Testing Intersection of XPath Expressions under DTDs

Stefan Böttcher
University of Paderborn
Faculty 5 (Computer Science, Electrical Engineering & Mathematics)
Fürstenallee 11, D-33102 Paderborn, Germany
stb@uni-paderborn.de

Abstract

XPath expressions play a central role in both querying XML databases and searching for XML data which is distributed over the web. Our work focuses on XPath query optimization for XML data, which is distributed over several nodes in a database or several nodes on the web, and which follows a single global schema defined by a DTD, i.e. multiple nodes contain fragments of a global ‘virtual’ XML document. Whenever the accessible XML content stored in a node is described by an XPath expression, it may be considerably advantageous to search for XML data only on those nodes which contain a relevant fragment of the distributed ‘virtual’ XML document. To enable this optimization, we present an intersection test for XPath expressions that works on the DTD and on the XPath expressions alone, i.e. it does not require any access to XML fragments. The key idea is to construct a graph containing all paths selected by one XPath expression and to show that none of these paths can be selected by the other XPath expression in any document that is valid according to a given DTD.

1. Introduction

1.1 Problem origin and motivation

During the last few years, XML has become one of the major data storage formats within both databases and the web. The standards around XML include DTD and XPath: DTDs characterize the subset of valid XML documents, and XPath expressions are used for queries and searches, but also within transformers like XSL(T) and other query languages e.g. XQuery. Whenever XML data is accessed or transformed (say by XSL or XQuery), the searched or accessed part of an XML document can be characterized by an XPath expression.

We focus on XML data and XML documents that are distributed over several nodes in a distributed XML database or on the web, but follow a single global DTD, i.e. the distributed XML fragments are considered to be a part of a global ‘virtual’ XML document or database. We assume that data distribution can be described by publicly known XPath expressions in the following sense. Firstly, each node chooses one or more XPath expression(s) XP1 that describe all the content of this node. Secondly, whenever the data stored on a node is changed in such a way that the node contains XML data which is not described by its XPath expression(s) XP1, the XPath expression(s) summarizing the content of the node is (or are) modified and published to the other nodes.

We decide whether or not a node in a distributed XML database or in the web must be searched, based only on XP1 and an XPath expression XP2 used within a query or within an XSL style-sheet or for searches. Whenever our intersection tester can prove that the content of a database or web node described by an XPath expression XP1 has an empty intersection with the node set selected by XP2, we are then sure that the following optimizations can be applied. First, the fragment needs not to be searched for answers to a query using XP2. Second, when an XSLT template using an XPath expression XP2 is processed, the fragment described by XP1 needs not to be searched for matching nodes.

A different application, which also profits from such an intersection tester, is the optimization of XSLT document processing as discussed in [11]. Within this contribution, we focus on the intersection tester itself, whereas some of the optimization opportunities of XML data access are discussed in [4] and [11].

1.2 The supported subset of XPath expressions

The subset of XPath expressions supported in this paper matches a subset of core XPath as defined in [9]. We define this subset by the following EBNF grammar:

\[
\begin{align*}
\text{exp} & ::= \text{locationpath} | '/' \text{locationpath} \\
\text{locationpath} & ::= \text{locationstep} ('/' \text{locationstep})^* \\
\text{locationstep} & ::= x '::' t ( [']' \text{pred} ['] )^* \\
\text{pred} & ::= \text{pred} \text{and} \text{pred} | \\
& \quad \text{pred} \text{or} \text{pred} | \text{not}'(\text{pred} ') | \text{exp}
\end{align*}
\]

Thereby, ‘x’ is a shortcut for one of the following axis specifiers: self, child, descendant, descendant-or-self and attribute, and when ‘x’ occurs in the scope of a predicate ‘pred’, ‘x’ can also be the parent-axis, the ancestor-axis or the ancestor-or-self-axis. Furthermore, ‘t’ is a node test (either an XML Tag name or a wildcard ‘*’). Different from [9], we forbid the axis specifiers following, preceding, following-sibling and preceding-sibling.

Note that within this subset ‘[pred1] [pred2]’ and ‘[pred1 and pred2]’ are equivalent because the order of sibling nodes does not matter within our subset of XPath.
1.3 Used terminology and problem definition

Within this section, we use the XPath expression XP=E1//E2/E1/E2/E3 to explain some terms which are used in the remainder of our presentation. The nodes selected by XP (in this case, nodes with element name ‘E3’) can be reached from the root node by a path which passes at least the nodes E1, E2, E1, E2 and E3 (in this order). These paths are called paths selected by XP or selected paths for short.

Every path selected by XP must start with the root (‘/’) directly followed by an element E1, and must contain an element E1 which is directly followed by an element E2 which is itself directly followed by E3. We call a list of elements which are ‘connected’ by child-axis location steps in XP (e.g. E1/E2/E3), an element sequence of XP. We also use the term element sequence for a single element (like E2 within //E2/), i.e. an element where neither the location step before it nor after it are child-axis location steps. Thus, /E1, E2 and E1/E2/E3 are the element sequences of this example, where /E1 can be regarded as a shortcut for root/E1.

The input of our tester consists of the root (‘/’) and the two XPath expressions (XP1 and XP2) and a DTD which restricts the set of all valid XML documents. Our tester has to decide based on this input alone, i.e. without any access to the XML fragment stored on the database node or on a web site, whether or not the intersection of the node set selected by XP1 and of the node set selected by XP2 is empty for all the documents which are valid according to the given DTD.

Section 2 describes how we construct a graph that contains all the paths selected by the XPath expression XP1. Section 3 describes how a query XPath expression XP2 is used to search a filter compatible path within such a graph, and that the intersection of two XPath expressions is empty if no such filter compatible path is found. Finally, Section 4 compares our work related work and contains the summary and conclusions.

2. A graph containing paths selected by XP1

In order to construct an XP1 graph which contains all the paths to nodes selected by an XPath expression XP1, we introduce the concept of DTD filters and the concept of a DTD graph.

2.1 Computing constraints and filters from a DTD

Each DTD defines a set of constraints on valid XML documents [3], e.g. the following DTD (Example 1) defines not only the elements Root, E1, E2 and E3, but also a set of constraints, which we express as filter implications, i.e. implications between filter expressions:

1. self::Root ⇒ /E1 and unique(/E1) and *∈ {E1}
2. self::E1 ⇒ *∈ {E2}
3. self::E2 ⇒ /E1 and unique(E1) and /E3 and *∈ {E1,E3}
4. self::E3 ⇒ not ∗

The first line of the DTD requires that each Root element has exactly one child element with the node name E1. We express this with the first filter implication, and we will use this filter implication to generate a filter for all Root elements. Within this filter implication, ‘/E1’ states that each Root element within each valid document has at least one child element E1. Similarly, ‘unique(E1)’ expresses the functional constraint [15] that each Root element has at most one child element E1. And *∈ {E1} expresses that every child of a Root node has the node name E1. We use unique(/E1) and *∈ {E1} as shortcuts for XPath expressions, e.g. *∈ {E1,E3} is a shortcut for ‘not *(./nodename()!=E1 and ./nodename() != E3)’.

The second line of the DTD states that every E1 element can only have children with the node name E2. The second filter implication is applied to every node with the node name E1, i.e. given that self::E1 is true, we use the implication to conclude that all successor nodes must have the node name E2. The third line of the DTD requires each node E2 to have exactly one child E1 and at least one child E3 and no other children except E1 and E3 children. This is expressed by the third filter implication. The fourth line of the DTD states that an E3 node has no children, which is expressed by the fourth filter implication.

Whenever the DTD defines an element A using an expression B, we transform this definition into a filter implication ‘self::A ⇒ DTDA’ which is applied to each element A on a path selected by an XPath expression. The following table summarizes some examples of how the filter [DTDA] is defined depending on the syntax of B.

<table>
<thead>
<tr>
<th>B</th>
<th>[DTDA]</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1*</td>
<td>[ *∈ {E1} ]</td>
</tr>
<tr>
<td>E1+</td>
<td>[ /E1 and *∈ {E1} ]</td>
</tr>
<tr>
<td>E1</td>
<td>[ /E1 and *∈ {E1} and unique(/E1) ]</td>
</tr>
<tr>
<td>E1?</td>
<td>[ *∈ {E1} and unique(/E1) ]</td>
</tr>
<tr>
<td>E1/E2</td>
<td>[ (./E1 exclusive-or ./E2) and unique(/E1) and unique(/E2) and *∈ {E1,E2} ]</td>
</tr>
<tr>
<td>E1,E2</td>
<td>[ /E1 and /E2 and unique(/E1) and unique(/E2) and *∈ {E1,E2} ]</td>
</tr>
</tbody>
</table>

Table 1: Transformation of the DTD into DTD filters.
2.2 Embedded reduced DTD graphs

Furthermore, we follow an idea presented in [2] and use the DTD in order to construct a so called DTD graph. A DTD graph is a directed graph $G=(N,V)$ where each node $v \in V$ corresponds to an element of the DTD and an edge $v \in E1$, $v=E1, E2$ from E1 to E2 exists for each element E2 which is used in order to define the element E1 in the DTD. This DTD graph contains all the paths which may occur within any XML document which is valid according to the DTD. The reduced DTD graph contains all the paths which may occur on at least one path from E1 to E2 in the DTD. This DTD graph is embedded within the left E1 element and the right E2 element through the use of two 0-edges, as shown in the example of Figure 2.

![Figure 1: DTD graph of Example 1.](image)

![Figure 2: Embedded reduced DTD graph for the location step E1/descendant-or-self::E1.](image)

An edge (E1,0,E1) within a path represents the fact that within each document both nodes E1 in the path represent the same element E1 of the document.

2.3 A graph for paths selected by the expression XP1

We transform the XPath expression XP1 into a graph that contains both all the paths selected by XP1 and the filters that apply to the paths, and call this graph the XP1 graph for the remainder of the paper [2]. Each element occurring in a location step of XP1 is represented by a node in the XP1 graph. Whenever the XP1 location step associates a filter to this element, the filter is attached to the node which represents the element in the XP1 graph.

The following Algorithm 1 (which extends an algorithm given in [2]) computes the XP1 graph from a given DTD graph and an XPath expression XP1:

```java
1. GRAPH GetXP1Graph(DTD G, XP1)
2. (GRAPH X1P1Graph=NEW GRAPH(DTD.GetRoot())
3. NODE lastNode = DTD.GetRoot();
4. while(not XP1.IsEmpty()) {
5. LOCATION STEP step =
6. XP1.RemoveFirstLocationStep();
7. NODE node = NEW NODE (step.nodeTest);
8. X1P1Graph.AddNodeAndFilters(node, step.filters);
9. if (step.axisSpecifier = '/')
10. XP1Graph.AddEdge(lastNode, 1, node);
11. else
12. X1P1Graph.AddEmbeddedReducedDTDGraph{
13. lastNode = node;
14. }
15. markAsEndNode(node);
16. return X1P1Graph;
```

![Figure 4: Algorithm 1: Computation of the XP1 graph from an XPath expression XP1 and the DTD graph.](image)

At first, Algorithm 1 generates an XP1 graph node which represents the root-element of each document (line (2)). Thereafter (lines (4)-(13)), Algorithm 1 processes one location step of XP1 after the other (line (5)) and thereby generates the XP1 graph as follows. A new XP1 graph node is generated for the node test of each location step of XP1 (line (6)), and this XP1 graph node is added to the XP1 graph together with the filters for this location step (line (7)). Whenever the actual location step is a child-axis location step (lines (8)-(9)), Algorithm 1 inserts an edge with distance label 1 between the last node and the most recently inserted node in the XP1 graph. Otherwise, the location step is a descendant-axis or descendant-or-self-axis location step, and Algorithm 1 embeds a reduced DTD graph between the last node and the most recently inserted node (line (11)). When graph nodes and filters have been generated for all location steps, Algorithm 1 marks the most recently generated node as an ‘End’ node and returns the generated XP1 graph (lines (14)-(15)).

Example 3: Consider the DTD graph of Example 1 and an XPath expression XP1 = /Root/E1/descendant-or-self::E1[@a]/ descendant-or-self::E1[@b] // E3 [.../@c].

Figure 3 (below) shows the XP1 graph for XP1. We have added the node IDs 0,...,11 in order to refer to a certain node of the XP1 graph.
The node name tests of XP1 (i.e. Root,E1,E1,E1,E3) are represented by the nodes with the IDs 0, 1, 4, 7 and 11, and the filters [@a], [@b] and [@c] are attached to the nodes with the IDs 4, 7 and 11 respectively. The first descendant-or-self location step is represented by the reduced DTD graph with the node IDs 2 and 3, and the reduced DTD graph is embedded between the node with ID 1 and the node with ID 4. Similarly, the reduced DTD graph representing the second descendant-or-self location step has the node IDs 5 and 6 and is embedded between the nodes with the IDs 4 and 7. Finally, the descendant-axis location step (//) is represented by the reduced DTD graph with the node IDs 8, 9 and 10, and which is embedded between the nodes with the IDs 7 and 11.

The edges with distance 0 are used in order to guarantee that every path from the Root node (ID 0) to the node E3 (ID 11) applies the filters in the correct order. For example, the filter [@a] may be applied to the same element E1 as the filter [@b] (e.g. on the path 0->1->2->4->5->7->8->9->10->11), the distance from node 4 to node 7 is 0, i.e. both nodes refer to the same element E1). Alternatively, the filter [@a] may be applied to an element E1 which occurs prior to the element E1 to which the filter [@b] is applied (e.g. on the path 0->1->2->4->5->6->7->8->9->10->11, the distance from node 4 to node 7 is 1+1, i.e. both nodes refer to different elements E1). However, the filter [@a] is never applied to an element E1 which occurs after an element E1 to which the filter [@b] is applied.

### 2.4 Node name filters and DTD filter implications

There are further constraints that apply to each node in the XP1 graph, which can be expressed through the use of node name filters and DTD filter implications. A node name filter simply requires each node with the label E1 to satisfy the additional filter condition [self::E1]. Therefore, the XP1 graph shown in Figure 3 is augmented by one node name filter for each node. Furthermore, the DTD filter implications apply, i.e. they induce additional filters.

For example, each of the nodes with the IDs 1,2,4,5,7,8 gets a node name filter [self::E1], and because of the second filter implication of Example 1, each of these nodes gets an additional filter [*e E2].

### 3. Path search in the XP1 graph

The XP1 graph contains all the paths that can be selected by XP1 in any document. Within the XP1 graph, we search for a path which can be taken by XP2 in a valid document and which is filter-compatible to all the element sequences of XP2. We conclude that the intersection of XP1 and XP2 is empty for each valid document, if such a filter-compatible path does not exist in the XP1 graph.

#### 3.1 Correspondence between an XP1 graph path and an XP2 element sequence

We define the manner in which XP2 elements correspond to XP1 graph nodes as follows. An XP1 graph node and a node name test which occurs in an XP2 location step correspond to each other, if and only if the node has a label which is equal to the element name of the location step or the node name test of the location step is .

We say, a path (or a node sequence) in the XP1 graph and an element sequence of XP2 correspond to each other, if the following holds:

1. The first node of the path corresponds to the first element of the sequence.
2. The th node of the path corresponds to the th element of the sequence, if the (n-1)th node corresponds to the (e-1)th element and the edge from the (n-1)th node to the nth node has the edge distance label 1.
3. The th node of the path corresponds to the th element of the sequence, if the (n-1)th node corresponds to the th element and the edge from the (n-1)th node to the nth node has the edge distance label 0.

For example, the path 0->1->2->4->5 within the XP1 graph given in Figure 3 corresponds to the element sequence Root/E1, because the element Root corresponds to the node with ID 0 and the element E1 corresponds to the nodes with the IDs 1,2,4, and 5 within this path.
3.2 A filter compatibility test for a path in the XP1 graph and a corresponding XP2 element sequence

Within this section, we consider only one XP2 sequence E1[F1]/…/En[Fn] and only one path in the XP1 graph which starts at a given startNode and which corresponds to the sequence.

Within both the given path in the XP1 graph and the XP2 sequence, we right-shuffle predicate filters as far as possible, i.e. into the final location step of the XP2 sequence and into the last corresponding node of the XP1 graph respectively.

The shuffling of a filter [F] by one child-axis location-step to the right involves attaching the filter [.J/F] to the next location step. For example, an XPath expression XP2=//E1[@b]/E2.[@a] is transformed into an equivalent XPath expression XP2=//E1/E2.[@b and J[@a].

When a filter [F] which is attached to a node N is right-shuffled along a path in the XP1 graph to a successor node N2 of N, we have to distinguish two cases. If the filter [F] is right-shuffled along an edge (N,1,N2), N2 gets a filter [.J/F]. However, if the filter [F] is right-shuffled along an edge (N,0,N2), N2 gets the same filter [F] that N has.

The filter compatibility test is performed on these right-shuffled XP1 filters [f1] and XP2 filters [f2] which are attached to the last element of the given sequence and the last node of the corresponding path respectively. The filter compatibility test returns that the filters [f1] and [f2] are incompatible (i.e., the intersection of the node set selected by [f1] and the node set selected by [f2] must be empty), if and only if the formula ‘f1 and f2’ is unsatisfiable.

In order to check whether or not a formula ‘f1 and f2’ is unsatisfiable, any predicate tester which extends the Boolean logic to include features of XPath expressions can be used (e.g. the tester presented in [4]). A predicate tester for such formulas has to consider for example, that the combination of two filters [f1]=[.J[@a="abc"] and [f2]=[not .J[@a] is unsatisfiable, because [not .J[@a] is equivalent to [not .J[@a="abc” and not .J[@a]="abc"]

Whenever a path of the XP1 graph corresponds to an XP2 element sequence and the filter compatibility test returns that the right-shuffled filters are compatible, we say that the path and the element sequence are filter-compatible.

3.3 Intersection test algorithm for XPath expressions

The following main algorithm (c.f. Figure 4 below) summarizes our test as to whether or not two XPath expressions can overlap. Basically, it checks whether or not at least one path of the XP1 graph may contain all the sequences of XP2 in the correct order and in such a way that the filters of the path and XP2 are compatible. The algorithm places one sequence of XP2 after the other from left to right (lines (3)-(11)).

A variable called startSet contains the set of all the nodes to which the actual XP2 sequence can be applied. This variable startSet is initialized with a set which contains only the root node of the XP1 graph (line (2)). Each call of the procedure removeFirstSequence() returns and removes the first element sequence of XP2 (line (4)).

Each call of the procedure reachedBy(sequence, startNode,XP1Graph) computes the set of all the nodes which are reached by sequence from a given startNode within the XP1 graph. A node can be reached by a sequence, if and only if a filter-compatible path from the startNode to this node exists. The variable reachedSet is used within lines (5)-(7) in order to collect all nodes which are reached by the actual sequence of XP2 from any startNode which is contained in startSet, i.e. the given set of start nodes.

Whenever the element sequence which has just been placed is not the last element sequence of XP2, i.e. the remainder of XP2 is not empty (line (8)), the next sequence can be placed at a node which has been reached (i.e. which is contained in reachedSet) or at any successor node of a reached node in XP1 graph. The set of nodes where the next sequence can be placed (i.e. startSet) is computed by the function call reachedSet.plusAllSuccessors(XP1Graph) (line (10)).

After all sequences of XP2 have been placed, we only have to check whether or not the set of reached nodes (reachedSet) contains the node marked as the ‘End’ node of the XP1 graph (line(12)). The result of this check will be returned as the result of our function canOverlap.

(1) Boolean canOverlap(GRAPH XP1Graph,XPATH XP2)
(2) {startSet := SetOf( XP1Graph.getRoot() ) ;
(3) while ( XP2.isNotEmpty() )
(4) {sequence = XP2.removeFirstSequence();
(5) reachedSet = emptySet();
(6) for each startNode in startSet
(7) reachedSet = reachedSet U reachedBy(sequence,startNode,XP1Graph);
(8) if ( XP2.isNotEmpty() )
(9) startSet =
(10) reachedSet.plusAllSuccessors(XP1Graph);
(11) };
(12) return ( reachedSet.containsEndNode() );
(13) }

Figure 4: Main Algorithm: the complete overlap test

Whenever a call to this function returns false, we then know that the intersection of the nodes selected by both XPath expressions must be empty for every valid document, and we can apply the optimization to searches, query processing or XSLT style-sheet processing.
4. Related work, summary and conclusions

We have developed an intersection test which checks whether or not two XPath expressions XP1 and XP2 select disjointed node sets from every XML document that is valid according to a given DTD. Our contribution to intersection tests of XPath expressions is related to other contributions to query containment tests for XPath [6, 12, 13, 15] and semi-structured data [5, 8] and to equivalence and simplification of XPath expressions [1, 14].

Similar to [1, 6, 12, 13, 14, 15], we only use the XPath expressions and the DTD for our intersection test. However, unlike [6, 12, 13, 15] which focus on decidability and upper and lower bounds of the complexity of the containment problem for XPath expressions, we focus on the intersection test and present an incomplete tester which can efficiently determine for a large subset of XPath expressions is empty for every document which is valid according to a given DTD. While the contributions of [12, 13, 15] use tree containment problem for XPath expressions, we focus on the intersection of two XPath expressions which is similar to the data guards presented in [10] or to the automata as used in [7]. Our XP1 graph construction is an extension of graphs constructed in [2] by 0-edges in order to support descendant-or-self-axis location steps within XPath expressions and by node name filters like those presented in [3]. However, unlike [3] that translates two XPath queries into two similar graphs, we take an asymmetric approach (by transforming one query into a graph within which we search for filter-compatible paths).

In comparison to all the other contributions, our contribution combines a transformation of the DTD into a set of filter implications with the construction of an XP1 graph (including 0-edges and node name filters) and the search for paths that are filter-compatible to the element sequences of XP2.

Whenever our intersection tester proves that a path in the XP1 graph which is filter-compatible to the XP2 sequences does not exist, we are sure that the intersection of the node sets selected by both XPath expressions is empty for every valid XML document. As our intersection test can use any extension of a Boolean logic tester which obeys the special rules for XPath, we can choose the power of an extended Boolean logic tester, depending on the run-time and completeness requirements of the concrete application, i.e. we can use a fast but rather incomplete tester (e.g. [4]) for the query optimization (or search or XSL transformation) on small web sites or database fragments, whereas we can use a rather complete tester for query optimization (or search or XSL transformation) on large fragments.

Finally, the results presented here are not just limited to DTDs, but can be extended in such a way that they also apply to XML schema, which is one focus of our future work.

References