XML Subtree Queries: Specification and Composition

Michael Benedikt and Irini Fundulaki

Bell Labs, Lucent Technologies, USA

Abstract. A frequent task encountered in XML processing is to filter an input document to produce a subdocument; that is, a document whose root-to-leaf paths are root-to-leaf paths of the original document and which inherits the tree structure of the original document. These are what we mean by subtree queries, and while they are similar to XPath filters, they cannot be naturally specified either in XPath or in XQuery. Special-purpose subtree query languages provide a natural idiom for specifying this class of queries, but both composition and evaluation are problematic. In this paper we show that for natural fragments of XPath, the resulting subtree query languages are closed under composition. This closure property allows a sequence of subtree queries to be rewritten as a single subtree query, which can then be evaluated either by a subtree-query specific evaluator or via translation to XQuery. We provide a set of composition algorithms for each common XPath fragment and discuss their complexity.

1 Introduction

In many aspects of data and document processing, one requires queries that describe a *subdocument* (*subtree*) of the document on which the queries are evaluated. For instance, a data integration application might describe a view of multiple (virtual) XML documents into a single document by specifying subtrees of the documents of each source to be merged; an access-control view might be imposed on top of this, filtering out part of the resulting integrated document for access control purposes, while an end-user query may ask for yet another subtree of the filtered view. The ultimate result delivered to the end-user is logically the composition of the three queries.

Motivation: Consider a source document D for the XMark [16] schema illustrated in Fig. 1 and suppose that a subscriber to this dataset is only permitted to see the subtree of the original document that includes information about the European region. This view is the result of query Q_1, which evaluates the XPath expression `/site/regions/europe` and closes the result set upwards and downwards. The result of this subtree query on document D is given in Fig. 1 (where the returned nodes are marked in grey).

One group of users in the subscribing company is permitted to see only the data about (i) the items that *are not associated with* a quantity, and (ii) the

G. Bierman and C. Koch (Eds.): DBPL 2005, LNCS 3774, pp. 138–153, 2005.

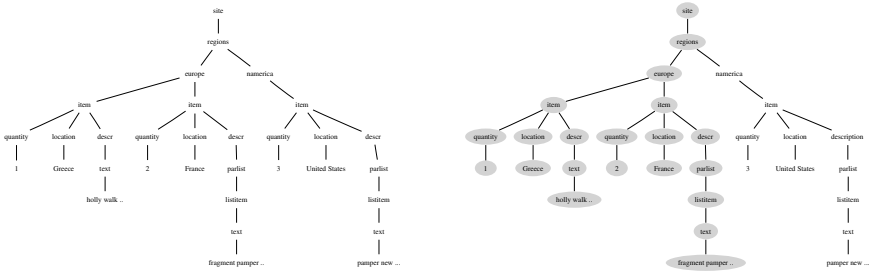

Fig. 1. XMark Document D and the subtree associated with Q_1 for D

locations of items. This subdocument is returned by query Q_2 that takes all nodes above or below the result of XPath expression: `//item[not(quantity)] | //item/location`. Finally, an end-user wants to obtain the subtree that contains the items that have either a location or a description. This subdocument is returned by evaluating query Q_3, which takes the result of XPath expression `//item[location | descr]` and closes the result set upwards and downwards.

For this family of queries, XQuery does not give a natural syntax. In fact, XQuery does not, strictly speaking, allow the expression of subtree queries at all, since in an XQuery result fresh node identifiers will always be generated. Fig. 2 shows an XQuery expression that would mimic the subtree query Q_2, matching the intended result up to the renaming of node identifiers. Observe that, even allowing a result that is correct only "up to node identifiers", the translation of XPath expressions used to specify subtree queries to the XQuery representation of these subtree queries is not straightforward: in the presence of union, we need to retain the order of the elements from the original document in the output result, while a naive translation of the subtree query would result in different parts of different components of the union being contiguously ordered in the result. The complexity of the translation escalates for subtree queries based on XPath with negation, upward axes and union.

It is fairly clear that one does not wish to write XQuery representations of subtree queries directly. An obvious solution is to use the XPath expressions themselves as the user syntax. The subtree query language is thus parameterized by a fragment of XPath and converts an XPath nodeset query into a subtree query that "filters" the input document by the XPath expression to return a subdocument.

Subtree queries can be evaluated via conversion on top of an arbitrary XQuery or XSLT processor. However, this "evaluation via translation" is unsatisfactory in terms of query performance. Since subtree queries are analogous to XPath expressions, it is natural that they be evaluated using processing techniques similar to those used for XPath. In a generated XQuery expression as in Fig. 2, however, the subtree nature of the query is hidden and cannot be exploited by the XQuery processor.

The problem is compounded when one composes subtree queries. Suppose one wishes to perform a user query like Q_3 on a chain of views given by queries Q_2

```
for $a in doc('xmark.xml')/site[regions[europe[item[self :: *[not(quantity)] |
                                                    self :: *[quantity][location]]]]] return
< site >
  {  for $b in $a/regions[europe[item[self :: *[not(quantity)] |
                                        self :: *[quantity][location]]]] return
     < regions >
      {for $cin$b/europe[item[self :: *[not(quantity)]| self :: *[quantity][location]]]return
        < europe >
          { for $d in $c/(child :: *[self : item[not(quantity)]],
                          child :: *[self :: item[quantity][location]]) return
              if ($d[self :: item[not(quantity)]]) then $d else
              if ($d[self :: item[quantity][location]]) then
              < item > { for $e in $d/location return $e } < /item > else 1} }
        < /europe > }
     < /regions > }
< /site >
```

Fig. 2. XQuery representation for Q_2

and Q_1. For efficiency and security reasons, one might wish to send the composed query $Q_3 \circ Q_2 \circ Q_1$ to the source, instead of evaluating the queries within the application or middleware. Following the "XQuery-translation approach" to subtree queries outlined above, one would compose by translating each Q_i to an XQuery expression, use XQuery's composition operator to obtain the composed query, and evaluate with a general-purpose XQuery engine. But now the fact that this composition is actually another subtree query is completely buried. What we require, then, is a composition algorithm that works *directly on the XPath surface syntax of these subtree queries*, generating a new subtree query within the same language. The resulting queries could be evaluated either via a special-purpose evaluator, or via translation to XQuery.

In this work, we deal with composition algorithms for subtree queries based on fragments of XPath 1.0 [6]. We show that for natural fragments of XPath, the resulting subtree query languages are *closed under composition*, and present the associated composition algorithms. In [2] we present experimental evidence that these composition algorithms provide benefits both when the composed queries are evaluated with a special-purpose subtree query engine, and also when they are translated to XQuery and evaluated with an off-the-shelf XQuery processor. Unlike for XQuery, composition algorithms for subtree queries are nontrivial – indeed we will show that for some choices of the XPath fragment, the corresponding subtree query language is not closed under composition at all. This is because our subtree query languages have no explicit composition operator; furthermore, since the XPath fragments we deal with are variable-free, the standard method of composition by means of variable introduction and projection is not available[1]. Our algorithms make heavy use of both the subtree

[1] Note that we do not deal with the compositionality of XPath location paths on the same document, but of the corresponding subtree queries.

nature of the queries and the restrictions imposed by the different XPath fragments.

Summarizing, the main contribution of the paper is a set of composition algorithms for each variant of the subtree query language (depending on the XPath fragment), which allow us to compose subtree queries to form new subtree queries in the same language, and also allow the composition of a subtree query with an XPath user query.

Although we study here a particular XPath-based subtree query language, we believe our composition algorithms should be applicable to other contexts in which XPath is used implicitly or explicitly to define subdocuments of documents (e.g. [1, 5]).

Organization: Section 2 presents the framework for subtree queries and the fragments of XPath we will focus on. Section 3 presents the composition algorithms and discusses complexity results. Finally we conclude in Section 4.

Related Work: Subtree queries cannot be expressed faithfully in XQuery, but they can be expressed in various XML update language proposals [17]; indeed, our work can be thought of roughly as giving special-purpose composition and evaluation algorithms for the "delete-only" subset of these languages. We know of no work dealing with the problem of efficiently composing declarative XML updates. Our framework for subtree queries was first introduced in [8]. In [15, 14] a very restricted family of these queries was studied in the context of the Bell Labs GUPster project. [14] presents translation methods for subtree queries (from this restricted family) to XQuery queries and XSLT programs.

Our work on composition stems partially from an earlier study of closure properties of XPath [4]. That work studies closure under *intersection* for XPath using the standard nodeset semantics – the emphasis is on proving/disproving closure not on algorithms or complexity bounds. In this work we deal with composition for the corresponding subtree query languages, and give effective algorithms, upper bounds, and lower bounds. Although composition is quite different from intersection (e.g. composition is not commutative), we believe that the basic ideas here can be used to give effective algorithms and bounds for the intersection problem as well. To our knowledge, composition algorithms for XQuery have been considered only in [7], where an algorithm is given for a much broader fragment of XQuery. It is not presented stand-alone, but in conjunction with query generation in a publishing setting. The algorithm of [7] would not preserve the XPath-based sublanguages we give here. Not only are these sublanguages easier to analyze and optimize than general XQuery, they are also much easier to evaluate – since our subtree queries are based on XPath 1.0, they can be evaluated with linear time combined complexity [9]. Even in the presence of data value comparisons, languages based on XPath 1.0 have low evaluation complexity [10]. The fact that XPath 1.0 is variable-free leads to the low complexity, but also makes the composition problem much more difficult than for XPath 2.0.

2 XPath-Based Subtree Queries

XPath: Subtree queries are based on XPath expressions, which describe the nodes in the input document we wish to retain in the output. XPath expressions are built up from an infinite set of labels (tags, names) Σ. The fragments of XPath[2] studied in this paper are all contained in the fragment denoted by $\mathcal{X}^{\uparrow}_{r,[\,],\neg}$ that is syntactically defined as:

$$p ::= \epsilon \mid \emptyset \mid l \mid * \mid // \mid .. \mid ..^* \mid p/p \mid p \,|\, p \mid p[q]$$

where ϵ, $\emptyset$, l in the production rules above denote the empty path ('.' in XPath), the empty set, and a name in Σ, respectively; '|' and '/' stand for union and concatenation, '$*$' and '..' for the *child*-axis and *parent*-axis, '//' and '..*' for the *descendant-or-self*-axis and *ancestor-or-self*-axis[3], respectively; and finally, q in $p[q]$ is called a *qualifier* and defined by:

$$q ::= p \mid \text{label} = l \mid q \textit{ and } q' \mid \mathsf{not}(q)$$

where p is an $\mathcal{X}^{\uparrow}_{r,[\,],\neg}$ expression and l is a name in Σ. All of the semantics of these expressions are as usual in XPath (see, for example, [18, 6]).

XPath Fragments: We use the following notations for subclasses of the XPath fragment $\mathcal{X}^{\uparrow}_{r,[\,],\neg}$: all the fragments will include the subscript '[ ]', to indicate that the subclass allows qualifiers, where qualifiers always include $q ::= p \mid \text{label} = l$.

Subscript 'r' indicates support for recursion in the fragment (the *descendant-or-self* '//' and *ancestor-or-self* '..*' axes), superscript '$\uparrow$' denotes the support for upward modality (the *parent* '..' and the *ancestor-or-self* '..*' axes in the case of subscript r) while the subscript '$\neg$' indicates that filters allow negation.

Thus the smallest class we consider is $\mathcal{X}_{[\,]}$, which has only the child axis and does not allow negation in filters. The classes $\mathcal{X}_{r,[\,]}$, $\mathcal{X}^{\uparrow}_{[\,]}$, $\mathcal{X}_{[\,],\neg}$ extend this basic fragment with the descendant and the parent axes, and negation in filters, respectively. Above these, we have the fragments $\mathcal{X}_{r,[\,],\neg}$ $\mathcal{X}^{\uparrow}_{r,[\,]}$, and $\mathcal{X}^{\uparrow}_{[\,],\neg}$ which combine two of the three features. The relationship of these fragments is shown in the left diagram of Fig. 3.

In [4] it is shown that $\mathcal{X}_{[\,]}$ and $\mathcal{X}^{\uparrow}_{[\,]}$ return the same node-sets when evaluated on the root of a tree (*root equivalence*, which we will always use here by default). It is also shown there that $\mathcal{X}_{r,[\,]}$ and $\mathcal{X}^{\uparrow}_{r,[\,]}$ are equally expressive. By similar means it can be shown that upward navigation can be removed from $\mathcal{X}^{\uparrow}_{[\,],\neg}$, to obtain an equivalent expression in $\mathcal{X}_{[\,],\neg}$: this is discussed further in Section 3.4. Thus, up to expressive equivalence, the fragments appear in the right diagram in Fig. 3. In the figure, the fragments shown at the same vertex are equally

[2] For syntactic convenience we permit nesting of unions in expressions, which is forbidden in XPath 1.0; in XPath 1.0 this would be simulated by rewriting these nested unions as top level ones.

[3] We introduce the notation '..*' for the ancestor-or-self axis here in the absence of an abbreviated syntax for this in XPath 1.0.

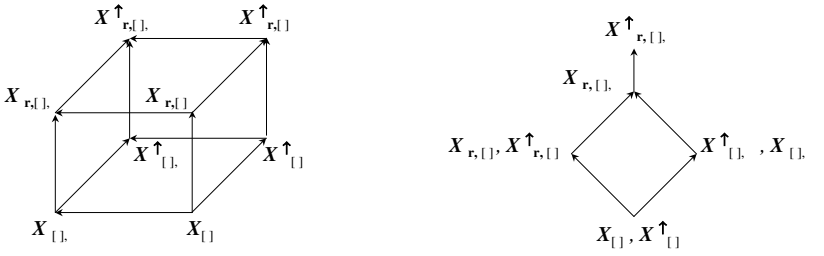

Fig. 3. XPath fragments (left) and their relationship based on their expressive power (right)

expressive. If an arrow connects two fragments F_1 and F_2, then F_1 is strictly less expressive than F_2.

Semantics of Subtree Queries: We present here a framework for *turning XPath queries into subtree queries*. Under the subtree semantics, the result of the evaluation of an XPath expression q on a document D is document $q(D)$ (subdocument of D) obtained as follows:

1. evaluate q using the usual XPath semantics;
2. for each node n obtained from step 1, get its *descendant* and *ancestor* nodes up to the root node of D;
3. finally, construct $q(D)$ by removing all nodes of D that are not in the set of nodes obtained from the previous step.

Note that the resulting document $q(D)$ is a subdocument of D whose root-to-leaf paths are a subset of the set of root-to-leaf paths of D and which inherits the tree structure of D.

Subtree Query Notation: For two subdocuments D_1 and D_2 of D, $D_1 \mid_D D_2$ is the subdocument whose nodes (edges) are the union of the nodes (edges) in D_1 and D_2.

From this point on, when we write an XPath expression in this paper, we will by default consider it as returning a document, using the subtree semantics. The reader can thus think of subtree queries as particularly simple XML queries, with a concise syntax. When we want to make clear whether an XPath expression E is to be considered under the standard XPath semantics we write $\langle E \rangle$, and under the subtree semantics we write $\lfloor E \rfloor$. Similarly, we write $\langle E \rangle(D)$ to denote the result of evaluating E under the XPath semantics and $\lfloor E \rfloor(D)$ to denote the result of evaluating E on document D under the subtree semantics.

3 Composing Subtree Queries

For any XPath fragment F used to define the subtree queries, the *subtree composition problem* for that fragment can be defined as follows: given two expressions

E_1 and E_2 in F, find a single subtree query E' such that for any document D it holds that:

$$\lfloor E' \rfloor(D) = \lfloor E_1 \rfloor(\lfloor E_2 \rfloor(D))$$

From this point on, we refer to expression E_1 as the outer query and E_2 as the inner query (the former is evaluated on the result of the evaluation of the latter), and use $E_1 \circ E_2$ to denote the function satisfying the definition above.

Thus there is a variant of this problem for each XPath fragment; indeed, there are many XPath fragments where no such E' exists for such queries. Closure under composition fails for tree patterns (see below); it also fails for XPath with ancestor and descendant axes, but without qualifiers. Our goal is to give efficient algorithms that solve the composition problems for important fragments of XPath. We begin with a composition algorithm for tree patterns in Section 3.1, extend this to unions of tree patterns in Section 3.2, and to $\mathcal{X}_{r,[\,],\neg}$ in Section 3.3. In Section 3.4 we show how these techniques extend to fragments in our diagram with upward axes. A related problem is the composition of an XPath query with a subtree query in which the outer query E_1 and the desired E' are both interpreted under XPath semantics. The algorithms presented here discuss the composition of subtree queries. We leave the simple extension to the composition of XPath queries with subtree queries for the full paper.

3.1 Subtree Composition for Tree Patterns

As a step towards a composition algorithm for subtree queries based on XPath without negation, we begin with a composition algorithm for *tree patterns*. The algorithm is based on finding homomorphisms of the outer query into the inner query (as done in query containment algorithms for conjunctive queries).

Tree Patterns: A tree pattern is a tree with labels on nodes and edges. Nodes are labeled by names of Σ. Edges are either *child edges* labeled with $*$, or *descendant edges*, labeled with //. There is also a set of *selected nodes* which denote the nodes returned by the pattern. A special case are the *unary tree patterns* (see [4, 3]) that have a single selected node. A general tree pattern Q is evaluated as follows: for each match of one of the patterns in Q, we return any of the nodes labeled as the selected node. Tree patterns return nodesets, and can be translated into XPath queries in $\mathcal{X}_{r,[\,]}$. Thus we can consider them under the subtree semantics as well. For two patterns P_1 and P_2, we write $\lfloor P_1 \rfloor = \lfloor P_2 \rfloor$ to mean these are equivalent under the subtree semantics.

We now note that *tree patterns are not closed under composition*: if one considers the tree patterns E_1=//`listitem` and E_2=//`parlist`, then it is easy to see that there is no tree pattern E' for which $\lfloor E' \rfloor(D) = \lfloor E_1 \rfloor(\lfloor E_2 \rfloor(D))$. Closure under composition can be obtained by adding top-level union. A *union of tree patterns* is a set of tree patterns, and similarly a union of unary tree patterns is a set of unary tree patterns. Such a query returns the union of the result of its component patterns. In [4] it is shown that unions of tree patterns with

descendant edges have exactly the same expressiveness as the XPath fragment $\mathcal{X}_{r,[\,]}$ consisting of expressions built up with descendant and child axes, while the XPath fragment $\mathcal{X}_{[\,]}$ is equivalent to tree patterns with only child edges. Our next goal will be to give a composition algorithm for unions of tree patterns (and hence for $\mathcal{X}_{r,[\,]}$ and $\mathcal{X}_{[\,]}$).

We say that a unary tree pattern is a *path pattern* if all its nodes are linear-ordered by the edge relation of the pattern and where the selected node is not necessarily a leaf node. Path patterns P have the property that for all unary tree patterns Q_1, Q_2, and for any document D:

$$\lfloor P \rfloor(\lfloor Q_1 \mid Q_2 \rfloor(D)) = \lfloor P \rfloor(\lfloor Q_1 \rfloor(D)) \;|_D\; \lfloor P \rfloor(\lfloor Q_2 \rfloor(D))$$

The equality follows because, according to subtree semantics, for some instance D, if the witness of the selected node of P is in $Q_i(D)$, then so are the witnesses of its ancestors and descendants in P. We will see that this gives a particularly simple algorithm for the composition of a path pattern P with a unary tree pattern Q.

Embeddings: Given a tree pattern Q, an *expansion* of Q is an extension Q' of the pattern with additional nodes, which must be either a) adjacent wildcard nodes as descendants of selected nodes or b) additional wildcard nodes inserted between two nodes, ancestors of selected nodes, that were connected in Q with a descendant edge. We refer to the nodes in $Q' - Q$ as *expansion nodes*.

A *pattern embedding* from a path pattern P to a unary tree pattern Q is a mapping m from all the nodes in P to some expansion Q' of Q, where 1) the range of m includes all the expansion nodes of Q' 2) m preserves child edge relationships from P to Q', maps nodes related by descendant edges in P to transitively related nodes in Q', and maps the root of P to the root of Q' 3) the label of node n in P is consistent with that of $m(n)$ (i.e. either they match, or the label of either n or $m(n)$ is the wildcard) 4) all nodes in the range must be comparable to a selected node (i.e. be either ancestors or descendants of the selected node).

Given m, P and Q, let $m(P,Q)$ be the tree pattern obtained from Q' by adding, for every expansion node k in Q' that is a wildcard such that $m(n) = k$, the label of n. The selected node of $m(P,Q)$ is (a) the selected node in Q, if $m(n)$ is an ancestor of the selected node in Q', or (b) $m(n)$, if $m(n)$ is a child wildcard expansion node in Q'. In the presence of a DTD, we can optimize using *DTD-based pruning*: we can check whether any parent/child or ancestor/descendant edge in $m(P,Q)$ contradicts the dependency graph of the DTD (which can be pre-computed), and discard $m(P,Q)$ from the result if there is such a clash.

Example 1. Consider the unary tree pattern $Q_4 =$ `/site/regions//item[quantity]` and the path pattern $Q_1 =$ `/site/regions/europe` given previously (they are both shown as tree patterns in Fig. 4). A double line signifies a descendant edge, a single line a child edge and the selected nodes are starred. To calculate the embedding from Q_1 to Q_4, we expand Q_4 to Q_4' (given in Fig. 4). The dotted lines between the

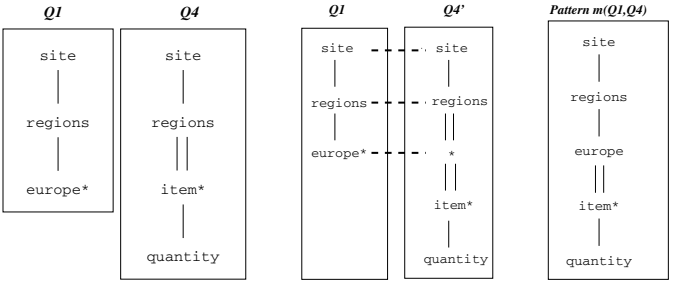

Fig. 4. Path pattern Q_1 and unary tree pattern Q_4 (left), the expansion Q'_4 of Q_4 and the embedding of Q_1 into Q_4 (middle) and the pattern $m(Q_1, Q_4)$ (right)

nodes of Q_1 and Q'_4 show the embedding from Q_1 into Q_4. The pattern $m(Q_1, Q_4)$ obtained from the embedding of the path pattern Q_1 into the unary tree pattern Q_4 is illustrated in Fig. 4.

Lemma 1. *Let P be a path pattern and $Q = \bigcup Q_i$ be a union of unary tree patterns Q_i. For any document D and node n in $\lfloor P \rfloor(\lfloor Q \rfloor(D))$, there is some pattern embedding m from P to Q and node n' returned by $m(P, Q)$, such that n is either an ancestor or descendant of n' in D. In particular $P \circ Q$ is the union of $m(P, Q)$ over all pattern embeddings m.*

In the case where neither P nor Q contain descendant axes there is *at most one* embedding m of P to each Q_i which can be found in time linear in $|P| + |Q_i|$ by proceeding top-down in parallel in P and Q_i, where in Q_i we proceed down the path towards the selected node. We can be more efficient by noting that to find embeddings of P into Q_i, we care only about the query nodes in the path to the selected node of Q_i. Hence we can abstract away the subtrees outside of this path (i.e. the filters associated with the nodes), and leave them as additional annotations on the nodes of Q_i, to be used only after the embedding is found.

The algorithm runs in time bounded by the size of the number of embeddings of the outer query into the inner query. This number is exponential in the worst case (e.g. bounded by $min(2|outer|^{2|inner|}, (2|inner|)^{|outer|})$, since the embeddings are order-preserving). The following result shows that a blow-up is unavoidable:

Proposition 1. *For queries I and O, let $F(I, O)$ denote the minimum size of a tree pattern query expressing the composition $O \circ I$, and let $f(n)$ denote the maximum of $F(I, O)$ over I and O of size at most n. Then $f(n) \in \Omega(2^{\lfloor n/2 \rfloor})$.*

The proof is by considering the composition of outer query $E_1 = //A_1//\ldots//A_n$ and inner query $E_2 = //B_1//\ldots//B_n$ where the A_i and B_j are tags. It is easy to see that a tree pattern contained in $E_1 \circ E_2$ can represent only one interleaving of the B's and A's, hence a composition must include a distinct pattern for each interleaving, and there are more than 2^n of these. Note that if n' is the size of the queries, then $n = n'/2$.

3.2 Subtree Composition for Positive XPath

For P a general unary tree pattern, and $Q = \bigcup Q_i$, one needs to deal with the fact that distinct paths in P may be witnessed in distinct Q_i. Hence we find it convenient to first calculate the possible *fusings* of the inner query and then look at embeddings into these fused patterns. The fusings of the inner queries are represented as tree patterns with multiple selected nodes (recall that these are interpreted semantically in the subtree languages by taking all root-to-leaf paths through any of the selected nodes).

Example 2. Consider the union of tree patterns $q = q_1 \mid q_2$ (inner query) below and the unary tree pattern $p =$`/site/regions/europe/item[location]/descr/text`.

$$q_1 = \texttt{/site/regions/europe/item[quantity]/location}$$
$$q_2 = \texttt{/site/regions/europe/item/descr}$$

One can observe that distinct paths in p are witnessed in distinct q_i's: location sub-elements of item elements are returned by q_1 and descr sub-elements are returned by q_2. The queries obtained by fusing q_1 and q_2 are:

$q_a =$`/site/(regions/europe/item[quantity]/location|regions/europe/item/descr)`
$q_b =$ `/site/regions/(europe/item[quantity]/location | europe/item/descr)`
$q_c =$ `/site/regions/europe/(item[quantity]/location | item/descr)`
$q_d =$ `/site/regions/europe/item[quantity]/(location | descr)`

In general, a fusing of tree patterns $P_1 \ldots P_n$ is determined by an equivalence relation E on the underlying nodes in expansions $P_1' \ldots P_n'$ of the patterns, such that equivalent nodes have consistent labels, every equivalence class contains a node in some P_i, and such that: if for two nodes n and n' it holds that nEn', then the parent of n is E-equivalent to the parent of n'. The corresponding fusing is obtained by identifying equivalent nodes and arranging node and edge labels via the strictest node and edge relation represented in the equivalence class. The selected nodes are the equivalence classes of selected nodes in any P_i'. It can be shown that the conditions above guarantee that the resulting quotient structure is a tree pattern. The fusings between two patterns can be enumerated using a top-down algorithm that finds all matches for the siblings of the root of one tree with descendants of the root in another tree, and then recursively searches for matches of the subtrees of these siblings. This can be extended to multiple patterns by using the equation $Fusings(F_1 \mid F) = \bigcup_{P \in Fusings(F)} Fusings(F_1, P)$. We can trim the number of fusings by restricting to those that correspond to some embedding of root-to-leaf paths from the outer query into nodes of the inner query. Each such embedding generates an equivalence relation on the inner query, by considering which nodes are mapped to by the same element of the outer query.

The last restriction shows that when the outer query is a single tree pattern, the size of the output of the composition algorithm can be bounded by the number of functions taking root-to-leaf paths in the outer query to nodes in an

expansion of the inner query. This in turn is bounded by $(2|inner|)^{|outer|}$. In the case that the outer query is a union of tree patterns with multiple components, note that $\lfloor (O_1|O_2) \circ I \rfloor(D) = \lfloor (O_1 \circ I) \rfloor(D) \mid_D \lfloor (O_2 \circ I) \rfloor(D)$, so we can treat each component of the outer query separately. For a union of tree patterns Q, let $br(Q)$ be the number of trees in Q and $l(Q)$ be the maximum size of any tree in Q. The argument above shows that a composition can be produced in time at most $br(O)(2|I|)^{l(O)}$. Again a worst-case exponential lower bound in output size holds (using the same example as in Proposition 1).

From the algorithm above, we can get composition closure for $\mathcal{X}_{[\,]}$ and $\mathcal{X}_{r,[\,]}$, since each XPath query in these fragments can be translated into a union of tree patterns. However, the translation to tree patterns itself requires an exponential blow-up, since our $\mathcal{X}_{[\,]}$ and $\mathcal{X}_{r,[\,]}$ allow the '|' (union) operator to be nested arbitrarily (in contrast to XPath 1.0, where union is permitted only at top-level). One can give modifications of these embedding algorithms that work directly on $\mathcal{X}_{[\,]}$ and $\mathcal{X}_{r,[\,]}$, and give only a single-exponential blow-up; we omit these variations for space reasons and explain only the corresponding bounds. We redefine br and l, and extend them to filters, using $br(E_1 \mid E_2) = br(E_1) + br(E_2)$, $br(E_1/E_2) = max\{E_1, E_2\}$, $l(E_1 \mid E_2) = max\{l(E_1), l(E_2)\}$,$l(E_1/E_2) = l(E_1) + l(E_2)$. br and l are both preserved in filter steps and are equal to 1 for step expressions. For filters, $\wedge$ and $\vee$ are handled analogously to '|' and '/' above. Then the number of embeddings from $\mathcal{X}_{r,[\,]}$ expression O into $\mathcal{X}_{r,[\,]}$ expression I in this more general sense is again bounded by $br(O)(2|I|)^{l(O)}$, and so this gives a bound on the running time of an embedding-based algorithm for $\mathcal{X}_{[\,]}$ and $\mathcal{X}_{r,[\,]}$ as well.

3.3 Extending to Fragments with Negation

When we turn towards fragments with negation, we again have to deal with the fact that the outer query can interact with different components of the inner query. Consider for example the outer query Q_3 and the inner query Q_2:

$$Q_3 = \texttt{//item[location | descr]} \quad Q_2 = \texttt{//item[not(quantity)] | //item/location}$$

both first presented in Section 1. One can observe that information about item elements requested by the outer query Q_3 can be found in both subqueries of the inner query Q_2. However, in this case we can make use of negation to rewrite the inner query in such a way that different components of a union are "disjoint", and hence that all parts of a match lie in the same component. Once this normalization is done, we will have a straightforward inductive approach, with more succinct output. The fact that the algorithm for fragments with negation is simpler is not surprising, since generally we expect that the addition of more features to the language will make composition easier, while potentially making the composed queries harder to optimize.

Normalization Step: We define first the notion of *strongly disjoint queries* that we will use in the following in order to perform the normalization step.

Two XPath queries q and q' are said to be *strongly disjoint* if for any document D, for any nodes n returned by q and n' returned by q' on D, the root-to-leaf paths to n and n' meet only at the root. An XPath expression E is said to be in *separable normal form* (SNF) if for every subexpression of E of the form $E_1 \mid \ldots \mid E_n$, the E_i are *pairwise strongly disjoint*. Normalization is done by taking a subexpression $E_1 \mid E_2 \mid \ldots \mid E_n$ and breaking up any leading filters into boolean combinations to ensure that the filters are either disjoint or identical, grouping together expressions with the same filter, and recursing on the remaining subexpression. The significance of strong disjointness is the following simple lemma:

Lemma 2. *Suppose E_1 and E_2 are in separable normal form and P, E_1, E_2 are in the fragment $\mathcal{X}_{r,[\,],\neg}$. Then, for all documents D it holds that:*

$$\lfloor P \rfloor(\lfloor E_1 \mid E_2 \rfloor(D)) = \lfloor P \rfloor(\lfloor E_1 \rfloor(D)) \mid_D \lfloor P \rfloor(\lfloor E_2 \rfloor(D))$$

Using this we get a simple algorithm for composing any subtree query in the fragment $\mathcal{X}_{r,[\,],\neg}$ with any E in separable normal form. The algorithm is shown in Fig. 5, making use of an auxiliary function used for filters.

Example 3. Consider the outer query Q_3 and the inner query Q_2 given previously. When the inner query is translated into separable normal form, we obtain:

$$Q_2' = \texttt{//item}(\texttt{[not(quantity)]}/(\epsilon \mid \texttt{location}) \mid \texttt{[quantity]/location})$$

The query resulting from the application of the inductive algorithm, the application of the simplification rules from [4] ($E/\emptyset = \emptyset$, $E \mid \emptyset = E$ and $E \mid E = E$) and the final composed query are shown in Fig. 6.

The example above shows that our algorithms rely on post-processing using some basic simplification rules. Currently we use the rules from [4].

An important feature of the algorithm is that it runs in polynomial time in the size of the queries, under the assumption of normalization (for un-normalized queries, there is a provable exponential blow-up as before). Furthermore, it does not require normalization of the outer query, only the inner one. Another key advantage of this algorithm is that it can be easily extended to deal with data value equality and inequality: we have to add an extra case to handle filters $E_1 = E_2$ or $E_1 \neq E_2$ to the auxiliary function given in Fig. 5, and these are handled by a trivial recursive call. Indeed, one can show that there are subtree queries using data value equality but no negation, whose composition *requires* negation to be expressed.

3.4 Composition for Fragments with Upward Axes

We now discuss how these algorithms extend to the remaining fragments in Fig. 3. For $P, Q \in \mathcal{X}^{\uparrow}_{[\,]}$, one approach is to first apply the algorithms of [4, 13] to remove upward qualifiers, and then proceed as for $\mathcal{X}_{[\,]}$. Similarly, by removing upward axes from $\mathcal{X}^{\uparrow}_{r,[\,]}$ and applying the embedding algorithm for $\mathcal{X}_{r,[\,]}$, we get

function $\circ_{ind}(O{:}\mathcal{X}_{r,[\,],\neg},\ \ I{:}\ \mathcal{X}_{r,[\,],\neg}$ in SNF)
returns $\mathcal{X}_{r,[\,],\neg}$ expression
[1] **switch** O
[2] **case** $O_1|O_2$ **return**$(\circ_{ind}(O_1, I) \mid \circ_{ind}(O_2, I))$
[3] **case** $[F]/O_2$ **return**$([\circ_f(F, I)]/\circ_{ind}(O_2, I))$
[4] **case** A/O_2
[5] **switch** I
[6] **case** $I_1|I_2$ **return** $(\circ_{ind}(O, I_1) \mid \circ_{ind}(O, I_2))$
[7] **case** $[F]/I_2$ **return** $([F]/\circ_{ind}(O, I_2))$
[8] **case** $\emptyset$ **return** $(\emptyset)$
[9] **case** ϵ **return** (O)
[10] **case** A/I_2 **return** $(A/\circ_{ind}(O_2, I_2))$
[11] **case** B/I_2, $B \neq A$ **return** $(\emptyset)$
[12] **case** $//I_2$ **return** $((A/\circ_{ind}(O_2, //I_2)) \mid \circ_{ind}(O, I_2))$
[13] **end switch**
[14] **case** $//O_2$
[15] **switch** I
[16] **case** $I_1|I_2$ **return** $(\circ_{ind}(O, I_1) \mid \circ_{ind}(O, I_2))$
[17] **case** $[F]/I_2$ **return** $([F]/\circ_{ind}(O, I_2))$
[18] **case** $\emptyset$ **return** $(\emptyset)$
[19] **case** ϵ **return** (O)
[20] **case** A/I_2 **return** $(A/\circ_{ind}(O, I_2) \mid \circ_{ind}(O_2, I))$
[21] **case** $//I_2$ **return** $(//(\circ_{ind}(I, O_2)|\circ_{ind}(O, I_2)))$
[22] **end switch**
[23] **case** ϵ **return**(I)
[24] **case** $\emptyset$ **return**$(\emptyset)$
[25] **end switch**
end

function $\circ_f(F{:}\ \mathcal{X}_{r,[\,],\neg}$ filter, $I{:}\ \mathcal{X}_{r,[\,],\neg}$ query in SNF)
returns $\mathcal{X}^{\uparrow}_{r,[\,],\neg}$
[1] **switch** F
[2] **case** $\neg F_1$ **return**$(\neg \circ_f (F_1, I)$
[3] **case** F_1 *op* F_2, *op* $\in \{\mathsf{and}, |\}$ **return**$(\circ_f(F_1, I)$ *op* $\circ_f (F_2, I))$
[4] **case** E **return** $(\circ_{ind}(E, I))$
[5] **end switch**
end

Fig. 5. Inductive Algorithm and Inductive Filter Composition

an algorithm for $\mathcal{X}^{\uparrow}_{r,[\,]}$. For $\mathcal{X}^{\uparrow}_{[\,],\neg}$, one can also remove upward axes (although this is not stated explicitly in [13, 4]). Indeed, a simple inductive algorithm can compute from a $\mathcal{X}^{\uparrow}_{[\,],\neg}$ query Q a query Q' that is equivalent to Q in every context (not just the root), such that Q' is a union of queries either of the form $[F_0]/../[F_1]/.../../[F_n]/E_n$, where F_i are filters in $\mathcal{X}_{[\,],\neg}$ and E_i is an expression in $\mathcal{X}_{[\,],\neg}$, or of the form $[F]/E$. When one restricts to applying Q' at the root, the components of the union of the first form can be eliminated.

We refer to these as "eliminate-first algorithms', since they eliminate upward axes before composing, producing a composed query with no upward axes. Although the upward-axis removal algorithms of [4, 13] themselves require an exponential blow-up, they produce from a query Q a new query Q' without upward axes such that $l(Q') = l(Q)$ and $br(Q') < 2^{br(Q)}$. Combining these bounds with the upper bounds on the number of embeddings, we see that the output of eliminate-first composition algorithms has size bounded by $2^{br(O)}(2 \cdot 2^{|I|})^{l(O)} = 2^{br(O)}2^{(|I|+1)l(O)}$, single-exponential in both inputs.

Result of the inductive algorithm:

$Q_3 \circ Q_2 =$ //item [([not(quantity)]/(descr | ∅) | [quantity]/∅) |
([not(quantity)]/(location | location) | [quantity]/location)]

Application of the simplification rules:

$Q_3 \circ Q_2$=//item[[not(quantity)]/descr|([not(quantity)]/location |[quantity]/location)]

Composed Query:

$Q_3 \circ Q_2 =$ //item [[not(quantity)]/descr | [not(quantity)]/location] |
//item[quantity]/location

Fig. 6. Queries resulting from the application of the inductive algorithm and the simplification rules

An alternative is to compose queries first, and then (if possible, and if desired) remove upward axes from the result. Somewhat surprisingly, if one simply wants to perform composition for fragments with upward axes, one can do so in polynomial time. We show this for the largest XPath fragment, $\mathcal{X}^{\uparrow}_{r,[\,],\neg}$, leaving the details for other fragments for the full paper. Note that $\mathcal{X}^{\uparrow}_{r,[\,],\neg}$ does not eliminate upward axes (consider, for example, the query asking for all location nodes that have only item nodes as ancestors), so only an eliminate-first algorithm is not applicable here. The polynomial time method for composing subtree queries uses a trick from [12]. One can simply take the outer query and "pre-test" each node to see if satisfies the inner query. Officially, for each query Q, let Q^r be the query obtained from changing occurrences of $/R$ to the equivalent $*/[\text{label} = R]$, and then reversing each of the axes in Q that do not occur within a filter. Let IN_Q be the query Q^r[not(..)]. Clearly, the filter $[IN_Q]$ returns true exactly when a node is in the result of the XPath expression Q. If we take Q to be the inner query, and we add the filter $[(//|..^*)[IN_Q]]$ to each step of the outer query, we are checking whether nodes witnessed in the outer query are actually in the result of the inner query. Hence the result is the composition. In the case where the outer query has only downward axes, we need only add these filters to the leaves of the syntax tree. Similar algorithms are available for fragments without negation; for example, in the case of $\mathcal{X}^{\uparrow}_{[\,]}$, a root test not(..) is unnecessary, since one can test statically whether a path leads back to the root by just "counting steps". For $\mathcal{X}^{\uparrow}_{[\,]}$ and $\mathcal{X}^{\uparrow}_{r,[\,]}$ one can combine this PTIME algorithm with upward-axis removal, obtaining a composed query with no upward axes in exponential time.

These last composition algorithms are only useful when significant optimization is in place in the query evaluation engine; the resulting composed query clearly includes an enormous amount of upward and downward navigation, even when the initial queries lack upward axes completely. We are still developing optimization rules for fragments with upward axes and negation; However, to the extent that special-purpose optimizers are developed for $\mathcal{X}^{\uparrow}_{r,[\ ],\neg}$, composing while staying within this fragment can be helpful. This example also serves to emphasize that closure under composition becomes easier as the fragment becomes larger. Taking the algorithms altogether, what we have shown is the following:

Theorem 1 (Composition Closure). *Let F be any of the XPath fragments in Fig. 3, or either of the XPath fragment $\mathcal{X}^{\uparrow}_{r,[\ ],\neg}$ $\mathcal{X}_{r,[\ ],\neg}$ extended with data value equality. Then for any Q and Q' in F there is $Q'' \in F$ such that for every document D it holds that:*

$$\lfloor Q \rfloor(\lfloor Q' \rfloor(D)) = \lfloor Q'' \rfloor(D)$$

In addition for every such $Q, Q' \in F$ we can get $Q'' \in F$ such that

$$\langle Q \rangle(\lfloor Q' \rfloor(D)) = \langle Q'' \rangle(D)$$

4 Conclusion

In this paper we discussed the specification and composition of subtree queries for common fragments of XPath. We provided composition algorithms for each of the resulting subtree languages and showed that these languages are closed under composition. Despite the complexity of the composition algorithms, experiments in [2] show that we can have important benefits over XQuery trivial composition (when the composed subtree query is translated into an XQuery expression and evaluated with an off-the-shelf XQuery evaluator). The composition algorithms presented in this paper are used in the Incognito access control system being developed at Bell Laboratories. Incognito uses the Vortex rules engine [11] to resolve user context information and applies the composition algorithms to compose user queries with access control views (all of these being subtree queries) to compute the authorized user query that will be evaluated against XML documents.

References

1. S. Abiteboul, A. Bonifati, G. Cobena, I. Manolescu, and T. Milo. Dynamic XML Documents with Distribution and Replication. In *SIGMOD*, 2003.
2. B. Alexe, M. Benedikt, and I. Fundulaki. Specification, Composition and Evaluation of XML Subtree Queries. Technical report, Bell Laboratories, 2005. Available at http://db.bell-labs.com/user/fundulaki/.
3. S. Amer-Yahia, S. Cho, L. V. Lakshamanan, and D. Srivastava. Minimization of Tree Pattern Queries. In *SIGMOD*, 2001.

4. M. Benedikt, W. Fan, and G. Kuper. Structural Properties of XPath Fragments. *Theoretical Computer Science*, 2003.
5. E. Bertino and E. Ferrari. Secure and Selective Dissemination of XML Documents. *TISSEC*, 5(3):290–331, 2002.
6. J. Clark et. al. (eds.). XML Path Language (XPath) 1.0. W3C Recommendation, 1999. http://www.w3c.org/TR/xpath.
7. M. Fernandez, Y. Kadiyska, D. Suciu, A. Morishima, and W.-C. Tan. SilkRoute: A framework for publishing relational data in XML . *TODS*, 27(4):438–493, 2002.
8. I. Fundulaki, G. Giraud, D. Lieuwen, N. Onose, N. Pombourq, and A. Sahuguet. Share your data, keep your secrets. In *SIGMOD*, 2004. (Demonstration Track).
9. G. Gottlob and C. Koch. Monadic Datalog and the Expressive Power of Languages for Web Information Extraction. In *PODS*, 2002.
10. G. Gottlob, C. Koch, and R. Pichler. Efficient Algorithms for Processing XPath Queries. In *VLDB*, 2002.
11. R. Hull, B. Kumar, and D. Lieuwen. Towards Federated Policy Management. In *Int'l Workshop on Policies for Distributed Systems and Networks*, 2003.
12. M. Marx. XPath with conditional axis relations. In *EDBT*, 2004.
13. D. Olteanu, H. Meuss, T. Furche, and F. Bry. XPath: Looking forward. XMDM, 2002.
14. A. Sahuguet and B. Alexe. Sub-document queries over XML with XSquirrel. In *WWW*, 2005.
15. A. Sahuguet, B. Alexe, I. Fundulaki, P. Lalilgand, A. Shikfa, and A. Arnail. User Profile Management in Converged Networks. In *CIDR*, 2005.
16. A. Schmidt, F. Waas, M. Kersten, M. Carey, I. Manolescu, and R. Busse. XMark: a benchmark for XML Data Management. In *VLDB*, 2002.
17. I. Tatarinov, Z. Ives, A.Y. Halevy, and D.S. Weld. Updating XML. In *SIGMOD*, 2001.
18. P. Wadler. Two Semantics for XPath. Technical report, Bell Laboratories, 2000. Technical Memorandum.