We propagate $\mathcal{M}_{v,e}$ to the appropriate parent-matchings.

2. If v is not a leaf, $\mathcal{M}_{v,e}$ represents a total matching at v, if and only if, all submatchings are non-empty. Otherwise, no total matching exists. If we had found appropriate total matchings for each of the children of v in the x-tree, they would have been propagated to $\mathcal{M}_{v,e}$ by the time the end element event for e is processed. As above, if $\mathcal{M}_{v,e}$ represents a total matching, we propagate it to all appropriate parent-matchings.

If at the end of processing the document (when we receive the end element event for Root), $\chi\alpha o\varsigma$ finds that all the submatchings of $\mathcal{M}_{\mathsf{Root},\mathsf{Root}}$ are non-empty, we have a total matching at Root.

The presence of ancestor and parent edges in the x-tree complicates this process because one may not be able to determine conclusively whether a total matching exists for a $\mathcal{M}_{v,e}$ by the end of element e. For example, in Figure 3a, one might not find a total matching for the subtree rooted at Z, until after one sees the end of an element matching W. The propagation process remains the same, except for a x-node that has an incoming or an outgoing edge labeled ancestor or parent. For a $\mathcal{M}_{v,e}$, the modified steps are as follows:

- If there is an outgoing edge (v, v') labeled ancestor or parent, and the submatching for v' is empty, we cannot assert that there exists no total matching at v. We, optimistically, propagate each child-matching, $\mathcal{M}_{v',e'}$, into the appropriate submatching of $\mathcal{M}_{v,e}$. We then proceed as before. If all submatchings are satisfied, $\mathcal{M}_{v,e}$ is propagated to its parent-matchings. For an example, please refer to Steps 13 and 22 in Table 2.

- If there is an incoming edge (v', v) labeled ancestor or parent, then $\mathcal{M}_{v,e}$ may have been propagated optimistically to its parent-matchings. If we can determine conclusively that $\mathcal{M}_{v,e}$ cannot represent a total matching at v, we undo the propagation of $\mathcal{M}_{v,e}$. The removal of $\mathcal{M}_{v,e}$ from a submatching of a parent-matching $\mathcal{M}_{v',e'}$ may result in that submatching becoming empty — $\mathcal{M}_{v',e'}$ is no longer a total matching at v'. We then recursively undo the propagation of $\mathcal{M}_{v',e'}$ from its parent-matchings. For an example, please refer to Step 23 in Table 2.

4.4 Emitting Output

At the end of processing the document, if the submatchings of $\mathcal{M}_{\mathsf{Root},\mathsf{Root}}$ are all non-empty, we have at least one total matching at Root. The output is emitted by traversing the matching structure, and emitting an element e when we visit $\mathcal{M}_{v,e}$, where v is the output x-node of the $\mathcal{R}$*xp*. For example, in Figure 4, we output $W_{7,4}$ when we first visit $\mathcal{M}_{W,7}$ and $W_{8,5}$, when we first visit $\mathcal{M}_{W,8}$.

5 Extensions

In this section, we discuss optimizations to our algorithm, and extensions to handle features such as *or* expressions, multiple outputs, and intersections and joins of XPath expressions.

5.1 Optimizations

We have described our algorithm in terms of storing all total matchings, and subsequently, traversing the matching structure to emit elements. We do not, however, need to build matching-structures for many of the x-nodes in the x-tree. For example, if the x-tree contains a subtree that does not contain the output node, it is not necessary to store matching structures for the nodes in that subtree. It is sufficient to store a boolean value as to whether a total matching exists at that subtree. Furthermore, often it is not necessary to wait until the end of a document to emit output, but emit elements more eagerly. A detailed discussion of these optimizations is beyond the scope of this paper.

5.2 *Or* expressions

Or expressions can be handled by converting an XPath expression into an equivalent one in "disjunctive normal form." $\chi\alpha o\varsigma$ can be run on each of the operands of the top-level 'or' independently. While this process may be exponential in terms of the size of the XPath expression, we do not expect this to be an issue since XPath expressions are, in general, of small size.

5.3 Multiple Outputs

One extension of XPath expressions is to allow for more than one output node in an XPath. If we use "$" to mark output nodes, the extended XPath expression, $a/$b returns all (a, b) pairs in an input document such that a is the parent of b. These expressions have use in the compilation of XQuery and SQLX statements [12]. Our algorithm can handle these extended expressions easily. Given an extended XPath expression in this form, we generate an x-dag in the same manner as before, except that it may now contain more than one x-node marked as an output node. Our matching semantics are independent of the number of output nodes in the x-dag, and the matching structure allows easy production of the resultant tuples — the only change is in the output traversal. A detailed description is beyond the scope of this paper.

5.4 Intersections and Joins of XPath expressions

The x-dag representation can also be viewed as a representation of the intersection of two XPath expressions. For example, the x-dag of Figure 3b can be interpreted as //Y[U]//W $\cap$ //Z[V]//W. In other words, it returns all W elements that are in the solution set of both XPath expressions when they are evaluated on an input XML document.

An x-dag with multiple output nodes, derived from an extended XPath as described previously, can also be used as a representation for joins of XPath expressions. For example, assume that the x-nodes, U, W and V were marked as output nodes in Figure 3b. The x-dag then could be interpreted either as:

/descendant::Y[child::$U]/descendant::$W[ancestor::Z/child::$V], or

/descendant::Y[child::$U]/descendant::$W $\bowtie_{\mathsf{W}}$ /descendant::Z[child::$V]/descendant::$W

Since these joins and intersections can be expressed as an x-dag with multiple output nodes, as mentioned previously, we can handle these expressions in our framework. The x-dag representation of intersections and joins allows these expressions to be evaluated in a single pass during parsing. In contrast, TurboXPath [12] advocates a more complex two-phased approach in which the burden of evaluating the intersections or joins is shifted to a backend database.

6 Experimental Results

The $\chi\alpha o\varsigma$ algorithm examines each element event exactly once and the processing of an event involves only constant-time operations. We would, therefore, expect the execution time of $\chi\alpha o\varsigma$ algorithm to be linear in terms of the input document size. Furthermore, $\chi\alpha o\varsigma$ stores only those elements relevant to the calculation of the final solution. We would, therefore, expect the $\chi\alpha o\varsigma$ algorithm to show better memory utilization than Xalan [2], which stores the whole document in memory. In this section, we provide experimental results that validate these claims. We, first, provide results using documents generated by XMark [14]. To gain further insight into the relative performance of $\chi\alpha o\varsigma$ and Xalan, we also run experiments using a custom XPath and XML document generator.

All experiments were run on a 550 MhZ, 256 MB, Pentium III box, running Linux 2.4. $\chi\alpha o\varsigma$ was written in C++, and we use Xalan-C++ 1.3.1. Both $\chi\alpha o\varsigma$ and Xalan were compiled using gcc -O (version 2.92).

6.1 Experiments using XMark

Using XMark, we generated documents with scale factors .03125, .0625, .125, .25, .5, 1, 2, and 4, respectively. These correspond to documents ranging in size from 3.5 MB to 446 MB. We then evaluate the XPath expression, //listitem/ancestor::category//name on these documents, using both $\chi\alpha o\varsigma$ and Xalan. Figure 5 reports the results of these experiments.

Note that Xalan fails to complete on the two largest documents (approx. 222 MB and 446 MB), and furthermore, that there is a sharp spike in going from 55 MB to 111 MB. These effects can be attributed to the memory requirements of Xalan (the spike is the region where Xalan exhibits thrashing behavior in memory). On the other hand, $\chi\alpha o\varsigma$ scales linearly, as is expected. Table 3 reports the number of elements discarded by the algorithm as not being relevant. As can be seen from the results, a very small percentage of elements in a document (less than .2 %) is stored and processed, resulting in a signficant reduction in storage requirements.

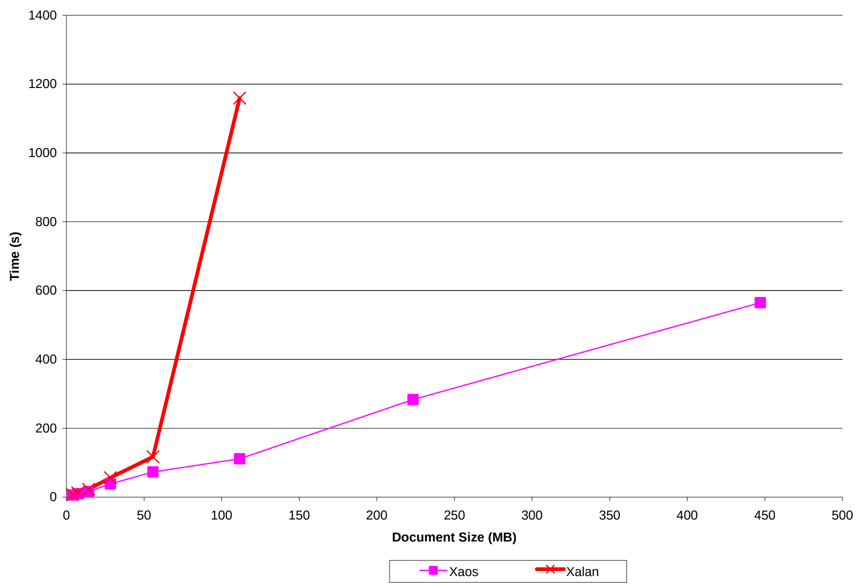


Figure 5: Time in seconds on XMark-generated documents: $\chi\alpha o\varsigma$ versus Xalan. The XPath expression executed is //listitem/ancestor::category//name

Table 3: Number of elements discarded by $\chi\alpha o\varsigma$ in processing of XMark-generated documents

Scale Factor	Document Size	Total Elements	Percentage Discarded
.03125	3.49 MB	52069	99.8 %
.625	6.88 MB	103999	99.8 %
.125	13.86 MB	210538	99.8 %
.25	27.87 MB	417160	99.8 %
.5	55.32 MB	832911	99.8 %
1	111.12 MB	166311	99.8 %
2	222.90 MB	3337649	99.8 %
4	446.71 MB	6688651	99.8 %

6.2 Custom XPath generator

We use a custom XPath generator to generate a set of random XPath expressions (of size 6 – six node tests in the expression), and for each XPath expression, we generate a random XML document based on the XPath expression. The generated XML document has the characteristic that, for large document sizes, the XPath expression will have many matches (and near matches) in the document.

We use two versions of $\chi\alpha o\varsigma$ in our comparison. The first, $\chi\alpha o\varsigma$(SAX), uses the Xerces SAX parser [3], which is also used by Xalan. To factor out the costs of parsing and building a tree from the time to evaluate an expression, we also implemented a version of $\chi\alpha o\varsigma$ on top of Xalan. $\chi\alpha o\varsigma$(DOM) builds an internal version of the input document in the same way that Xalan does. We then traverse this tree in a depth-first fashion and generate events that a SAX parser would. By subtracting the parsing and tree-building time from the overall time, we get an accurate measure of the time spent in evaluating the expression.

We vary the XML document size from 20,000 elements to 640,000 elements (200K - 6.7 MB). At each document size, we execute 10 runs of the following:

1. Generate an XPath expression.
2. Generate an XML document from the XPath expression.
3. Evaluate the XPath expression using $\chi\alpha o\varsigma$ and Xalan.

We report the average execution time and the standard deviation of the 10 runs at each XML document size.

6.2.1 Overall Execution Time

We first compare the performance of $\chi\alpha o\varsigma$ to that of using the Xalan XPath engine (SimpleXPathAPI). Figure 6 plots the average execution time (average over the 10 runs at each document size) versus document size (in number of elements). The error bars represent the standard deviation from the mean. All times include the cost of parsing.

As can be seen from the graph, $\chi\alpha o\varsigma$(SAX) is roughly 25% faster than the Xalan XPath engine. With documents of size 640,000 elements (6.7 MB) the average times are $\chi\alpha o\varsigma$: 39.0 seconds, Xalan XPath: 52.28 seconds. Note the difference in the standard deviations between the two lines (the error bars in the

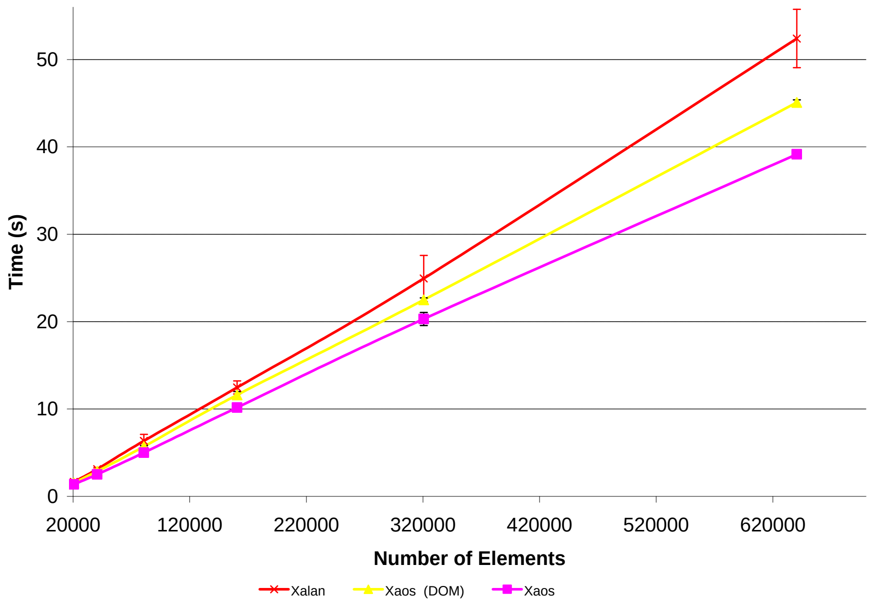


Figure 6: Overall Time in seconds: $\chi\alpha o\varsigma$ versus Xalan

plot). Whereas the standard deviation for $\chi\alpha o\varsigma$ is relatively constant, that of Xalan XPath is fairly high. We shall discuss the cause of this behavior in the next section.

6.2.2 Comparison Excluding Parsing Times

Excluding parsing costs, the performance of our XPath engine is more than twice that of the Xalan engine (Figure 7). This is mainly due to avoiding unnecessary traversals of the tree. Note that the difference in standard deviation is much more apparent in this graph. The cause of this high variance is the bimodal behavior of the Xalan XPath engine. On "good" XPath expressions, where it does not perform many unnecessary traversals, the performance of the Xalan XPath engine is similar to that of ours. On "bad" XPath expressions, such as those involving the use of the descendant axes, its performance can be four times worse. Our XPath engine's performance, however, is linear in the size of the XML document and shows little variance.

7 Summary

We have presented a novel algorithm for handling backward and forward XPath axes in a streaming fashion. Our experiments reveal that significant performance benefits can be obtained by using the $\chi\alpha o\varsigma$ algorithm for evaluating XPath expressions on XML documents in a streaming fashion. Furthermore, $\chi\alpha o\varsigma$ has significantly lower storage requirements. The ideas presented in this paper can be applied to other XPath

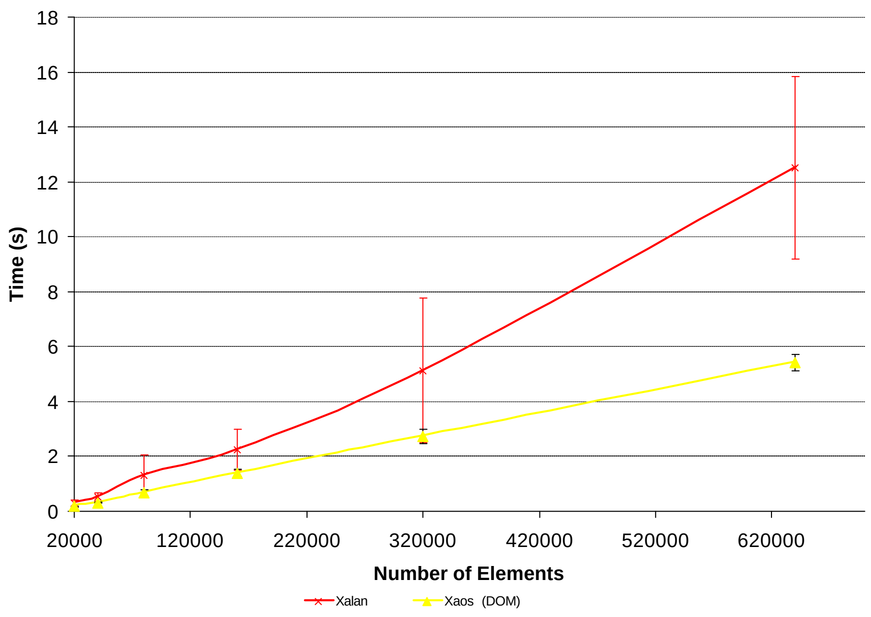


Figure 7: Searching Time in seconds: $\chi\alpha o\varsigma$ versus Xalan

processors. For example, TurboXPath is currently integrating our algorithm into their system. We are working on extending the $\chi\alpha o\varsigma$ engine to handle more of XPath, including predicate evaluation, position and count functions, etc., building on the framework we have described in this paper.

References

[1] M. Altinel and M. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *Proceedings of VLDB*, Cairo, Egypt, September 2000.

[2] Apache.org. Xalan XSLT stylesheet processor. http://xml.apache.org.

[3] Apache.org. Xerces XML parser. http://xml.apache.org.

[4] T. Bray, J. Paoli, C. Sperberg-McQueen, and E. Maler. Extensible Markup Language (XML) 1.0 (second edition). Technical report, W3C Recommendation, October 6 2000. http://www.w3.org/TR/REC-xml.

[5] C.-Y. Chan, P. Felber, M. Garofalakis, and R. Rastogi. Efficent filtering of XML documents with XPath expressions. In *18th International Conference on Data Engineering*. IEEE, February 2002.

[6] J. Chen, D. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *Proceedings of SIGMOD 2000*, pages 379–390, 2000.

[7] J. Clark. XSL transformations (XSLT) version 1.0. Technical report, W3C Recommendation, November 16 1999. http:/www.w3.org/TR/xslt.

[8] J. Clark and S. DeRose. XML path language (XPath) version 1.0. Technical report, W3C Recommendation, November 16 1999. http://www.w3.org/TR/xpath.

[9] Y. Diao, P. Fischer, M. Franklin, and R. To. Yfilter: Efficient and scalable filtering of XML documents. Demo at ICDE, February 2002.

[10] M. Fernandez, J. Marsh, and M. Nagy. XQuery 1.0 and XPath 2.0 data model. Technical report, W3C Working Draft, April 30 2002. http://www.w3.org/TR/xquery-datamodel.

[11] A. L. Hors et al. Document object model (DOM) level 3 core specification. Technical report, W3C Working Draft, April 9 2002. http://www.w3.org/TR/DOM-Level-3-Core.

[12] V. Josifovski, M. Fontoura, and A. Barta. Enabling relational engines to query XML streams. IBM Internal publication.

[13] J. Melton et al. XML related specifications (SQL/XML). Technical report, ISO/ANSI Working Draft, June 2001. http://www.sqlx.org/.

[14] A. Schmidt et al. The XML benchmark project. Technical Report INS-R0103, CWI, Amsterdam, Netherlands, April 2001.

[15] A. Tozawa and M. Murata. Tableau construction of tree automata from queries on structured documents. IBM Internal publication.

[16] R. Whitmer. Document Object Model (DOM) level 3 XPath specification version 1.0. Technical report, W3C Working Draft, March 28 2002. http://www.w3.org/TR/DOM-Level-3-XPath.

A Rules for Building an X-tree

We represent an $\mathcal{R}$*xp* expression as a rooted tree, called X-tree, with labeled vertices and edges, $\mathcal{T} = (V_{\mathcal{T}}, E_{\mathcal{T}})$, where the root is labeled Root. For each *NodeTest* in the expression, there is an x-node in the x-tree labeled with the nodetest. Each x-node (with the exception of Root) has a unique incoming edge, which is labeled with the *Axis* specified before the *NodeTest*. One of the x-nodes is designated to be the output x-node. There are functions, $label : V_{\mathcal{T}} \to String$, and $axis : E_{\mathcal{T}} \to \{ancestor, parent, child, descendant\}$ that return the labels associated with the x-nodes and edges respectively. An x-tree-like structure is also defined for a *RelLocationPath*. We will call this structure an x-forest because it consists of two rooted trees, one rooted at Root, and the other rooted at a special x-node labeled context, which, like Root, has no incoming edges. The structure corresponding to a *PredicateExpr* may either be an x-tree or an x-forest, but none of the x-nodes is designated as an output x-node.

The following rules can be used inductively (based on the structure of the $\mathcal{R}$*xp*) to build a x-tree from an $\mathcal{R}$*xp*.

$Step ::= Axis :: NodeTest$ The x-forest for $Step$ contains three x-nodes labeled Root, context, and $NodeTest$ (designated as the output node), and an edge from context to $NodeTest$ labeled $Axis$.

$Step ::= Step_1\ '['\ PredicateExpr\ ']'$ Let T_1 refer to evaluatethe x-forest resulting from $Step_1$, and T_2 refer to the x-forest or x-tree resulting from $PredicateExpr$. The x-forest for $Step$ is obtained by merging the output x-node of T_1 with the context x-node of T_2 (if any), and merging the root x-nodes of T_1 and T_2. The output x-node of T_1 is designated as the output x-node of the resulting x-forest.

$RelLocationPath ::= Step\ '/'\ RelLocationPath_1$ Let T_1 and T_2 refer to the x-forests obtained from $Step$ and $RelLocationPath_1$ respectively. The x-forest for $RelLocationPath$ is obtained by merging the output x-node of T_1 with the context x-node of T_2, merging the root x-nodes of T_1 and T_2, and designating the output x-node of T_2 as the output x-node of the resulting x-forest.

$PredicateExpr ::= RelLocationPath\ and\ PredicateExpr_1$ Let T_1 refer to the x-forest obtained from *RelLocationPath* and T_2 refer to the x-tree or x-forest obtained from $PredicateExpr_1$. The x-forest for $PredicateExpr$ is obtained by merging the context of T_1 with the context of T_2 (if any), and merging the root x-nodes of T_1 and T_2. None of the x-nodes is designated as an output vertex.

$PredicateExpr ::= AbsLocationPath\ and\ PredicateExpr_1$ similar to the previous case.

$AbsLocationPath ::=' /' RelLocationPath$ The x-tree is obtained by merging Root and context x-nodes of the x-forest obtained from $RelLocationPath$.