Optimized Internet Search Based on an Intersection Test for XPath Expressions under a DTD

Stefan Böttcher, Rita Steinmetz
University of Paderborn, Faculty 5 (Computer Science, Electrical Engineering & Mathematics)
Fürstenallee 11, D-33102 Paderborn, Germany

Abstract
Whenever XML data is distributed over several web sites but follows a global schema defined by a DTD, i.e. multiple sites contain fragments of a global ‘virtual’ XML document, and the accessible XML content stored on a site can be described by an XPath expression, it may be considerably advantageous to search for XML data only on those sites which contain a relevant fragment of the distributed XML document. Our optimization uses the XPath expression which describes the XML content of a web site and the XPath expression given in a search query in order to decide, whether or not the site has to be searched for relevant XML data. This is done by testing whether or not both XPath queries select disjointed node sets. The key idea is to transform each of the XPath expressions into a graph which contains all paths selected by the XPath expression. Thereafter, an intersection graph is constructed for a pair of XPath expressions. Within this intersection graph, we search for a path which is compatible to all the filters attached to the XPath expressions.

Keywords: XML, XPath, DTD, query optimization, search, intersection test.

1. Introduction
1.1. Problem origin and motivation
Within the last few years, there has been a growing interest in using XML as data representation and data exchange format for web-based applications. XML has come with further standards as e.g. DTDs for describing the subset of valid XML documents and XPath as query language.

While XPath is widely used as a standard to search for nodes in a given XML document, our focus is to support the efficient evaluation of XPath queries over several XML fragments distributed over different sites of the web.

The development of our XPath intersection test was motivated by the development of a search engine for distributed XML data sources. The search engine takes advantage of XPath expressions which are provided for each site and describe the content of the XML fragment stored on this site. Note however that we assume that all the XML fragments distributed over multiple sites belong to one huge distributed virtual XML document, i.e. all the fragments follow a commonly defined DTD and can be queried as one single XML document.

The standard approach to search and query processing in XML data sources distributed over multiple sites in the web is to submit an XPath query to all the sites containing a fragment of the distributed virtual XML document. Our approach is to check in advance whether or not a site may contain XML data which is relevant to a given query and to avoid submitting a query to this site wherever possible.

We present an intersection test for two XPath expressions and a DTD. Whenever our intersection tester returns that an XPath expression used in a query does not overlap with the XPath expression describing the XML fragment of the web site, it is not necessary to query the fragment. Such an intersection test can be performed independently of the actual state of the XML data stored on the web site and prior to query processing. Therefore, the intersection tester can be used to determine in advance which sites may contain XML data fragments relevant to a query. We consider this to be a considerable advantage in comparison to querying all fragments of the distributed virtual XML document.

Our goal is to prove based on the knowledge of the DTD, without any access to the data stored on the web site, that two XPath expressions XP1 (which describes the content of a web site) and XP2 (i.e. the query) select disjointed node sets, i.e. the intersection of node sets selected by XP1 and XP2 is empty.

Due to the fact that we need a fast tester, we allow our intersection tester to be incomplete in the following sense. If our tester returns false, the XPath expressions XP1 and XP2 definitely ask for disjointed node
sets. We are then sure that we do not need to search for XML data on this particular web site. However, we allow our tester to return true, although the two XPath expressions do not select any node in common, and in this case the XML data fragment on the remote web site is searched for answers to a given XPath query.

Within this contribution, only the intersection test is described, whereas the optimization potential of XML data access is discussed in [3]. A different application which also needs an intersection tester, the optimization of XSLT document processing, is discussed in [10].

1.2. Relation to other work and our focus

Query optimization based on the disjointedness of a query and a data fragment has been proposed in several contributions (e.g. [13]) for distributed relational databases. The application of similar optimization techniques to web sites containing fragments of distributed virtual XML documents however requires an intersection test as e.g. the one presented in our paper. Our contribution to intersection tests of XPath expressions is related to other contributions to the area of intersection tests and containment tests for XPath and semi-structured data. [6,11,12,14] contribute to the solving of the containment problem for two XPath expressions under a DTD or under other given XML schema constraints. The main issues of the contributions [6,11,12,14] are the decidability and upper and lower bounds of the complexity of the containment problem for XPath expressions. We follow these contributions and examine the formulas of two XPath expressions XP1 and XP2. In contrast to these contributions, we focus on the intersection test and present an incomplete tester which can efficiently determine for a large subset of XPath expressions that given XPath expressions do not overlap under the constraints of a given DTD. Furthermore, other contributions (e.g. [11]) consider the equivalence of XPath expressions. Again a main difference is that we use an incomplete but efficient tester for disjointedness of XPath expressions.

[5] derives a set of constraints from the XPath expressions and from a DTD and provides a set of transformation rules as to how these constraints can be normalized, so that the containment problem on a small subset of XPath and on a restricted set of DTDs can be solved in polynomial time.

The contributions [11,12,14] use tree patterns in order to normalize the XPath query expressions and compare them to the XML document tree. While some of these approaches consider the DTD as a set of constraints (e.g. [14]), other approaches use a tree grammar approach and consider the DTD as an automaton [12]. In comparison to this, we extend a combination of both approaches: we transform both, the DTD and the XPath expressions, into graphs, and we translate the DTD into a set of constraints, which we use as filters in order to restrict the search for paths within the graphs. In comparison to our previous work [4], which treats both XPath expressions in a completely different way, our tester uses a symmetric approach to the intersection test, regards constraints given by the DTD and achieves a higher degree of completeness. Furthermore, our approach is similar to the transformation of an XPath query into an automaton, which was used in [7] in order to decide whether or not an XML document fulfills this query and which was used in [9] in order to process XML data streams. However, in contrast to all other contributions, our approach combines a graph based search for paths within an intersection graph with the shuffling of predicate filters in such a way, that the intersection test for XPath expressions can be reduced to an intersection test of filters on paths selected by both XPath expressions XP1 and XP2.

1.3. The supported subset of XPath expressions

The subset of XPath expressions supported in this paper matches mainly the set of core XPath as defined in [8]. This set is defined by the following EBNF grammar:

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

where \(x\) can be one of the following axis specifiers: self, child, descendant, descendant-or-self and attribute, and \(t\) stands for a node test (either an XML Tag name or a wildcard ‘*’; whereas wildcards are only allowed at the end of a path expression). \([''pred'']\) and \([''pred'']\ 'not' \(''pred'')\) are equivalent, as we exclude the sibling-axes, i.e., the order of sibling nodes is not considered. In contrast to [8] we forbid the axis specifiers following, preceding, following-sibling and preceding-sibling.

1.4. Used terminology and problem definition

Within this section, we use the XPath expression XP=//E2//E1/E3/E4 in order to explain some terms which are used in the remainder of the document. The nodes selected by XP (nodes with element name ‘E4’) can be reached from the root node by a path which passes at least the nodes E2, E1, E3 and E4 (in this order). These paths are called paths selected by XP or selected paths for short.

The input of our tester consists of the two XPath expressions (XP1 and XP2) and a DTD which restricts the set...
of 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 web site, whether or not XP1 and XP2 select disjointed node sets in all the documents which are valid according to the given DTD – or as we say: *XP1 and XP2 do not overlap*, if and only if there does never exist a path selected by XP1 and XP2 in any document which is valid according to the given DTD.

Section 2 describes the preparation steps for the intersection test, i.e., how we transform the DTD into a so called DTD graph, and how we use this DTD graph in order to compute a so called XP graph which describes all the paths selected by an XPath expression. Section 3 outlines the major parts of our intersection test, i.e., the construction of a graph for the intersection of XP1 and XP2, and the search for a path with a satisfiable combination of filters. Section 4 outlines the summary and conclusions.

**2.0. Preparing the Intersection Test**

In order to prepare both XPath expressions for the intersection test, we introduce the concept of *DTD filters* and the concept of a *DTD graph*, which we use in order to transform the XPath expressions into a graph.

**2.1. Translating the DTD into a set of constraints**

A DTD defines a set of constraints, which we translate into so called *DTD filters* which will later be attached to elements. For example (*Example 1*), consider the following DTD:

```xml
<!ELEMENT Root (E1)>
<!ELEMENT E1 (E2|E3+)>
<!ELEMENT E2 (E1)>
<!ELEMENT E3 (E4)>
```

The first line states that each element Root has exactly one child element with the node name E1. We express this as a filter [./E1 and unique(./E1) and *e {E1}]. Within this filter, ./E1 states that each Root element within each valid document has at least one child element E1. Similarly, unique(./E1) expresses the functional constraint [14] that each Root element has at most one child element E1. And *e {E1} expresses that every child of a Root node has the node name E1. We use *e {E1} as a shortcut, i.e. *e {E2,E3} is a shortcut for 'not *(./nodename()!=E2 and ./nodename()!=E3 )'.

Similarly, the second line of the DTD will be expressed by a filter [DTD1]=[self::E1 and (/E2 xor /E3) and unique(/E2) and *e {E2,E3}], where xor is the 'exclusive or'. Altogether, we get the following five DTD filters which must hold for elements Root, E1, E2, E3, E4:

-DTDRoot]=[self::Root and ./E1 and unique(/E1) and *e {E1}]],
-DTD1]=[self::E1 and (/E2 xor /E3) and unique(/E2) and *e {E2,E3}],
-DTD2]=[self::E2 and ./E1 and unique(E1) and *e {E1}],
-DTD3]=[self::E3 and ./E4 and unique(E4) and *e {E4}],
-DTD4]=[self::E4 and not ./].

Whenever a DTD defines an element A using an expression B, we transform this definition into a DTD filter [DTDA] for A. The following table summarizes some examples how [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>[*e {E1}]</td>
</tr>
<tr>
<td>E1+</td>
<td>[/E1 and *e {E1}]</td>
</tr>
<tr>
<td>E1</td>
<td>[/E1 and *e {E1} and unique(/E1)]</td>
</tr>
<tr>
<td>E1?</td>
<td>[*e {E1} and unique(/E1)]</td>
</tr>
<tr>
<td>E1/E2</td>
<td>[(/E1 xor /E2) and unique(/E1) and unique(/E2) and *e {E1,E2} ]</td>
</tr>
<tr>
<td>E1,E2</td>
<td>[/E1 and /E2 and unique(/E1) and unique(/E2) and *e {E1,E2} ]</td>
</tr>
</tbody>
</table>

**2.2. Constructing a DTD graph with filters**

Furthermore, we use the DTD in order to construct a so called *DTD graph* which contains all the paths which may occur within any XML document which is valid according to a given DTD. A *DTD graph* is a directed graph G=(N, V, M) where each node E∈N corresponds to an element of the DTD and an edge v ∈ V, 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. Furthermore, M is a mapping of DTD filters to nodes of the DTD graph (to be defined below). The DTD graph which is constructed from the DTD of Example 1 is given in Figure 1.

![Figure 1. DTD graph of Example 1](image-url)
When DTD filters are ignored, a DTD graph does not contain all the concepts which can be expressed by a DTD, e.g., it does neither distinguish between optional and mandatory elements nor between disjunctions and conjunctions expressed in the DTD. These concepts are added to the DTD graph by a mapping M which maps the previously defined DTD filters to the nodes of the DTD graph.

2.3. A graph for paths selected by an XPath expression

In order to represent the set of paths selected by an XPath expression XP, we use a given DTD graph (N,V,M) in order to construct a so called XP graph (Nodes,Edges,Map), where Nodes is a set of labeled nodes, Edges is a set of edges and Map is a mapping of filters to nodes in Nodes.

Whenever a location step of XP is of the form axis-specifier::Ei[Fi], i.e. the location step associates a filter [Fi] to the element Ei, that is represented by a node Ni∈Nodes, then ([Fi]⇒Ni) is added to Map, i.e. [Fi] is attached to Ni. If in the DTD graph a mapping ([FDTDi]⇒Ni) exists, this mapping is copied to Map. Whenever a location step in XP uses the child-axis between two elements (A/B), the XP graph connects A and B by a directed edge from A to B with distance label 1. We also use the notion (A,1,B) ∈ Edges. However, when a location step uses the descendant-or-self-axis between two elements of XP (A/B), a so called reduced DTD graph is inserted in the XP graph between the nodes which represent A and B and is connected to these nodes via directed edges with distance label ‘0’. The reduced DTD graph form E2 to E1 is a copy of the DTD graph which contains all paths from the node with label A to the node with label B.

Algorithm 1 describes the derivation of the XP graph from the DTD graph and the XPath expression XP. Each path selected by XP in any valid document corresponds to one path from the root node to one endnode of the XP graph. The XP graph contains a superset of all the paths selected by XP, because some paths contained in the XP graph may be forbidden paths, i.e. paths which have predicate filters which are incompatible with DTD constraints and/or the selected path itself. The following Algorithm 1 (which extends an algorithm given in [2]) computes the XP graph from a given DTD graph and an XPath expression XP:

```java
(1) GRAPH GETXPGRAPH(Graph DTD, XPath XP)
(2) (Graph XPGraph = new
    GRAPH(DTD.GETROOT()));
(3) Node lastGoal = DTD.GETROOT();
(4) while(not XP.ISEMPTY())
    (5) { Node goal =
        \(\text{XP.REMOVEFIRSTELEMENT}\(\));
        (6) if( XP.LOCATIONSTEPBEFORE(goal)
            == '/')
            (7) XPGraph.APPEND( Node(goal));
        else
            (8) { XPGraph.add_0_edge();
                (9) XPGraph.EXTEND(
                    (10) DTD.COMPUTEREDUCEDDTD(
                        lastGoal, goal);
                    (11) XPGraph.add_0_edge();
                )
                (12) lastGoal = goal;
            }
            (13) return XPGraph;
        }
```

Algorithm 1. Generating the XP graph

By starting with a node representing the root-element (line (2)), Algorithm 1 transforms the location steps of XP into a graph as follows. Whenever the actual location step is a child-axis location step (lines (6)-(7)), we add a new node to the graph and take the name of the element selected by this location step as the node label for the new node. Furthermore, we add an edge from the element of the previous location step to the element of the current location step with a distance of 1. However when the location step is a descendant-axis location step (A/B), Algorithm 1 attaches a reduced DTD graph to the end of the graph already generated. The reduced DTD graph, which is computed by the method call COMPUTEREDUCEDDTD(...,...), contains all paths from A to B of the DTD graph.

If XP ends with //*, i.e., XP1 takes the form XP = XP’/*, the XP’ graph is computed for XP’. Subsequently one reduced DTD graph which contains all the nodes which are successors of the end node of the XP graph is appended to the end node of the XP graph. All these appended nodes are then marked as end nodes of the XP graph. Similarly, if XP ends with /*, i.e., XP takes the form XP = XP’/*, the XP’ graph is computed for XP’. Afterwards all the nodes of the DTD graph which can be reached within one step from the end node are appended to the end node of the XP graph. Furthermore, instead of the old end node all these appended nodes are marked as end nodes of the XP graph.

3.0. Major Parts of the Test

3.1. Intersection graph construction

We use the XP graphs G1 and G2 of two XPath expressions XP1 and XP2 in order to compute an intersection graph, which summarizes all the paths contained in both XP graphs. The intersection graph P(G1,G2)=(NodesP, EdgesP, MapP) of two given XP graphs G1=(Nodes1,
Edges1, Map1) and G2=(Nodes2, Edges2, Map2), is constructed as follows.

The intersection graph contains a node for each pair of nodes from G1 and G2, which have the same node name, i.e. NodesP={(N1,N2) | N1 ∈ Nodes1 and N2 ∈ Nodes2 and N1.nodename=N2.nodename}.

The node name of a node (N1,N2) of the intersection graph is considered to be equal to N1.nodename() and N2.nodename().

The node (Root,Root) of NodesP is the start node of P(G1,G2). A node (End1,End2) of NodesP is called end node of P(G1,G2), if End1 is an end node of G1 and End2 is an end node of G2.

Edges in the intersection graph P(G1,G2) are defined based on edges in the graphs G1 and G2, where the edge distance found in G1 or G2 is taken as edge distance in P(G1,G2).

EdgesP={(((N11,N21),0,(N12,N22)) | (N11,0,N12) ∈ Edges1 AND (N21,0,N22) ∈ Edges2 OR (N11,0,N21) ∈ Edges1 AND (N21,0,N22) ∈ Edges2 OR (N11,0,N21) ∈ Edges1 AND (N21,0,N22) ∈ Edges2 AND N11=N12).}

An edge with label 1, i.e. (P1,1,P2), means that within each valid XML document the element corresponding to P2 is a child of the element corresponding to P1, whereas an edge with label 0, i.e. (P1,0,P2), means that P1 and P2 refer to the same element in each XML document.

Finally, the filters found in the graphs G1 and G2 are attached to the node of P(G1,G2) too.

MapP={[F]→(N1,N2) | [F]→N1∈Map1 OR [F]→N2∈Map2}.

Altogether the intersection graph P(G1,G2) contains all paths which are common paths in both graphs. Furthermore, it combines all filters from the XP graph G1 and all filters from the XP graph G2 and contains the combination of the filters in such a way, that a path in P(G1,G2) which is compatible to filters in P(G1,G2) always has a corresponding path in G1 (and G2) that is compatible to all the filters of G1 (and G2).

3.2. Right-shuffling filters

Filters can be right-shuffled along a selected path, which means to generate new filters for successor nodes. By right-shuffling a filter (N,[fexp]) of a node N along an edge (N1,1,N2) to a successor node N2, we get a filter (N2,[fexp]), i.e. one parent-axis step is added to the filter condition. However, right-shuffling a filter (N,[fexp]) of a node N along an edge (N1,0,N2) to a successor node N2 yields a filter (N2, [fexp]), the filter expression remains unchanged.

3.3. Searching accepted paths in the intersection graph

In general, it is not necessary to construct the whole intersection graph P(G1,G2). Instead it is sufficient to construct that part which is needed in order to compute a common filter compatible path. In other words, a path from the start node to a final node of the intersection graph is searched by step-wise path expansion using breadth first search as described in Algorithm 2. This includes the right-shuffling of filters.

Algorithm 2 uses a variable P for the paths of the intersection graph which have to be expanded and initializes P with the path that contains only the root node of the product graph (lines (2)-(3)) and the node name filter of the root node. In order to avoid infinite search within cycles, Algorithm 2 uses a variable reached to collect all the nodes, which have been reached so far, and initializes this variable with the root node of the intersection graph (line (4)). Lines (5)-(14) contain the stepwise path expansion, which is performed as long as not all paths of P are expanded and P does not contain a path to an end node. Within lines (9)-(10), each path of P is expanded by a call of the procedure reachedBy1Step (path,G1,G2), which returns the set of paths which extend the given path path by one step and are still compatible to the right-shuffled filters.

(1) SearchIntersectionGraph(G1,G2)
(2) { root=(G1.getRoot(),G2.getRoot());
(3) P = { [(root,[self::Root])] };
(4) reached = { Root };
(5) while (P≠∅ ∧ not
(6) P contains path to(End1,End2))
(7) { newPaths = {};
(8) for each path ∈P do
(9) { newPaths = newPaths U
(10) reachedBy1Step(path,G1,G2);
(11) }
(12) removeCyclePaths(reached,newPaths);
(13) P = newPaths;
(14) };
(15) if( (End1,End2)∈N )
(16) return "can overlap";
(17) else return "disjointed";
(18) }

Algorithm 2. Stepwise path expansion

A single path expansion step is carried out by the procedure reachedBy1Step:

(1) reachedBy1Step(path,G1,G2)
(2) { newPaths=∅ ;
(3) N1 = G1.follow(path) ;
(4) N2 = G2.follow(path) ;
(5) [F] = path.getFilter() ;
3.4. Testing satisfiability of filter expressions

Basically, we can use any theorem prover for first order predicate logic to check the (un)satisfiability of a filter expression (e.g. the one described in [3]), as long as the theorem prover regards the special rules of XPath. For example, such a theorem prover has to obey that a filter combination like [not @a] and [@a='7'] is unsatisfiable, and a filter [ @a='7' and @a!=7 ] is equivalent to the filter [ not @a ].

Note however that the satisfiability check of our filter formulas is restricted compared to a general purpose theorem prover for first order predicate logic which has to include arbitrary function symbols. In comparison, we do not have arbitrary function symbols, but have only the parent-child-axis instead. Similar to different function symbols which do not unify in predicate logic, two simple filter expressions preceded with a different number of parent-axis steps do not ‘unify’. Therefore, only filter conditions which may involve the same number of parent-axis steps have to be compared within our
filter tester, i.e. a filter like \[.//@a\] and not \[./@a\] is satisfiable.

4.0. Summary and Conclusions

Whenever the content of a virtual XML document is distributed over several sites in the internet, and each site provides an XPath expression, which characterizes the site's XML fragment, it may be preferable to check whether or not it is necessary to search this particular site in comparison to submitting a search query to all sites which participate on the virtual XML document. For this purpose, we have developed a tester which checks whether or not an XPath expression XP1 (which summarizes all the data of a web site) overlaps with an XPath expression XP2 (which represents a query).

In comparison to other contributions to the problem of XPath expression intersection tests, we transform the DTD into a DTD graph and DTD filters, and we derive from this graph and from each XPath expression XP the so called XP graph, a graph which contains all valid paths which are selected by XP. Two such graphs for XP1 and XP2 are used to construct the intersection graph which represents all the accepted paths which are contained in both, XP1 graph and XP2 graph. Within the intersection graph, we search for paths which are compatible with their right-shuffled filters. Path expansion stops, when it produces a filter which is unsatisfiable. Whenever no path from the root node to a final node in the intersection graph is found which has compatible right-shuffled filters, the intersection tester returns that the intersection is empty. However, when a path from the root node to a final node in the intersection graph is found which has compatible filters, the tester then assumes that the intersection is not empty. This is an assumption, as we have chosen to implement an efficient tester, which is incomplete (e.g. in the way it right-shuffles filters over cycles in the intersection graph). Whenever the tester returns that the intersection is empty, we are then sure that a web site does not have to be searched for the given query, i.e. we can perform an optimization. However, in the contrary case, we have to search within the XML fragment of this web site for answers to the given query.

The intersection test checks whether or not multiple filter expressions together are unsatisfiable. For this purpose, we can use any extension of a Boolean logic tester which obeys the special rules for XPath. This means that, depending on the concrete task, different testers for these filter expressions can be chosen: either a more powerful tester which can cope with a larger set of XPath filters, but may need a longer run time, or a faster tester which is incomplete or limited to a smaller subset of XPath.

To our impression, 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.

5.0 References: