An Algorithm for Streaming XPath Processing with Forward and Backward Axes

Charles Barton, Philippe Charles, Deepak Goyal, Mukund Raghavchari
IBM T.J. Watson Research Center
Marcus Fontoura, Vanja Josifovski
IBM Almaden Research Center

Published at ICDE 2003

Presented by
Jan Rittinger
27.01.03
University of Constance
Department of Computer and Information Science
0. Preface

In this paper $\text{\textasciicircum}$ - a streaming algorithm for evaluating XPath expressions - is described, which can process both forward and backward axes in a single document-order traversal of an XML document.

The first chapter contains some short background information for understanding the $\text{\textasciicircum}$ algorithm: XPath, Streaming, and Tree Model. Chapter 2 describes X-tree and X-dag, the two main concepts used in the $\text{\textasciicircum}$ algorithm. In the third chapter the $\text{\textasciicircum}$ algorithm and its Matching Structure is explained. The 4th and last chapter contains a complete example Go through.
1. Background Information

This Chapter explains some of the basics for understanding the χαος algorithm.

1.1 What is χαος?

χαος (Xaos, pronounced Chaos) is an acronym for XML Analysis, Optimization, and Stuff. χαος is a streaming algorithm for evaluating XPath expression, which can process forward axes as well as backward axes in a single document-order traversal.

1.2 XPath

XPath is a language for addressing parts of XML documents. It is an integral component for XML processing such as XSLT and XQuery. An XPath expression consists of location path steps. Each step includes an axis specifier, a nodetest and zero or more predicates. The XPath expressions are evaluated relative to a context node in the document tree. The context node for an absolute location path is always the root element.

In this paper the following subset of XPath is used:

AbsLocPath := ‘/’ RelLocPath
RelLocPath := Step ‘/’ RelLocPath | Step
Step := Axis :: NodeTest | Step ‘[‘ PredExpr ‘]’
PredExpr := RelLocPath and PredExpr | AbsLocPath and PredExpr
            | RelLocPath | AbsLocPath
Axis := ancestor | parent | child | descendant
NodeTest := String

1.3 Streaming XML

In this case streaming or event-based parsing means reading and processing XML documents by handling each element only once, producing events at element tags and other components of the document.

Figure 1 represents a streaming XPath processor. It handles events from an XML parser and matches the events on a given XPath expression. χαος is such an XPath processor.
1.4 Tree Model & Example Document

An XML document can be represented as a tree, whose nodes are the elements of the document and a virtual root element called Root, which contains the document element. To identify the nodes, every node an unique identifier and a level (the distance from root) is added. e.g. U₈,₃ in Figure 2 has the tagname U, is the 8. element and lies in the third level.

Figure 2 An XML document and the tree representation of the same document
2. Used Concepts

This Chapter explains the two main concepts of the $\chi\alpha\omega\zeta$ algorithm. Both are representations of an XPath expression. They are called X-tree and X-dag. The X-dag is a main aspect in the $\chi\alpha\omega\zeta$ algorithm. It converts the reverse axes from the XPath expression into forward axes. This conversion makes it possible to stream the XML documents. The X-tree’s function is, to map the found elements to a Matching Structure.

2.1 X-tree

The X-tree is the XPath expression translated into a rooted tree. Every nodetest in the XPath expression is a label for a vertice in the X-tree called x-node. Every x-node has an unique incoming edge (except Root), which is labeled with the axis specifier corresponding to the nodetest. Predicates are translated like location steps. One of the x-nodes is marked as output x-node.

Figure 3a is the X-tree representation of the XPath expression

/\text{descendant::Y}[\text{child::U}]/\text{descendant::W}[\text{ancestor::Z}/\text{child::V}]\), where e.g. the x-node labeled with Y has a corresponding incoming descendant edge from Root (because it’s the first node).

2.2 X-dag

The X-dag is also a rooted tree. It is obtained from the X-tree by transforming the ancestor and parent axes in the tree into descendant and child axes. The vertices in the X-tree remain the same in the X-dag. Every edge in the X-tree labeled with child and descendant is also an edge in the X-dag. The edges labeled ancestor and parent in the X-tree are reversed and relabeled with descendant and child. For any non-root x-node in the X-dag, which has no incoming edge, a descendant edge from Root is added.

Figure 3b is the X-dag corresponding to the X-tree in 3a with the XPath expression

/\text{descendant::Y}[\text{child::U}]/\text{descendant::W}[\text{ancestor::Z}/\text{child::V}]\). Here the only reverse axis ancestor::Z now has an edge from Z to W labeled with descendant and an edge descendant from Root to Z.

![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](image-url)
3. χαος Algorithm

χαος builds both the X-dag and the X-tree from the XPath expression. The X-dag is used, to find and filter the set of relevant elements in an XML document produced by an event-based parser (e.g. SAX). If an element is found, which matches an x-node from the X-tree, a Matching Structure is created. At the end of the processing the results of the XPath expression are encoded in the Matching Structure M\textsubscript{Root,Root}.

3.1 Filtering Events

Every element in an XML document produces a start and an end event. Every element which produced a start event, but not an end event is called an open element. χαος now uses the X-dag to process only the ‘relevant’ elements. An element is relevant only if it matches an x-node and its parent in the X-dag tree is an open relevant element. This means that at the start of an XML document only the elements are relevant which are reachable from Root in the X-dag. A Looking-for Set, which is maintained at every event, is created to evaluate wether the element associated with the next event is relevant. The Looking-for Set consists of pairs L(label,lev). Label stands for an element. Lev is the level, where the searched element is relevant. Lev is an integer if the axis in the X-dag is a child axis (current level + 1) and * if the axis in the X-dag is a descendant axis (* stands for every possible level). If one element in the X-dag is matched, new elements which are reachable from the matched element are added into the Looking-for Set.

e.g. in Figure 3b U is not relevant as long as there is no open element Y. If a start event for Y is seen two new entries are inserted in the Looking-for Set (U,(current level of Y)+1) and (W,*). After that insertion U is relevant in the level of Y + 1. See also Go through (Chapter 4).

3.2 Building the Matching Structure

In this phase of the algorithm, for each matching start event in the X-tree a Matching Structure is created. For each matching end event in the X-tree the Matching Structures is discarded or propagated in the appropriate upper hierarchical level. Special Cases are the reverse axes ancestor and parent. Examples are given in the Go through (Chapter 4).

3.2.1 Matching

A matching is a partial mapping from x-nodes of the X-tree V to elements of a document D, where the axis of the element in D is consistent to the axis in V. Consistent means, if v\textsubscript{1} and v\textsubscript{2} are two x-nodes in an X-tree connected by an edge x and d\textsubscript{1} and d\textsubscript{2} are two elements in a document, then (v\textsubscript{1},d\textsubscript{1}) is consistent to (v\textsubscript{2},d\textsubscript{2}) if d\textsubscript{1} and d\textsubscript{2} share the same axis x.

3.2.2 Start Event & Matching Structure

If a start event for an element e in the document matches an x-node v of the X-tree, a new Matching Structure M\textsubscript{v,e} for every matching x-node is created. A Matching Structure M\textsubscript{v,e} consists of a pointer to M\textsubscript{v,e}, a label with the tagname of v, and submatchings for every child of v in the X-tree. The submatchings of the Matching Structure are initially empty.

3.2.3 End Event & Propagation

An end event matching of an element e in the document with an x-node v of the X-tree follows a test wether it consists a Total Matching at the Matching Structure M\textsubscript{v,e} or not. A Total Matching exists either if v is a leaf in the X-tree or all submatchings of v are non-empty. At a Total Matching at M\textsubscript{v,e} the pointer of M\textsubscript{v,e} is propagated into all the appropriate parent
Matching Structures. So the Matching Structures are built recursively. If the Total Matching fails the Matching Structure $M_{v,e}$ is discarded.

### 3.2.4 Special Case: reverse Axes

The presence of ancestor and parent edges in the X-tree complicates the process. It is not clear whether there exists a Total Matching at the Matching Structure at an end event or not. A Total Matching can fail, if the ancestor or parent element in the document contains no Total Matching itself. Handling this uncertainty is not hard. The propagation process remains the same, except for x-nodes that have an incoming or outgoing edge labeled ancestor or parent. The Matching Structure $M_{v,e}$ which equals an x-node with an outgoing edge labeled ancestor or parent assumes, that the submatchings for their reverse axes are given. $M_{v,e}$ now propagates itself - if it creates a Total Matching - to the appropriate Matching Structures. If there is an incoming edge ancestor or parent the Matching Structure $M_{v,e}$ may have been propagated optimistically to its parent-matchings. If we can determine conclusively that $M_{v,e}$ cannot represent a total matching at $v$, we undo the propagation of $M_{v,e}$. The propagation is now undone recursively until a parent-matching remains a Total Matching.

### 3.3 Emit Output

At least one result is found, if the Matching Structure $M_{\text{Root},\text{Root}}$ creates a Total Matching. The Output nodes are emitted by traversing the Matching Structure and the X-tree simultaneously until the Output x-node is found. The corresponding Matching Structures are the searched result.

*In the example document from figure 2 and the X-tree and X-dag from figure 3 the following Matching Structure $M_{\text{Root},\text{Root}}$ is the result. The Output nodes are marked gray.*

![Figure 4 Matching Structure for example document Figure 2 with the X-tree & X-dag from Figure 3](image-url)
4. Go through

In this Chapter the XML document from Figure 2 is parsed with the XPath expression used for building the X-dag and X-tree example in Figure 3a and Figure 3b: /descendant::Y[child::U]/descendant::W[ancestor::Z/child::V]. For every start and end event the Looking-for Set is maintained, the Matches are shown, and a short explanation is given.

<table>
<thead>
<tr>
<th>Step</th>
<th>Event</th>
<th>Matches</th>
<th>Looking-for Set</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Start: Root(0,0) (Root,0)</td>
<td>{(Y,<em>), (Z,</em>)}</td>
<td>Y and Z (following Root in the X-dag) are now relevant elements and therefore added into looking-for set. Y and Z get the level *, because both have descendant axes.</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Start: X(1,1)</td>
<td>{(Y,<em>), (Z,</em>)}</td>
<td>element X is unknown – discarded</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Start: Y(2,2) (Y,*)</td>
<td>{(Y,<em>), (Z,</em>), (U,3)}</td>
<td>matched Y (Matching Structure for Y with 2 submatchings is created) - Y has a child-axis U in the X-dag; U is added into looking-for set with level 3 (currentlevel(2)+1)</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Start: Z(3,3) (Z,*)</td>
<td>{(Y,<em>), (Z,</em>)}, (W,*), (V,4)}</td>
<td>see above matched Z - W now can follow in any level (descendant) V in the next level (child)</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Start: V(4,4) (V,4)</td>
<td>{(Y,<em>), (Z,</em>), (W,*)}</td>
<td>matched V - no more looking for V because the next level is greater than 4</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>End: V(4,4) (V,4)</td>
<td>{(Y,<em>), (Z,</em>), (W,*)}, (V,4)}</td>
<td>V is a leaf in the X-tree consequently a total matching – V(4,4) is propagated to Z(3,3)</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Start: V(5,4) (V,4)</td>
<td>{(Y,<em>), (Z,</em>), (W,*)}</td>
<td>see step 5</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>End: V(5,4) (V,4)</td>
<td>{(Y,<em>), (Z,</em>), (W,*)}, (V,4)}</td>
<td>see step 6; now Z(3,3) has two submatchings</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Start: W(6,4) (W,*)</td>
<td>{(Y,<em>), (Z,</em>), (W,*)}</td>
<td>…</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Start: W(7,5) (W,*)</td>
<td>{(Y,<em>), (Z,</em>), (W,*)}</td>
<td>…</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>End: W(7,5) (W,*)</td>
<td>{(Y,<em>), (Z,</em>), (W,*)}</td>
<td>W has an outgoing backward constraint. The submatchings are propagated and W now contains a total match. W(7,5) is propagated to Y(2,2)</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>End: W(6,4) (W,*)</td>
<td>{(Y,<em>), (Z,</em>), (W,*)}</td>
<td>see step 11</td>
<td></td>
</tr>
</tbody>
</table>
13 End: Z(3,3) (Z,*) {(Y,*)} (Z,*) (U,3)}
Z has an incoming ancestor edge - all submatchings in Z are satisfied, so no clean up is necessary and Z is propagated into W(6,4) and W(7,5).
After the Z end event it is impossible to see a W, that is in the result. W is removed from the Looking-for Set.

14 Start: U(8,3) (U,3) {(Y,*)} (Z,*)
15 End: U(8,3) (U,3) {(Y,*)} (Z,*) (U,3)}
U is a leaf in the X-tree consequently a total matching – U(8,3) is propagated to Y(2,2)

16 End: Y(2,2) (Y,*) {(Y,*)} (Z,*)
all submatching in Y(2,2) are satisfied and Y(2,2) is propagated to Root

17 Start: Y(10,2) (Y,*) {(Y,*)} (Z,*) (U,3)}
18 Start: Z(11,3) (Z,*) {(Y,*)} (Z,*) (W,*) (V,4)}
19 Start: W(12,4) (W,*) {(Y,*)} (Z,*) (W,*)
20 End: W(12,4) (W,*) {(Y,*)} (Z,*) (W,*) (V,4)}
W has an outgoing backward constraint. The submatchings are propagated and W(12,4) now contains a total matching. W(12,4) is propagated to Y(9,2)

21 End: Z(11,3) (Z,*) {(Y,*)} (Z,*) (U,3)}
Z has an incoming ancestor edge - but the submatching is empty and clean up is recursively necessary (submatchings in W(12,4) and Y(9,2) are now empty again)

22 End: Y(10,2) (Y,*) {(Y,*)}
no match is reported

23 End: X(1,1) {(Y,*)} (Z,*)
24 End: Root(0,0) (Root,0) {(Root,0)}
one match at root - at least one matching element is found