Querying XML with Update Syntax

Philip Bohannon
Lucent Technologies-Bell Labs
bohannon@research.bell-labs.com

Gao Cong
Univeristy of Edinburgh
gao.cong@ed.ac.uk

Wenfei Fan
University of Edinburgh & Bell Laboratories
wenfei@inf.ed.ac.uk

Abstract

This paper investigates a class of *transform queries* proposed by XQuery Update [6]. A transform query is defined in terms of XML update syntax. When posed on an XML tree T, it returns another XML tree that would be produced by executing its embedded update on T, without destructive impact on T. Transform queries support a variety of applications incluing XML hypothetical queries, the simulation of updates on virtual views, and the enforcement of XML access control. In light of the wide-range of applications for transform queries, we develop automaton-based techniques for efficiently evaluating transform queries and for computing their compositions with user queries in standard XQuery. We provide (a) three algorithms to implement transform queries without change to existing XQuery processors, (b) a linear-time algorithm, based on a seamless integration of automaton execution and SAX parsing, to evaluate transform queries on large XML documents that are difficult to handle by existing XQuery engines, and (c) an algorithm to rewrite the composition of user queries and transform queries into a single efficient query in standard XQuery. We also present experimental results comparing the efficiency of our evaluation and composition algorithms for transform queries.

1. Introduction

A recent W3C XQuery Update Working Draft [6] proposes the notion of a *transform query* for XML. The idea is to use update syntax to define a query *without* affecting the underlying data store. As such, transform queries are in fact an XQuery syntax for *hypothetical query* [1, 4, 12, 16]. Traditionally, a hypothetical query has the form "Q when $\{\{U\}\}$", and is to find the value that query Q would return on a database that would be obtained by executing update U on the original database DB, without actually updating DB. Prior work has noted the utility of hypothetical queries in decision support, version management, active databases and integrity maintenance, among other things.

While SQL engines offer some support for hypothetical queries (for example, consider the **within group** syntax for finding the rank a hypothetical row that would have in an ordered list [7]), the syntax proposed in [6] would provide a general hypothetical-query facility for XML. We now identify some general applications of transform queries and present specific examples based on these applications. These examples also illustrate the difficulty of accomplishing certain simple tasks in XQuery without the **transform** syntax.

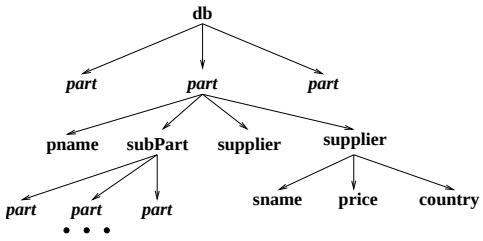


Figure 1: An example XML document

Example 1.1: Consider an XML document T_0 depicted in Fig. 1. The document contains a list of *part*s. Each *part* has a *pname* (part name), a list of *supplier*s and a *subpart hierarchy*; a *supplier* in turn has a *sname* (supplier name), a *price* (offered by the supplier), and a *country* (where the supplier is based).

• *Updates as queries.* One wants to write a query that finds all the information in T_0 *except price*; in other words, the query is to return an XML tree that is almost identical to T_0 *excluding* the *price* elements. Such a query cannot be easily expressed in standard XQuery [3] without complicated user-defined recursive functions, and similar queries are difficult to write in general to produce a document substantially similar to an existing document. In contrast, this can be readily expressed as a simple *transform query*:

transform copy $a := doc("foo") **modify do delete** *$a//price* **return** $a

where doc("foo") refers to T_0. Conceptually, the query first makes a copy T_1 of T_0, and then performs the update "**delete** $a//price" on T_1. The updated T_1 is returned as the result of the query. Although defined with update syntax, this query is *non-updating* [6]: it *has no destructive impact* of updates on the underlying data T_0.

• *Security views.* In an organization, a number of user groups with access to T_0 may be subject to different access-control policies. Such a policy might, for example, prevent disclosure of *price* information from suppliers of certain countries for which the group of users was not responsible. To enforce the access control policy, each group can be provided with a security view [10] that returns a document containing all the data from T_0 *excluding* the sensitive *price* information. Since each user group has a slightly different view, it is not in general reasonable to materialize and maintain each of the provided security views; thus the views should be kept *virtual*. Unfortunately, as mentioned above, such views are far from trivial to write by hand in XQuery. In this example, a recursive function is required since the *price* information may appear at arbitrary depths in T_0. In contrast, it is conceptually straightforward to define the view as a transform query:

transform copy $a := doc("foo") **modify**
do delete *//supplier[country='c_1' $\vee \ldots \vee$ country='c_n']/price* **return** $a

The view prevents the disclosure of price information of those suppliers based in countries $c_1, \ldots, c_n$. Note that the intention is *not* to delete this data *in the source*; instead, it is merely to define the security view of a client with the update syntax.

• *Updating virtual views.* Consider a virtual view defined as an XQuery view Q_V on top of T_0. A user may want to pose an update

on the virtual view. In this case, there may be *no sensible* notion of performing an update on the virtual data. Leveraging transform queries one could still query a new document that would result from such an update on the view. This can be done by (a) writing a transform query Q_t in terms of the desired update; (b) upon receiving a user query Q, composing Q with the transform query Q_t, denoted by Q_c; (c) composing Q_c with the view definition Q_v, denoted by Q'_c. Then, the user query Q can be answered by evaluating Q'_C directly on the underlying data T_0. Thus one can query *an "updated" virtual view without materialization* of the view. □

However important, to the best of our knowledge, no previous work has studied the evaluation of transform queries or their composition with user queries.

Automaton approach. We approach the evaluation of transform queries as a natural extension of the *automaton* approach to XPath evaluation [9, 17, 19, 8]. The idea is to combine the automaton evaluation of the XPath expression, which defines the update embedded in a transform query, with a recursive transformation of the tree to evaluate the transform query. This provides a general and *portable* approach to efficient transform query evaluation, and we demonstrate two very different implementations, one on top of XQuery and another inside a SAX parser. Further, this approach facilitates analysis of transform queries: for example, we use our automaton representation to define a query composition technique between transform queries and a class of XQuery expressions.

Copy-and-Update Approach. Naturally, an alternative approach to implement a transform query is to copy and update the affected data. While in the worst case, the automaton approach can be linear in the size of the data, just like copying, in the majority of cases the automaton approach can evaluate a query while touching only a small subset of the data, while copying always requires a complete traversal of the entire XML document, and thus always linear time and space. Furthermore, the automaton approach is amenable to query composition, unlike the copy-and-update approach. In addition, as opposed to copy-and-update approach, the automaton approach does not rely on efficient support of XML updates, which is not yet in place for some existing XQuery engines.

Contributions. We provide efficient automaton-based evaluation algorithms, as well as an algorithm for computing compositions, for a class of transform queries in which embedded updates are defined in terms of XPath expressions.

(1) Transform algorithms. We show three algorithms for transform-query evaluation, two of which are based on the automaton approach. Our first algorithm, referred to as Naive, is based on a simple *query rewriting* technique to translate transform queries (and their embedded updates) into standard XQuery. It has quadratic-time data complexity in the worst case. We then present two algorithms based on the automata approach, namely topDown and bottomUp. The topDown approach is appropriate when an efficient engine is available for evaluating qualifiers. Further, topDown can easily incorporate alternative XPath automaton techniques. The bottomUp algorithm incorporates qualifier evaluation, removing the requirement of an external engine and supporting streaming behavior, at the cost of increased complexity.

Our proposed algorithms have several useful features. (a) They can be readily migrated to XQuery engines that do not support updates. (b) They remain side-effect free and thus can benefit from the existing and upcoming optimizations utilizing referential transparency property of XQuery. (c) As opposed to the Naive evaluation strategy and the copy-and-update approach, algorithms topDown and bottomUp traverse only necessary part of an input XML tree T rather than the entire tree.

While an advantage of our approach is that it can be applied with other automata formalisms, we note that the one we present is actually simpler than previous automaton models developed for evaluating XPath expressions (e.g., [9, 17, 19, 8]) and avoids some efficiency issues (see Section 8 for a detailed discussion).

(2) Composing user and transform queries. Capitalizing on our automaton technique, we propose an algorithm for composing a user query Q and a transform query Q_t by computing the composition Q_c, a single query in standard XQuery. We show that the algorithm significantly outperforms the conceptual strategy that sequentially evaluates Q and Q_t one by one. Indeed, Q_c accesses only relevant part of an input XML document, *without copying or traversing the entire document* in many cases. To our knowledge, this is the first automaton-based composition algorithm for XML queries. The algorithm provides us with the ability to efficiently support hypothetical queries as well as query and "update" virtual views.

(3) An algorithm implemented on SAX. As most existing XQuery engines represent XML documents as memory intensive DOM trees, they do not handle large XML documents very well. To cope with this, we propose another algorithm for evaluating transform queries, referred to as twoPassSAX, based on a seamless integration of our automaton-based algorithm with SAX parsing. This approach makes the facility easy to integrate with an XQuery engine or with other applications, since the result of the **transform** may be accessed as a SAX event stream.

(4) Experimental Study. We have implemented our evaluation algorithms both *in XQuery* on top of XQuery engines (Naive, topDown, bottomUp) and within XQuery engines (twoPassSAX), and evaluated them with data from XMark [24]. Our results show the following. (a) Usable transform queries can be generated for a wide variety of XML updates. (b) Our algorithms are efficient and scale well, both *when* implemented in XQuery on top of query processor and as part of the query processor implementation. (c) Our algorithm twoPassSAX can handle very large XML documents while the memory overhead is very small. We have also implemented and evaluated our composition algorithm. Our experimental results demonstrate that our composition technique is effective.

Organization. Section 2 defines transform queries. Section 3 introduces algorithms Naive and topDown, as well as our automaton technique, for evaluating transform queries. As an immediate application of the automaton technique, we provide our algorithm for composing user and transform queries in Section 4. Section 5 presents algorithm bottomUp, our optimization technique for evaluating expensive XPath qualifiers. Section 6 provides algorithm twoPassSAX for evaluating transform queries on large XML documents. Experimental results are presented in Section 7, followed by related work in Section 8, and conclusions in Section 9.

2. Transform Queries

We study a class of transform queries [6] of the form:

transform copy $\$a$:= *doc("*T_0*")* **modify do** $u(\$a)$ **return** $\$a$

where u(\$a) is an *embedded update* expression. Here we study updates supported by most proposals for XML update languages [20, 21, 23, 25, 6, 14], which are of one of the following four forms:

insert	e **into** $\$a/p$	**delete**	$\$a/p$
replace	$\$a/p$ **with** e	**rename**	$\$a/p$ **as** l

where p is an XPath expressions, e is a constant XML element (subtree), and l is a label. Example transform queries can be found in Example 1.1.

To give the semantics of transform queries we first review XPath and XML updates defined in terms of XPath.

XPath. We consider core XPath [15] with downward modality. This class of queries, referred to as $\mathcal{X}$, is defined by:

$$p ::= \epsilon \mid l \mid * \mid p/p \mid p//p \mid p[q],$$
$$q ::= p \mid p = \text{`}s\text{'} \mid label() = l \mid q \wedge q \mid q \vee q \mid \neg q,$$

where ϵ, l, $*$ and '$/$' denote self (i.e.,'.'), a label, a wildcard, and the *child*-axis respectively, and '$//$' stands for */descendant-or-self::node()/*; and q in $p[q]$ is called a *qualifier*, in which s is a string value, and '$\wedge$', '$\vee$' and '$\neg$' denote conjunction, disjunction and negation, respectively. We abbreviate $p_1/\,//p_2$ as $p_1//p_2$.

The class $\mathcal{X}$ subsumes tree patterns commonly used in practice. We consider this practically useful fragment of XPath (and updates) to simplify the discussion and focus on the main idea of transform queries.

On an XML tree T, an XPath query p is evaluated at a *context node* v in T, and its result is the set of nodes of T reachable via p from v. We denote the result of the query by $v[\![p]\!]$.

XML updates. Given an XML tree T with root r (i.e., r takes the place of $\$a$ following $\$a$ = *doc('T_0')*), the update operation u does the following:

• **insert** e **into** r/p. This operation finds all the *elements* reachable from r via p in T, and adds the new element e as the last child of each of those elements. Specifically, (a) it computes $r[\![p]\!]$, and (b) for each element v in $r[\![p]\!]$, it adds e as the rightmost child of v.

• **delete** r/p. It first computes $r[\![p]\!]$ and then removes all the nodes in $r[\![p]\!]$ (along with their subtrees) from T.

• **replace** r/p **with** e. This operation first computes $r[\![p]\!]$ and then replaces each v in $r[\![p]\!]$ with the element e.

• **rename** r/p **as** l. This operation first computes $r[\![p]\!]$ and then for each v in $r[\![p]\!]$, changes the label of v to l.

Semantics of transform queries. Given an XML tree T_0 (i.e., *doc('T_0')*), the tree returned by a transform query Q_t is the one that *would be produced* by the following: (a) create a copy T of T_0, (b) apply update u on T, and (c) return T as the answer of Q_t. We denote the XML tree returned by Q_t as $Q_t(T)$.

This paper provides techniques to execute and *compose* transform queries without resorting to the naive approach suggested by the semantics.

3. Evaluating Transform Queries

In this section we propose two algorithms for evaluating transform queries on top of existing XQuery engines.

The first algorithm, referred to as the Naive Method, is based on a simple query rewriting technique that translates a transform query into an equivalent query in standard XQuery. It shows that transform queries can be readily supported by available XQuery engines. As XML updates can be embedded in transform queries, this provides the capability of supporting XML updates within the immediate reach of *existing* XQuery engines, by first expressing XML updates as transform queries, and then rewriting the transform queries into equivalent queries in standard XQuery.

While the Naive Method is conceptually simple, it may perform poorly. In light of this we present the second algorithm, referred to as the Top Down Method, based on an automaton technique. The Top Down Method provides performance guarantees, and the automaton technique also serves as the basis of the evaluation and composition algorithms to be introduced in later sections.

```
let $xp := doc(T)/p
return document  {for $n in doc(T)/* return local:insert($n, $xp)}

declare function local:insert($n as node(), $xp as node()*) as node()
  { if $n[element()]
    then element
      { fn:local-name($n)}
      { for $c in $n/(*|@*) return local:insert($c, $xp),
        if (some $x in $xp satisfies ($n is $x))   /*test if $n ∈ $xp*/
          then {e} else ()}
    else $n
  }
```

Figure 2: From transform queries to XQuery

3.1 The Naive Method

The Naive Method is based on query rewriting: given a transform query Q_t, it finds a query Q^s in standard XQuery such that $Q^s(T) = Q_t(T)$ for any XML document T.

First consider Q_t = **transform copy** $\$a$:= doc("T") **modify do insert** e **into** $\$a/p$ **return** $\$a$. Suppose that e evaluates to an XML element, and p is an XPath query. The query Q_t can be rewritten into Q^s in standard XQuery, as shown in Fig. 2, following recursive-query transformations suggested by the XQuery standard [3]. Let r be the root of T. In a nutshell, the query Q^s first evaluates the XPath query p to compute $r[\![p]\!]$, the set of nodes selected by p, and then invokes a function *insert( )*. This function takes a node $\$n$ and $r[\![p]\!]$ as input, and it processes $\$n$ as follows. If $\$n$ is an element, then it constructs an element that has the same label as that of $\$n$ and carries the children of $\$n$; furthermore, if $\$n$ is in $r[\![p]\!]$, then it adds e as the last child of $\$n$. The function then recursively processes the children of $\$n$ in the same way. If n is not an element, it is returned without change. Obviously $Q^s(T)$ produces the same result as $Q(T)$. This yields a generic complete-query template for **insert** transform queries. Observe that *no copy* of *doc(T)* needs to be made in order to evaluate Q^s.

Similarly, one can rewrite **delete**, **replace** and **rename** transform queries Q_t into equivalent queries Q^s in standard XQuery.

Note that doc(T)/p and e in this template can be instantiated with *arbitrary* XQ*uery queries*, not just $\mathcal{X}$ or constant expressions. Thus we have in fact shown that transform queries defined in terms of a wide variety of updates can be rewritten into standard XQuery.

However, unless the XQuery engine optimizes the test $n \in \$xp$, the rewritten queries are inefficient when the scope of the update is broad (i.e., when p is not very selective and $|\$xp|$ is large): in the worst case it takes quadratic ($O(|T|^2)$) time in the size of T.

3.2 An Automaton Abstraction

Our efficient transform evaluation algorithms are based on the notion of *selecting* NFA for XPath expressions, which is a mild extension of non-deterministic finite state automata (NFA). The purpose of this automaton is to inform our transform algorithms whether or not the embedded update should be executed at each node n encountered during a traversal of the document, i.e., whether $n \in r[\![p]\!]$. It accomplishes this task by (a) maintaining a set of states S, and (b) updating S as each node is encountered by using a function nextStates (). At any point, the current node is matched by p if any of the states in S are *final states* of the automaton. Details of the construction will be given in Section 3.4.

3.3 The Top Down Method

We now introduce the Top Down Method, denoted by topDown, that avoids the quadratic complexity of the Naive method by matching paths with an automaton and recursively producing the transform in the same step. We illustrate the Top Down Method for an insert transform query. The method is described by Algo-

function topDown (M_p, S, Q_t, n)
Input: NFA M_p, a set S of M_p states, Q_t: **transform copy** $a := doc("T")$ **modify do insert** e **into** $\$a/p$ **return** $\$a$, and node n in T.
Output: $Q_t(T')$, where T' is the subtree rooted at n.

1. S' := nextStates(M_p, S, n);
2. if $S' = \emptyset$ or n is not an element
3. then return n along with its subtree;
4. return element with same label as n and children consisting of
5. {for each v in $n/*$ do
6. topDown (M_p, S', Q_t, v);
7. if there is $(s,[q]) \in S'$ s.t. $(s,[q])$ is the final state, and checkp(q, n)
8. then {e } /* add e as the last child of n */
9. }

Figure 3: Algorithm topDown

rithm topDown given in Fig. 3; the algorithms for **delete**, **rename** and **replace** transform queries are similar.

The (recursive) algorithm takes as input an insert transform query Q_t, the selecting NFA M_p of XPath p embedded in Q_t, a set S of *current* states in M_p, and a node n in the XML tree T. When called with n as the root of an XML tree T and S consisting of (the ϵ-closure of) the start state for M_p, topDown computes $Q_t(T)$. Given the set S that keeps track of the states reached after traversing T from the root to the parent of n, topDown computes the set S' of the states for the current node n by using nextStates(). If S' is empty, then the subtree of n should not be changed, and thus it is simply copied to the result (lines 2–3). Otherwise topDown recursively processes the children of n, taking S' as a parameter (lines 5–6). Furthermore, if S' includes the final state and its corresponding qualifier is satisfied, then the new element e is evaluated and inserted as the last child of n (lines 7–8). The qualifier checking is done by calling a predefined function checkp(), where checkp(q_i, n) returns true iff $\epsilon[q_i]$ is non-empty at n.

Remark. Observe the following about topDown. First, it can be readily realized in a way that incurs no side effects and thus can work directly on any existing XQuery engine. Second, if checkp() takes constant time, then for any transform query Q_t on an XML tree T, the evaluation of Q_t takes at most $O(|T|\ |p|)$ time, where p is the $\mathcal{X}$ query embedded in Q_t. That is, it takes time *linear* in $|T|$. Third, the use of selecting NFA allows us to simply return unchanged subtrees without further processing. In particular, for a **delete** transform query, the selecting NFA is able to *prune* deleted subtrees *without loading them*. Fifth, other automaton-based techniques for evaluating p, such as those of [9, 17, 19, 8], can be used for topDown, as long as they can implement the nextStates interface. Finally, topDown does *not copy* T and does not rely on the support of XML updates by XQuery engines.

3.4 Automaton Construction

We now give details of our automaton construction.

Selecting NFA. Given an $\mathcal{X}$ expression p, we generate the selecting NFA of p, denoted by M_p, to identify nodes in $r[\![p]\!]$. Observe that p can be rewritten to an equivalent form $\beta_1[q_1]/\ldots/\beta_k[q_k]$, where β_i is either label l, wildcard $*$ or descendant //. We define the selecting NFA $M_p = (K, \Gamma, \delta, s, f)$, where

1) the set K of states consists of the *start state* $s = (s_0, [true])$, and for each $i \in [1,k]$, a state $(s_i, [q_i])$ denoting the step β_i with the qualifier $[q_i]$, where the *final state* f is $(s_k, [q_k])$;

2) alphabet Γ consists of all the tags in p and a special wildcard $*$;

3) the transition function δ is defined for each i in $[0, k-1]$:

- $\delta((s_i,[q_i]), \beta_{i+1}) = \{(s_{i+1}, [q_{i+1}])\}$ if β_{i+1} is a label or $*$,
- $\delta((s_i,[q_i]), \epsilon) = \{(s_{i+1}, [q_{i+1}])\}$, and
- $\delta((s_i,[q_i]), *) = \{(s_i, [q_i])\}$ if β_{i+1} is //.

function nextStates (M_p, S, n)
Input: NFA M_p with transition function δ, a set S of M_p states, node n
Output: S' the next states of M_p on reaching a node with label l.

1. l := *fn:local-name(n)*;
2. $S^+ := \bigcup_{(s,[q]) \in S} \delta((s,[q]), *) \cup \delta((s,[q]), l)$;
3. $S' := \{(s,[q]) \in S^+ \mid \text{checkp}(q, n)\}$;
4. **return** ϵ-*closure(S')*;

Figure 4: Computing the set of next states at a node

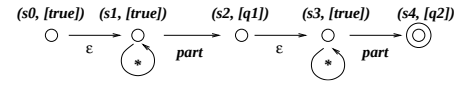


Figure 5: An example selecting NFA of an $\mathcal{X}$ query

Example 3.1: Consider p_1 = *//part[q_1] //part[q_2]* in $\mathcal{X}$, where q_1 is *[pname = 'keyboard']* and q_2 = *[¬ supplier/sname = 'HP' ∧ ¬supplier/price < 15].* Figure 5 gives the selecting NFA of p_1. □

Remark. A selecting NFA M_p has the following notable features. First, M_p has a *semi-linear* structure: the only cycles in M_p are self-cycles labeled $*$ and introduced by //. Note that from any state $(s_i, [q_i])$ at most two states can be reached via the δ function. Second, while M_p is based on the "selecting path" of p, it incorporates its qualifiers into the states, which, as remarked earlier, is effective in *pruning unaffected subtrees*. Third, M_p can be constructed in $O(|p|)$ time, and its size is bounded by $O(|p|)$.

As we have already seen, the set S of (current) states in M_p is maintained during a top-down traversal of the input XML tree T. For each node n in T encountered, n's label is used to change S to S' according to the function nextStates() shown in Fig. 4, which will be described shortly. Based on the set S', Algorithm topDown takes action at the node as follows: (1) if S' includes the final state of M_p, then n is selected by p and the appropriate update action is simulated such that the updated subtree will be returned; (2) if S' is empty, then no change is made to the subtree rooted at n and thus it can be simply copied and returned; and (3) otherwise, n may be on a path to a node selected by p, and the top down traversal proceeds to the children of n. In both cases 1 and 3, topDown proceeds to process the subtree of n in the same way.

Example 3.2: Consider a transform query Q_t with embedded update **insert** c **into** p_1, where c is a *supplier* element with name HP and p_1 is given in Example 3.1. Given the root of the XML tree T_0 of Fig. 1, the NFA of Fig. 5, the query Q_t, and a set S consisting of the start state $(s_0, [true])$ of M_p and $(s_1, [true])$, topDown returns an XML tree that is the same as T_0 except that supplier HP is added to every *part* whose states contain the final state s_4. □

Next States. The function nextStates(), shown in Fig. 4, handles state transitions in M_p when encountering a node n. For each state $(s,[q])$ in S, nextStates() computes the M_p states $(s', [q'])$ reached from $(s,[q])$ by inspecting the label of n and the transition function δ of M_p (line 2); moreover, it checks whether the qualifier $[q']$ is satisfied at n by calling a predefined function checkp().

Note that, to cope with the ϵ transitions in the NFA M_p, we need to compute the ϵ-*closure* of S' (line 4), which is the set of all the states reachable from any state of S' via one or more ϵ transitions in M_p. It is easy to compute the ϵ-*closure* of S' in $O(|p|)$ time. Also, by the construction of selecting NFAs given earlier, if $\delta((s,[q]), *)$ (or $\delta((s,[q]),$ *fn:local-name(n)*)) is defined, then it maps to a *single* state rather than a set. The cardinality of S' is bounded by $O(|p|)$ when computed by repeated calls to nextStates().

4. Composing User and Transform Queries

Based on the automaton technique given in the last section, we next develop an algorithm for computing an efficient composition of a user query and a transform query.

Given a transform query Q_t followed by a user query Q, we want to compute a query Q_c in standard XQuery such that $Q(Q_t(T)) = Q_c(T)$ for any XML document T. As remarked in Section 1, this is important for, among other things, processing hypothetical queries as well as querying and updating virtual views.

Since XQuery allows query composition, a straightforward rewriting Q_c can be given by:

`let` $\$d$ `:=` $Q_t(T)$ `let` d' `:=` $Q(d)$ `return` $\$d'$

This gives us the desired query Q_c in XQuery. We refer to this method as the Naive Composition Method.

Example 4.1: Consider an access control policy that denies a user group the access to suppliers from country 'A'. As shown by Example 1.1, this can be enforced by defining a (virtual) security view in terms of a transform query Q_t. Now suppose the user poses a query Q in XQuery on the security view, which is to find suppliers for *keyboard*. The composition Q'_c of the two can be written as follows, which shows both the transform query Q_t that defines the virtual security view, and the user query Q.

```
<result> {
  let $n := transform copy $a := doc("foo") modify do      /* Q_t */
                delete $a//supplier[country = 'A'] return $a
  for $x in $n/part[pname = 'keyboard']/supplier            /* Q */
  return $x
} </result>
```

This is an example of the Naive Composition Method. □

The Naive method, however, may not be efficient since, given such Q_c, an XQuery engine will evaluate Q_t and Q sequentially, one after the other. It is more costly if the conceptual copy-update strategy is used to evaluate Q_t.

We now describe a more efficient technique, referred to as the Compose Method, that makes use of the automaton representation of the transform query to compose a user query with it. This technique can avoid copying the input document as well as performing update on those parts of the document that are not needed by the user query. This technique is illustrated by the example below.

Example 4.2: Recall the user query and the security view defined in Example 4.1. Leveraging the Compose Method, the composition Q_c of the two queries can be written as follows:

```
1. <result> {
2.     for $y1 in part[pname = 'keyboard'],
3.         $y2 in $y1/supplier
4.     let $x := $y2
5.     return if empty($x[country = 'A']) then $x else ( )
6. } </result>
```

Contrast this with the composed query Q'_c given in Example 4.1. Instead of separating the evaluations of the transform query and the user query, Q_c integrates the two queries and can be answered without traversing or copying the entire input document, which is particularly important when operating on a virtual document. □

We first present user queries and then the Compose Method.

User Queries. To simplify the discussion we focus on a simple form of XQuery specified by for, where and return clauses:

for $\$x$ in ρ
where $\rho'_1 = \rho''_1$ and ... and $\rho'_k = \rho''_k$
return $exp(\varrho_1, \varrho_2, \ldots, \varrho_m)$

where (a) ρ is an XPath expression in $\mathcal{X}$; (b) ρ'_j, ρ''_j and ϱ_s are either a text-value constant or are of the form of $\$x/\rho'$, where ρ is an $\mathcal{X}$ expression; (c) exp is an XML *element template*, which is the same as an XML element except that it may contain $\varrho_1, \ldots, \varrho_m$ as parameters; a template yields an XML element given a substitution of concrete XML element for $\$x$. The semantics of the query is standard as defined by XQuery [3].

It should be mentioned that the Compose Method can be extended to handle more general user queries.

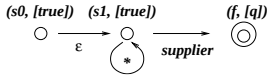


Figure 6: The selecting NFA of the $\mathcal{X}$ query in Example 4.2

The Compose Method. Consider a user query Q and a transform query Q_t, in which XPath expression p in $\mathcal{X}$ is embedded. Given Q_t, Q, the Compose Method finds a composed query Q_c such that $Q_c(T) = Q(Q_t(T))$ for any XML document T. The key idea of the method is to rewrite the XPath expressions ρ, ρ'_i, ρ''_i and ϱ_i in the user query Q based on the $\mathcal{X}$ expression p in Q_t, by simulating the evaluation of p on these path expressions. More specifically, leveraging the selecting NFA M_p of p, we treat the path expressions in Q as "words" and execute M_p on them; when a final state of M_p is reached, we add actions in the composed query Q_c to capture the corresponding "updates" in Q_t. In this way we evaluate both Q_t and Q via a *single* pass of the input XML document.

Below we outline the composition algorithm. We first rewrite the for clause (for $\$x$ in ρ) of Q in terms of M_p. Note that the presence of qualifiers and '//' in *both* the selecting NFA M_p and ρ in the user query Q make the rewriting nontrivial.

Recall from Section 3 that ρ can be rewritten to an equivalent form $\beta_1[q_1]/\ldots/\beta_k[q_k]$, where β_i is either label l, wildcard $*$ or descendant-or-self //, and $[q_i]$ is either a qualifier or [true]. To simplify the rewriting of qualifiers $[q_i]$ based on M_p, we first rewrite the for clause into an equivalent sequences of for clauses:

```
for $y1 in β1          where not empty($y1[q1])   return
  for $y2 in $y1/β2    where not empty($y2[q2])   return
  ...
    for $yn in $y(n-1)/βn where not empty($yn[qn])
    let $x := $yn
```

If either $[q_i]$ is [true] or is disjoint from M_p (to be illustrated shortly), there is no need to have separate where and return clauses in the for loop for $\beta_i[q_i]$, as shown in Example 4.2 (lines 2-4).

For i in $[1, n]$, we rewrite the for loop for $\beta_i[q_i]$ as follows.

• *Computing the states S_i of M_p.* By treating each step β_i as an input "letter" of the NFA M_p, we find the set S_i of states of M_p reached via β_i from the set S_{i-1} of states. As will be seen shortly, we use S_i to determine whether or not we should rewrite the for loop to accommodate the corresponding update operation in the transform query Q_t. The initial set S_0 of states (for β_1) is the ϵ-closure of the start state of M_p. Given S_{i-1}, the set S_i can be computed by using a mild variation of the function nextStates() of Fig. 4: we extend the transition function δ of M_p to define δ' such that (1) $\delta'((s,[q]),*)$ also includes $(s',[q'])$ if $\delta((s,[q]),l)$ contains $(s',[q'])$ for any tag l, and (2) $\delta'((s,[q]),//)$ includes all the states that are reached given an (unbounded) sequence of $*$.

Referring to Example 4.2, the selecting NFA of the transform query Q_t is shown in Fig. 6, in which q denotes *country = 'A'*. The initial set S_0 is $\{(s_0,[\text{true}]),(s_1,[\text{true}])\}$, and S_1, S_2 (for the first and second for loop) are $\{(s_1,[\text{true}])\}$ and $\{(f,[q])\}$, respectively.

• *Handling qualifiers and the final state in S_i.* If a state in S_{i+1} is obtained by applying δ' to a state $(s,[q])$ in S_i and if $q \neq$ 'true', then the qualifier q needs to be checked at this stage. Let C be

the conjunction of all such qualifiers in S_i. We rewrite the return clause of the for loop by adding a conditional statement:

return if empty (y_{i-1}/C) then F_1 else F_2,

where both F_1 and F_2 denote the rest of the query "for $\beta_{i+1}[q_{i_1}]/\ldots/\beta_n[q_n]$ where ... return ...". That is, we separate the treatment (F_2) when C is satisfied from the handling (F_1) when C is false. While we proceed to rewrite F_2 in the same way, F_1 remains unchanged, since the update in Q is not invoked if the qualifiers in C are not satisfied.

Furthermore, if the final state is in S_i, then the corresponding update operation of the transform query Q_t should be incorporated into the composed query Q_c. More specifically, if Q_t is an insert, then we add a let clause before the for clause: "let $z_i := T_i$" and change the for loop to: "for y_i in $z_i/\beta_i[q_i]$", where T_i denotes y_{i-1} incremented by adding the new element e of Q_t as the last child of y_{i-1}, which can be coded in XQuery as shown in Fig. 2. If Q_t is an delete, then T_i is the empty tree '( )'. The update and replace operations are accommodated similarly.

For example, Example 4.2 (line 5) illustrates the processing of qualifier and final state w.r.t. a delete transform query.

Another special case is when S_i is empty, i.e., β_i is *disjoint* from M_p. If so no rewriting is needed for the rest of the query.

• *Processing the qualifier* $[q_i]$ *in the* for *loop.* Along the same lines. the qualifier $[q_i]$ in $\beta_i[q_i]$ is processed by rewriting the where clause of the for loop for $\beta_i[q_i]$. More specifically, each path expression in q_i is treated as a "word" for M_p, ignoring the Boolean operators in q_i. These expressions are processed in the same way as ρ, by introducing necessary for and/or let clauses into the where clause. Again if a path expression is disjoint from M_p, no rewriting is needed. Furthermore, $[q_i]$ is a Boolean query and in some cases can be evaluated to a truth value directly when some path expression in q_i reaches a final state of M_p (see Example 4.3).

For instance, y_1*/part[pname = 'keyboard']* in Example 4.2 is disjoint from M_p and is left unchanged in the composed query.

The rewriting of where and return clauses and their associated path expressions (namely, ρ_i', ρ_i'' and ϱ_i) are carried out in the same systematic way, except the handling of the values to be returned.

• *The value to be returned.* When processing "return $exp(\varrho_1, \varrho_2, \ldots, \varrho_m)$" following the rewriting method given above, we may reduce a path expression ϱ_j to a variable z with a *nonempty* set S of states in M_p. To capture the effect of the transform query Q_t, we need to add a let clause "let $y := $ topDown (M_p, S, Q_t, z)" and substitute y for ϱ_j, where topDown () is the function given in Fig. 3. In this case topDown () is included in the rewritten query as a user-defined function. It should be mentioned that in many cases topDown () is not needed, e.g., when the states in S do not have outgoing edges to self-cycles ('//') in M_p (in this case one can inline the code of topDown () rather than recursively invoking it).

Example 4.3: Following the Compose Method given above, the following pairs (Q_i, Q_i') of transform and user queries can be rewritten into single composed queries Q_c^i in standard XQuery.

Q_1: **transform copy** $r:=doc("f") **modify do delete** $a/b[q]$ **return** $r
Q_1': for $x in $a/b/c$ return $x
Q_c^1:
```
for $y1 in a, $y2 in $y1/b
return if empty($y2[q])
       then for $y3 in $y2/c
            let $x := $y3
            return $x
       else ( )
```

Q_2: **transform copy** $r:=doc("f") **modify do delete** $a/b/c$ **return** $r
Q_2': for $x in a/b where not ($x/c = 'A') return $x

```
function QualDP (L_Q, n, csat_n, dsat_n)
Input: list L_Q of sub-qualifiers, node n in an XML document T,
       and status of qualifiers q in L_Q for n's children (csat_n(q))
       and descendants (dsat_n(q))
Output: sat_n(q), for q ∈ L_Q
1.  for each q in the order of L_Q do
2.    case q of
3.    (1) ε:  sat_n(q) := 1;
4.    (2) ε[q']/p:  sat_n(q) := sat_n(q') and sat_n(p)
5.    (3) */p:  sat_n(q) := csat_n(p);
6.    (4) //p:  sat_n(q) := sat_n(p) or dsat_n(p);
7.    (5) ε = 's':  sat_n(q) := (text() = s);
8.    (6) label() = l:  sat_n(q) := (fn:local-name(n) = l);
9.    (7) q1 ∧ q2:  sat_n(q) := sat_n(q1) and sat_n(q2);
10.   (8) q1 ∨ q2:  sat_n(q) := sat_n(q1) or sat_n(q2);
11.   (9) ¬q1:  sat_n(q) := not sat_n(q1);
12. return sat_n;
```

Figure 7: Algorithm QualDP

Q_c^2:
```
for $y1 in a, $y2 in $y1/b
    let $x := $y2
    return $x
```

Note that the condition in the where clause is already evaluated to true at compile time, taking into account of the deletion.

Q_3: **transfrom copy** $r:=doc("f") **modify do insert** e **into** $a//c$ **return** $r
Q_3': for $x in a/b return $x
Q_c^3:
```
for $y1 in a, $y2 in $y1/b
    let $x := $y2
    return topDown (M_p, S, Q_3, $x)
```

where e is a new element to be inserted, M_p is the selecting NFA of Q_3, and S consists of the states of M_p reached by following a from the start state of M_p, which is computed at compile time.

Note that Q_c^3 includes topDown ( ) as a user-defined query. □

Remark. The size of the final composition query Q_c obtained as above is linear in the sizes of Q_t and Q. Note that Q_c combines the evaluations of Q_t and Q, and does *not* need to make a copy of a tree T when evaluating both Q_t and Q on T. As will be seen in Section 7, Q_c is often more efficient than the composed query computed by using the Naive Composition Method.

5. Handling Expensive Qualifiers in One Pass

The Top Down method (algorithm topDown) of Section 3 has a linear-time data complexity if checkp() can be implemented in constant time. In this section we present an algorithm, bottomUp, that implements checkp(). Taken together with algorithm topDown, algorithm bottomUp yields an implementation of transform queries that is guaranteed to execute in time linear in the size of the document, including the cost of implementing checkp(). Practically, if complex qualifiers are handled well by the XQuery processor, the bottomUp algorithm is not necessary. However, (1) not all processors handle complex qualifiers efficiently [15]; (2) one may use bottomUp for only those qualifiers that are not known to be handled efficiently, and (3) new techniques will be introduced in the next section to efficiently evaluate transform queries on large XML documents, and these techniques extend bottomUp.

In a nutshell, given a transform query Q_t over an XML tree T, bottomUp evaluates all the qualifiers in the XPath expression p embedded in Q_t via a single bottom-up traversal of T, and annotates nodes of T with the truth values of related qualifiers. Given the annotations, at each node checkp() takes constant time to check the satisfaction of a qualifier at the node.

Qualifiers and Sub-Qualifiers. In the algorithm below, we deal with a list of qualifiers L_Q that includes not only all the qualifiers appearing in p, but also all *sub-expressions* of these qualifiers. Fur-

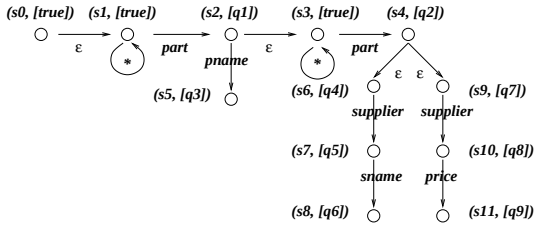


Figure 8: An example filtering NFA of an $\mathcal{X}$ query

thermore, L_Q is topologically sorted such that for any expression e in L_Q, if s is a sub-expression of e, s appears before e in L_Q. To simplify the presentation we adopt a "normalized" form of $\mathcal{X}$ qualifiers such that each path p in a qualifier is of the form η/p' where η is one of $*$, // or $\epsilon[q]$, and p' is a path. This normalization can be achieved by using the following rewriting rules: (1) l to $*/\epsilon[label() = l]$; (2) $p[q]$ to $p/\epsilon[q]$; (3) $p[q_1]\ldots[q_n]$ to $p[q]$ where $q = q_1 \wedge \ldots \wedge q_n$; and (4) $p =$'s' to $p[\epsilon =$'s'$]$. The normalization process can be done in $O(|p|)$ time.

Example 5.1: For the $\mathcal{X}$ query p_1 given in Example 3.1, the list L_Q contains the expressions q_3 = *[ϵ='keyboard']*, q_1 = *[pname[q_3]]*, q_6 = *[ϵ = 'HP']*, q_5 = *[sname[q_6]]*, q_4 = *[supplier[q_5]]*, q_9 = *[ϵ < 15]*, q_8 = *[price[q_9]]*, q_7 = *[supplier[q_8]]* and $q_2 = [\neg q_4 \wedge \neg q_7]$. Note that all expressions are in the normal form mentioned above, and sub-expressions appear before their containing expressions. □

Dynamic Programming. A key step of bottomUp is the evaluation of qualifiers. It is done based on dynamic programming, as follows. Assume that the truth values of all the qualifiers q in L_Q are already known for (1) the immediate children of n (denoted by $\mathsf{csat}_n(q)$), and (2) for all the descendants of n excluding n ($\mathsf{dsat}_n(q)$). Then, in order to compute the satisfaction of the qualifiers at n, denoted by $\mathsf{sat}_n(q)$, it suffices to do a *constant amount* of work *per qualifier*, as summarized in function QualDP() in Fig. 7.

Special care is needed for this recursion to work when computing $\mathsf{sat}_n(q)$ at leaf nodes n of the tree. To do this, we define $\mathsf{csat}_\perp(q)$ (resp. $\mathsf{dsat}_\perp(q)$) such that it is false when q ranges over expressions of the form $*/p$; otherwise it is computed in the same way as in QualDP().

One can verify that the truth values for *all qualifiers* in L_Q can be computed in time $O(|Q_t|)$ at any node in a tree T.

Filtering NFA. Another key issue for bottomUp is to determine the list L_Q of qualifiers to be evaluated at each node of T. To do this we introduce a notion of filtering NFA. Given an $\mathcal{X}$ query p, we construct a NFA, referred to as the *filtering* NFA of p and denoted by M_f, which is an extension of selecting NFAs used in topDown. In a nutshell, M_f is built on both the selecting path and the qualifiers of p, stripping off the logical connectives in the qualifiers; the states of M_f are also annotated with corresponding qualifiers. We use M_f to keep track of whether a node n is possibly involved in the node selecting of p and what qualifiers are needed at n. The size of the filtering NFA M_f for an $\mathcal{X}$ query p is in $O(|p|)$. We illustrate filtering automata with an example instead of giving its long yet simple definition (which is similar to its selecting NFA counterpart).

Example 5.2: The filtering NFA for the query p_1 of Example 3.1 is depicted in Fig. 8, in which qualifiers $[q_1]$ – $[q_9]$ are given in Example 5.1. □

For a set S of states of a filtering NFA M_f, we use $L_Q(S)$ to denote the list of all qualifiers appearing in the states of S, along with their sub-expressions, properly ordered with sub-expressions preceding their containing expressions.

function bottomUp (M_f, S, n)
Input: node n in an XML document T, filtering NFA M_f with transition function δ, a set S of states in M_f.
Output: a list consisting of (1) head: the node n annotated with $\mathsf{sat}_n(q)$, $\mathsf{rsat}_n(q)$, $\mathsf{rdsat}_n(q)$ for qualifiers q computed from M_f, S, n with its annotated subtree; (2) tail: annotated right siblings of n.

1. $S^+ := \bigcup_{(s,[q])\in S} \delta((s,[q]),*) \cup \delta((s,[q]),l)$;
2. $S' :=$ ϵ-*closure*(S^+);
3. if n has a right sibling
   /* call bottomUp at its first right sibling */
4. then let L_s = bottomUp (M_f, S, following_sibling(n)[1]);
5. else let L_s = [ ];
6. if $S' \neq \emptyset$ /* pruning */
7. then if n has a child
   /* call bottomUp at the left-most child of n */
8. then let L_c = bottomUp (M_f, S', *n/*[1]*);
9. else let L_c = [ ];
   /* left-most child */
10. let n_c = if $L_c \neq$ [ ] then first(L_c) else $\perp$;
11. let n_s = if $L_s \neq$ [ ] then first(L_s) else $\perp$; /* next sibling */
    /* $\mathsf{rsdat}_\perp$ and $\mathsf{rsat}_\perp$ are well-defined */
12. sat_n = QualDP ($L_Q(S')$, n, rsat_{n_c}, rdsat_{n_c});
13. else let L_c = the children of n;
14. return element n' followed by L_s
    where n' has the same label as n and its children consists of
15. { L_c;
16. if $S' \neq \emptyset$
17. then for q in L_Q return
18. $\mathsf{sat}_n(q)$;
19. $\mathsf{rdsat}_n(q)$ as $\mathsf{sat}_n(q)$ or ($n_c \neq \perp$ and $\mathsf{rdsat}_{n_c}(q)$) or ($n_s \neq \perp$ and $\mathsf{rdsat}_{n_s}(q)$);
20. $\mathsf{rsat}_n(q)$ as $sat_n(q)$ or ($n_s \neq \perp$ and $\mathsf{rsat}_{n_s}(q)$);
21. }

Figure 9: Algorithm bottomUp

Bottom Up Computation of Qualifiers. The algorithm, bottomUp, is given in Fig. 9. Its input consists of (1) a node n in T, (2) the filtering NFA M_f for p, and (3) a set S consisting of the M_f states reached after traversing T from the root to the parent of n. Using M_f, S and the label of n, the algorithm computes the new set of states S' (in a manner similar to nextStates() but without calls to checkp()). From these states, the qualifiers $L_Q(S')$ that need to be computed at n are derived and evaluated.

To compute $\mathsf{sat}_n(q)$ the algorithm associates two vectors of boolean values with n:

- $\mathsf{rsat}_n(q)$ holds iff q is satisfied at n or at any *right siblings* of n (if any);
- $\mathsf{rdsat}_n(q)$ holds iff q is satisfied at n, or at a descendant of n, or at a descendant of a right sibling of n.

These vectors have the following properties. Assume that n_c and n_s are the left-most child and the immediate right sibling of n, respectively. Then for $q \in L_Q$, $\mathsf{rsat}_{n_c}(q)$ is true if and only if there exists a child of n that satisfies q and thus $\mathsf{rsat}_{n_c} = \mathsf{csat}_n$. Furthermore, $\mathsf{rdsat}_{n_c}(q)$ is true if and only if there exists a descendant of n at which q is satisfied, thus $\mathsf{rdsat}_{n_c} = \mathsf{dsat}_n$. Observe that $\mathsf{rsat}_n(q)$ and $\mathsf{rdsat}_n(q)$ can be computed based on $\mathsf{rsat}_{n_s}(q)$, $\mathsf{rdsat}_{n_c}(q)$ and $\mathsf{rdsat}_{n_s}(q)$ by their definitions.

Taken together, the algorithm bottomUp first computes the set S' of M_f states reached from S by inspecting the label of n and the transition function δ of M_f (lines 1–2)—these steps mirror nextStates(), but omit the checking of qualifiers. Next, bottomUp calls itself recursively on its right sibling (line 3) and left-most child (line 8), which returns the children list L_c and the list of right sib-

Input: Q_t = **insert** *const-expr* **into** a/p **return** a,
and an XML tree T with root r, indicated by a.
Output: $Q_t(T)$.

1. compute filtering NFA M_f for p, with start state $(s_0, [true])$;
2. S := ϵ-*closure*$(s_0, [true])$ in M_f;
3. let T' = bottomUp (M_f, S, r);
4. compute selecting NFA M_p for p, with start state $(s_0, [true])$;
5. S := ϵ-*closure*$(s_0, [true])$ in M_p;
6. return topDown (M, S, Q_t, root(T'));

Figure 10: Implementation of an insert transform query

lings L_s. It uses QualDP() to compute sat_n (line 13). Finally, bottomUp returns a list (lines 14–21) with an element n' as the head, which has the same label as n, carries children L_c and is annotated with sat_n, $\mathsf{rsat}_n(q)$ and $\mathsf{rdsat}_n(q)$; the tail of the list is the right-sibling list L_s.

It is to cope with the *referential transparency (side-effect free) of* XQ*uery* that we simulate the bottom-up traversal of the XML tree by recursively invoking bottomUp at the left-most child and the immediate right sibling of n, if any. In this way we ensure that each node is visited at most once. Observe that the emptiness check of S' (lines 6) allows us to avoid recursively processing the subtrees that will contribute neither to the node-selecting path of p nor to the qualifiers needed in the node selecting decision. That is, only if S' is not empty, bottomUp are invoked at the children of n and QualDP() is called.

The combined complexity of bottomUp is $O(|T|\ |p|^2)$ in the worst case, and its data complexity is linear in $|T|$. In practice $|p|$ is often small. Like in topDown of Fig. 3, the emptiness check of S' allows us to *prune* unaffected subtrees.

Example 5.3: Consider again p_1 of Example 3.1. Given the root of the tree T_0 of Fig. 1, the filtering NFA of M_f in Fig 8 and the ϵ-closure of the initial state of M_f, the algorithm bottomUp computes $\mathsf{sat}_n(q)$, $\mathsf{rsat}_n(q)$ and $\mathsf{rdsat}_n(q)$ for each node n in T_0 and its related qualifiers q, and returns T_0 annotated with boolean values. Note that, for example, only qualifiers $[q_5]$, $[q_6]$, $[q_8]$ and $[q_9]$ are evaluated at *supplier* elements, rather than the entire $[q_1]$–$[q_9]$.

As another example, given $p' =$ *supplier//part* and the root r of T_0, bottomUp returns T_0 right after checking the immediate children of r, since the filtering NFA for p' reaches no state from r, which has no *supplier* children. □

Algorithm twoPass**.** Putting bottomUp and topDown together, one immediately gets an implementation of transform queries, referred to as twoPass, conducted by invoking bottomUp followed by topDown. For example, the evaluation of an **insert** transform query is shown in Fig. 10 (similarly for **delete**, **replace** and **rename**). Now checkp(q, n) in topDown simply checks $\mathsf{sat}_n(q)$ associated with node n, and thus takes constant time. Since the NFAs M_f and M_p can be computed in $O(|p|)$ time, and topDown, bottomUp are in $O(|T|\ |p|)$ and $O(|T|\ |p|^2)$ time, respectively, the data complexity of the evaluation of Q_t is linear-time in $|T|$.

Remark. The twoPass implementation of transform queries Q_t has several salient features. First, it is *optimal*: the entire computation of $Q_t(T)$ can be done with two passes of T, which, as shown in [19], are necessary for evaluating the embedded XPath query p alone (note that our algorithm is quite different from that of [19]; see Section 8). Second, twoPass can be *readily coded in* XQ*uery* and can thus benefit from the any optimization techniques utilizing the referential transparency property of XQuery. Indeed, the list L_Q and the NFAs can be coded in XML, sat, rsat and rdsat can be treated as XML attributes, and assignment statements can be easily replaced with side-effect free function calls. Third, twoPass can be implemented on top of any existing XQuery engine, *without* relying on the support of XML updates. Finally, in contrast to the conceptual evaluation strategy for transform queries, twoPass does *not* need to copy the entire input document.

6. Integrating Evaluation with SAX Parsing

The evaluation algorithms given so far aim to be implemented in XQuery on top of existing XQuery engines. However, as remarked in Section 1, most XQuery engines employ memory-intensive DOM trees and thus do not handle large XML documents very well. In this section we present another algorithm, referred to as twoPassSAX, which shows that algorithms bottomUp and topDown can be naturally combined with SAX parsing to answer transform queries on large XML documents.

SAX Parsing. A SAX parser reads an XML document and generates a stream of SAX events of five types, whose semantics is self-explanatory: *startDocument*(), *startElement*(n), *text*(t), *endElement*(n), *endDocument*(), where n is an element node and t is a string (PCDATA).

Algorithm twoPassSAX**.** Given a transform query Q_t and an XML document T, the algorithm evaluates $Q_t(T)$ via two passes of SAX parsing on T. In the first pass, the algorithm bottomUp is integrated with an SAX parser to evaluate the qualifiers in Q_t. Making use of the boolean values of qualifiers returned from the first pass, topDown is combined with the second pass of SAX parsing to compute the updated tree.

Integrating bottomUp **with SAX parsing.** We first outline the SAX-based bottomUp algorithm. The algorithm takes as input the same parameters as those of bottomUp (Fig 9). However, instead of returning an annotated XML tree, it produces a list L_d of boolean values (0 or 1) of the top-level qualifiers (excluding sub-qualifiers) in Q_t, and writes it to disk as output. Each of these truth values is associated with a unique id determined by the traversal order of T by SAX parsing, which will be used in the second pass of parsing to identify both the corresponding node and qualifier.

More specifically, the algorithm maintains two variables: a cursor c and a stack S. The cursor c is used to generate the ids for qualifiers; whenever we encounter a qualifier that needs to be evaluated at a node, we assign the value of cursor c as its id and increment c by 1. Each entry of the stack S has the following components: (1) the set of automaton states at element node n, (2) the list of sub-qualifiers L_Q (see Section 5) to be evaluated at element node n, (3) the three vectors sat_n, csat_n and dsat_n to be used to compute the values of (sub-)qualifiers as in the algorithm in Fig. 7, (4) the PCDATA of text children and attributes of node n, if any, which will be used to evaluate (sub-)qualifiers, and (5) the ids of the top-level qualifiers that need to be evaluated at node n. Initially, the cursor c is set to 0 and the stack is empty.

The SAX-based bottomUp algorithm incorporates the state transition of filtering NFA M_f and the evaluation of qualifiers into the processing of each SAX event, as follows.

• *startDocument()*. An entry corresponding to the root of the input XML tree T is pushed onto the stack S, in which the NFA state is the ϵ-closure of the start state s_0 of M_f.

• *startElement*(n). When the start tag of an element node n is encountered, the algorithm computes a new entry of S as follows: (1) computing the set of M_f states reached from the set of states of its parent node, which is stored in the entry at the top of S; (2) identifying the list L_Q of (sub-)qualifiers to be evaluated at node n (see Section 5); (3) for each top-level qualifier that needs to be evaluated at node n, setting its id to be the value of the cursor c and incre-

menting c by 1; and (4) extracting the attributes, if any, of node n. This new entry is pushed onto the stack S and will be used to evaluate (sub-)qualifiers at subsequent SAX events.

• *endElement(n).* When the end tag of n is encountered, the algorithm invokes algorithm QualDP ( ) of Fig. 7 to compute the vector sat $_n$ for the list L_Q of (sub-)qualifiers, which was identified at the event *startElement*(n). Note that at this point, the values of vectors csat $_n$ and dsat $_n$ are available since all the descendant nodes of n have been processed. We then pop the entry corresponding to n off stack S. Now the top entry of the stack corresponds to the parent node of n. The algorithm then updates the values of vectors csat and dsat of the parent node of n with the value of sat $_n$. If there are any qualifiers evaluated at n, the algorithm outputs their truth values and associated ids, which are appended to the list L_d.

• *text(t).* This event occurs after *startElement*(n), but before *endElement*(n), where n is the element containing text t. The text t is stored in the top entry of the stack, which corresponds to n, and will be used to evaluate (sub-)qualifiers at subsequent SAX events.

• *endDocument().* Now the top entry is popped off the stack. This is the last event of the SAX-based bottomUp processing.

Example 6.1: Recall the transform query Q_t given in Example 3.2, its filtering NFA in Fig. 8, and the XML tree in Fig. 1. For *startSocument()*, an entry corresponding to the root *db* is pushed onto the stack. The entry records the set of states $\{s_0, s_1\}$ reached from the start state s_0 via ϵ transitions, while the rest of the entry are empty. At the event *startElement()* of the first *part* node n under the root, another entry is pushed onto the stack, which contains (a) $\{s_1, s_2, s_3\}$: the states reached from $\{s_0, s_1\}$ via *part* and then ϵ transitions; (b) $[q_3, q_1]$: the list of sub-qualifiers to be evaluated at n; (c) q_1: the qualifier that needs to be evaluated as part of the output at n along with its id 0 (the first qualifier encountered); and (d) sat, csat and dsat: the vectors, initialized with *false*. □

Integrating topDown **with SAX parsing.** The bottomUp phase is followed by a second pass of SAX parsing of the document T that incorporates topDown (Fig. 3) processing. The SAX-based topDown algorithm takes as input the same parameters as topDown. Using the list L_d of truth values computed in the first pass, it computes $Q_t(T)$ as output. It also uses cursor c and stack S as variables, where c is to identify the node and qualifier with which a truth value in L_d is associated, and each entry of S simply records the current states of the selecting NFA M_p.

Recall that the SAX-based bottomUp algorithm associates an id with a truth value of a top-level qualifier q at a node n, and that the id is determined by the traversal order of SAX parsing of T. Following the same traversal order on the same document T, SAX-based topDown repeats the same process of assigning ids to qualifier and incrementing the cursor c. Furthermore, note that the selecting automaton M_p used in this phase and the filtering NFA M_f used in the last phase have the same structure when sub-qualifiers and their associated paths are striking out. In other words, M_p and M_f share the same set of top-level qualifiers. Putting these together, one can verify that the value of the cursor c suffices to map the qualifier q at node n to its truth value in the list L_d.

The SAX based topDown algorithm incorporates the state transition of selecting NFA and the computation of $Q_t(T)$ into the processing of SAX events. Due to the space limitation, we only give the integration of topDown processing with one type of SAX events.

• *startElement(n).* When the start tag of an element n is encountered, the algorithm computes the set of states reached from its parent node (the current top entry of the stack), and checks the truth values of top-level qualifiers, if any, at n by looking up the list L_d using the value of the cursor c as an index. A new entry consisting of the set of current M_p states is pushed onto the stack. In this event, the algorithm is required to perform update operation if the final state is reached. For example, for a **delete** transform query, the start tag will not be returned as output. If no update operation is required, the start tag of n together with its attributes will be output.

U1	/site/people/person
U2	/site/people/person[@id = "person10"]
U3	/site/people/person[profile/age $>$ 20]
U4	/site/regions//item
U5	/site//description
U6	/site/closed_auctions/closed_auction/annotation/description /parlist/listitem/parlist/listitem/text/emph/keyword
U7	/site/open_auctions/open_auction[bidder/increase$>$5]/ annotation[happiness $<$ 20]/description//text
U8	/site/open_auctions/open_auction[initial $>$ 10 and reserve $>$50]/bidder
U9	/site/regions//item[location ="United States"]
U10	/site//open_auctions/open_auction [not(@id ="open_auction2")]/bidder[increase $>$ 10]

Figure 11: Embedded XPath queries

Example 6.2: Recall the processing of the query Q_t described in Example 6.1. In the topDown phase, at the event *startElement()* for the first *part* node n under the root, the entry pushed onto the stack consists of the states $\{s_1, s_2, s_3\}$ of the selecting NFA of Fig. 5. The value of the top-level qualifier q_1 at n is extracted from the list L_d of boolean values. This is done by using the current value of the cursor c as the index, which is 0, precisely the same as when q_1 at n was encountered in the bottomUp processing. □

Remark. Observe the following about algorithm twoPassSAX. First, it inherits the nice properties of the twoPass algorithm presented earlier; for instance, twoPassSAX has a linear data complexity and in fact, as demonstrated by our experiments, it takes not much more time than two passes of SAX parsing. Second, compared to DOM-tree based implementation it requires much less memory. Indeed, the size of the stack S is bounded by the depth of the XML document T and the size of each entry of the stack S is linear in the size of transform query Q_t. The list L_d of boolean values is written to secondary storage and in fact, it does not incur much memory overhead even when stored in main memory. Indeed, our experimental results show that it uses no more than 5M memory when processing XML documents of 1.1G. These show that twoPassSAX is capable of efficiently evaluating transform queries on very large documents. It should be mentioned, however, in contrast to the evaluation algorithms given in previous sections, twoPassSAX is not implemented in XQuery.

7. Experimental Study

Our experimental study focuses on the performance of the implementation and composition methods developed in this paper for transform queries. To demonstrate the effects of datasets, transform queries and the size of affected areas in a document by transform queries, we ran a set of experiments on Qizx [1] and GalaX [13]. Due to space constraint we only report the experimental results *on Qizx* that is faster than Galax on our queries; the results on GalaX are comparable. It should be remarked that although we chose Qizx and GalaX to conduct our experiments, our techniques are also *applicable to other* XQ*uery 1.0 engines*.

We used datasets generated by XMark [24]. We generated a set

[1] http://www.axyana.com/qizxopen.

of XML files by varying XMark scaling factors between 0.02 and 0.34, to obtain files of size 2.22M, 11.1M, 19.9M 29.1M, 37.8M respectively. We also generated five files from 224M to 1.1G by varying XMark scaling factors from 2 to 10. The results presented here are mainly based on **insert** transform queries; we have found that transform queries of *the other types consistently yield qualitatively similar results*.

The experiments were performed on a PC with a Pentium IV 2.4 Ghz CPU and 500MB RAM, running Linux. Each experiment was repeated 5 times and the average is reported here; we do not show confidence interval since the variance is smaller than 5%.

We evaluate our algorithms for evaluating transform queries implemented both in XQuery and as part of a query processor in Section 7.1, and composition of user and transform queries in Section 7.2. We give a summary and discussion in Section 7.3.

Note that except for the SAX based two-pass algorithm all of our algorithms are coded *in* XQ*uery* and *implemented on top of Qizx*.

7.1 Transform Query Evaluation

We first evaluate our proposed techniques for implementing transform queries in XQuery, namely, the rewriting-based Naive Method of Section 3.1, the TopDown method of Section 3.3 and the Two-pass method of Section 5, denoted as NAIVE, GENTOP and TD-BU, respectively. We also consider the SAX based Two-pass algorithm of Section 6, denoted as twoPassSAX, and an implementation of transform query in GalaX (which supports XML updates and transform queries), denoted by GalaXUpdate. We should remark that Qizx supports neither updates nor transform queries.

Based on the benchmark queries of XMark [24], we designed 10 insertion transform queries, which differ only in the XPath expressions used to select the target nodes. The embedded XPath expressions are shown in Fig. 11. In transform queries U1-U3, the paths contain at most one simple qualifier and no descendant axis. Transform queries U4 and U5 have descendant axis, U6 has a long path, U7 and U8 have a relatively complicated qualifier, and transform queries U9 and U10 contain both descendant axis and qualifiers.

Figure 12 shows the running time of each of the five methods for evaluating transform queries, for U1-U10, using the dataset of size 2.22M. As shown in this figure, twoPassSAX performs the best in all five methods while GENTOP usually runs the fastest among the three algorithm implemented on top of Qizx. Both twoPassSAX and GENTOP outperform the GalaXUpdate.

While NAIVE does reasonably well on U2-U3, U6 and U10, it fares worse on queries when the set of selected nodes is large (i.e., when $|\$xp|$ in Fig. 2 is large), as expected. For example, in U2, NAIVE performs well due to $|\$xp| = 1$ while its performance deteriorates greatly in U1 since $\$xp$ is the set of all *person*s in the XML file. The Top Down method improves on NAIVE since they use the NFA to prune the search space and avoid the lookup in $\$xp$.

TD-BU handles qualifiers with algorithm bottomUp while the other two methods on top of XQuery engines utilize the native processing ability of Qizx. As shown in these figures, although TD-BU pays a price for implementing the complexity of bottomUp on top of XQuery, TD-BU is usually comparable with GENTOP when the qualifiers are relatively simple, e.g., in U1-U6. When the qualifiers get complicated, e.g., in U7 and U8, or when transform queries contain both descendant axis and qualifiers, e.g., in U9 and U10, the performance disparity between TD-BU and GENTOP becomes larger. This disparity may be emphasized by Qizx, which handles a wide variety of qualifiers very efficiently according to our experimental study. Of course, our top-down variants can easily switch between native qualifier evaluation and bottomUp, and we believe TD-BU may still provide advantages if XQuery processors with different performance tradeoffs are encountered.

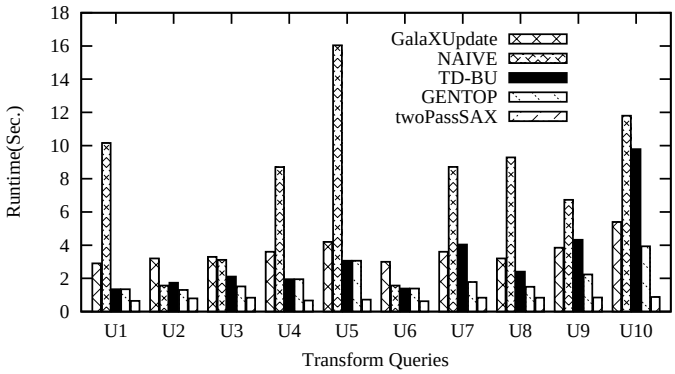


Figure 12: Execution time

Scalability with data size. To show scalability, we chose several representative transform queries, while the other transform queries display properties similar to the four selected queries. Figure 13 shows that, as expected, NAIVE does not scale well with XML file size when the portion of the file affected by the transform queries is significant, while the two methods implemented on top of Qizx, TD-BU and GENTOP, are linearly scalable. twoPassSAX is also linearly scalable. This is consistent with our discussions in the previous sections. When the size of the set of selected nodes $|\$xp|$ is fixed with the change of file size, the Naive method will be linearly scalable, such as U2 in Fig. 13(a). GalaXUpdate ran out of memory on data with factor 0.26 on queries U2, U4 and U7, and on data with factor 0.18 on query U10 (in Fig. 13(d), GalaXUpdate works at the first two data points and is a bit slower than TD-BU; the line for GalaXUpdate partly overlaps with that for TD-BU). It appears that Galax implements transform queries by taking a snapshot of XML files while our techniques *do not make a copy (snapshot)* of XML files and have smaller memory overhead.

Handling large data. To show the feasibility of SAX based Two-pass algorithm for handling large datasets using small memory, we evaluate it on five files of size varying from 224M to 1.1G using four representative queries, U2, U4, U7 and U10. Figure 14 shows that algorithm twoPassSAX scales well on all the transform queries. The memory consumption of twoPassSAX is independent of file size. We observed that twoPassSAX used less than 5M memory while *none* of the other four algorithms ran to completion on data of such sizes.

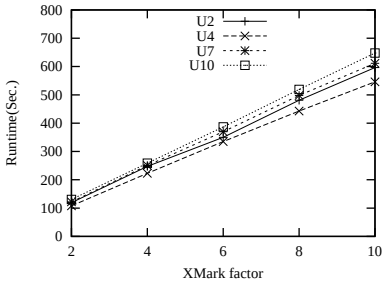
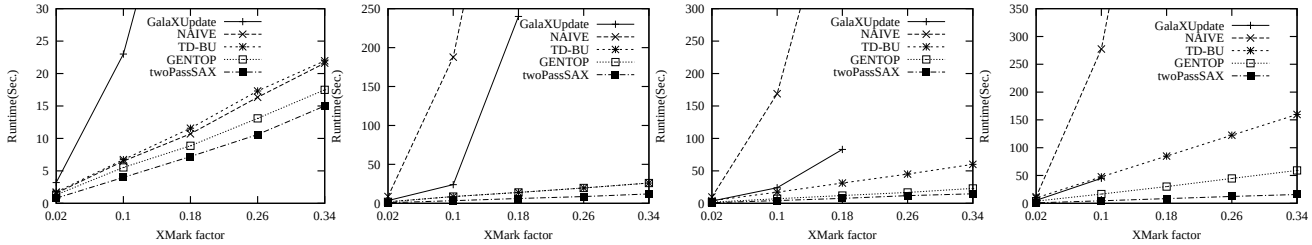


Figure 14: The SAX-based algorithm: scalability with file size

7.2 Composition of User and Transform Queries

We next evaluate our proposed techniques for composing a user query and a transform query, namely the Naive Composition method and the Compose method presented in Section 4. For the Naive Composition method, we adopt the algorithm GENTOP to



(a) U2 (b) U4 (c) U7 (d) U10

Figure 13: Scalability with the file size

evaluate the transform query since it outperforms the other algorithms implemented on top of Qizx as reported in Section 7.1.

Using the XPath queries in Fig. 11, we generated four representative pairs of transform and user queries, (U1, U2), (U9, U1), (U9, U4) and (U8, U10), where U1 and U9 in the first two pairs are **insert** transform queries and U9 and U8 in the last two pairs are **delete** queries, while U2, U1, U4 and U10 are their respective user queries.

Figure 15 shows the runtime of four composed queries on XML files of various sizes. The algorithm Compose performs consistently better than the Naive Composition algorithm, and the improvement is especially significant on larger XML files as shown in all the four graphs. There are two reasons for the improvement: first, algorithm Compose integrates the evaluation of the user and transform queries; second, it avoids computing part of updates in transform queries on documents that are not needed by the user queries. The second reason is dominating for the improvement of algorithm Compose over Naive Composition on the pair (U9, U1) as shown in Fig. 15(b), where the user query U1 is largely disjoin from transform query U9 and the update in the transform query U9 is not required when evaluating user query U1. Note that the query results from these user queries are much smaller in size than those of their corresponding transform queries, and thus take less time to produce. Figure 15 also shows that both algorithms are linearly scalable while algorithm Compose scales better.

7.3 Summary and Discussion

Our experimental results demonstrate the practicality of efficiently supporting transform queries by existing XQuery engines. We show that our proposed algorithm GENTOP using the native predicate evaluation facility of Qizx performs the best. Moreover, GENTOP and TD-BU are linearly scalable to XML file size. We also show the ability of twoPassSAX to handle large files. For the composition of user and transform queries, our experimental results show that algorithm Compose is more efficient than Naive Composition while both algorithms scale well with the file size.

For our proposed techniques implemented on top of XQuery engine, we expect that the performance will be improved as XQuery processors mature, for example in I/O management and handling recursive queries in XQuery.

We note that algorithm twoPassSAX implemented as part of a query processor scales very well on large XML documents.

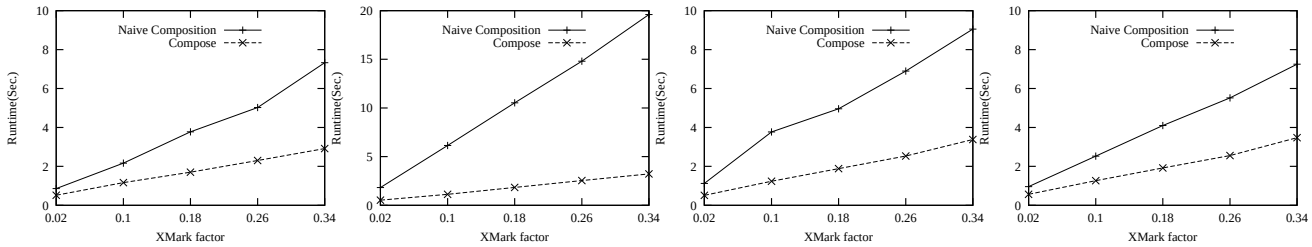
8. Related Work

Transform queries are proposed by W3C XQuery Update [6] and XQuery! [14]. To our knowledge, the only implementation of transform queries so far is Galax [13]. No published work has addressed technical issues in connection with efficient evaluation of transform queries either on top of or within XQuery engines, or composition of transform queries and user queries. Prior work on XML updates has mostly focused on proposals for update languages [20, 21, 23, 25]. While there has been work on implementing XML updates on XML data stored in relational DBMS [26] or in native stores [18], the problems tackled there are to perform updates in place, which is entirely different from the idea of transform queries.

The idea of defining queries as updates has been well studied for traditional databases, notably hypothetical queries (recall from Example 1.1) and their evaluation techniques [1, 4, 12, 16]. The importance of hypothetical queries has long been recognized. When it comes to the implementation of hypothetical queries, it is already observed that while rewriting updates into relational queries is conceptually easy in absence of recursion in updates [1], the presence of recursion in updates makes the problem highly nontrivial. Indeed, [4] shows that datalog extended by hypothetical insertions and deletions has EXPTIME data complexity as opposed to the PTIME data complexity of datalog. In this work we study transform queries defined in terms of limited recursive XML updates (i.e. descendant-or-self::node() in embedded XPath queries). Rewriting and optimization techniques have been developed for hypothetical queries on relational databases [16]. These, however, do not apply directly to the evaluation of transform queries due to the tree data model of XML and the XQuery target language.

There is an analogy between transform queries and functional updates. Functional databases enforce *referential transparency* and support updates via versioning [27], which allows interesting optimization, e.g., parallel evaluation, that are also applicable to XML transform-query evaluation. Also related to transform queries are restructuring queries that can be viewed as updates [5], and a variation of XPath queries studied in [2]. While the queries of [2] are a subset of transform queries defined in terms of **delete** (starting from the root of a source document), no specific evaluation or optimization techniques are published for their queries.

Several automaton-based evaluation algorithms have been developed for XPath, e.g., tree automata of [19], alternating finite state automata (AFA) of [17], and a form of NFA [8, 9]. Our proposed selecting and filtering NFA (Sections 3 and 5) differ from these automaton machinery in several aspects. As opposed to [19], the automata used in this work do not require to transform an XML tree to a binary-tree representation and moreover, they are linear in the size of input queries in contrast to possibly very large bottom-up and top-down tree automata of [19]. The AFA of [17] and NFA of [8, 9] aim at evaluating *Boolean* multiple XPath queries on stream XML data treated as SAX events. The AFA [17] are required to be deterministic and are thus possibly quite large, and the NFA [8, 9] do not handle nested qualifiers. In contrast, our automata deal with *transform queries* that subsume a larger class of *node-selecting* XPath queries, they work on both SAX events and DOM trees; furthermore, they are much less costly than the AFA of [17] and are capable of handling complex qualifiers as opposed to [8, 9]. It is worth remarking that the implementation of our transform queries combines the evaluation of XPath and the handling of XML updates,



(a) (U1, U2) (b) (U9, U1) (c) (U9, U4) (d) (U8, U10)

Figure 15: The composition of user and transform queries

which has not been considered by any previous work for XPath.

To the best of our knowledge no previous work has considered composition of transform queries and user queries (in XQuery). Composition of simple XQuery queries and canonical XML views of relational data was studied in [11]. In contrast to our composition algorithm of Section 4, the algorithm of [11] considers neither queries with embedded updates, nor complicated XML views (e.g., '//' or qualifiers). Recently rewriting of XQuery using views is studied in [22], which is an entirely different problem. No previous work on query composition has studied automaton-based rewriting, which we explore in this work.

9. Conclusion

We have proposed several techniques for evaluating transform queries and for composing user queries with transform queries, which can be readily implemented on top of *any existing* XQ*uery engines*. These provide an *immediate capability* for these engines to support transform queries. We have also developed a scalable solution to evaluating transform queries as part of a query engine. This allows us to deal with *large* XML *documents* beyond what most publicly available XQuery engines using DOM trees can handle. Our experimental results verified that these techniques are effective and efficient. These techniques are useful in supporting hypothetical queries, updating virtual views, composing queries and updates, and enforcing XML access control, among other things.

For future work, first, we plan to evaluate the performance of our proposed techniques on other XQuery engines (e.g., http://monetdb.cwi.nl/XQuery/Overview/Benchmark). Second, we recognize the challenges of efficiently processing transform queries defined with more involved updates [6, 14], which are a subject of our ongoing work. It is worth noting that complex XML updates actually demand a seamless integration of update and query processing in a uniform framework, and thus advocate the need for techniques to efficiently support transform queries. Third, we plan to extend our composition techniques to work with the SAX based two-pass algorithm. Finally, one issue that is not discussed in the paper is concurrency control, and we plan to study the impacts of transform queries on transaction management.

10. References

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] M. Benedikt and I. Fundulaki. XML subtree queries: Specification and composition. In *DBPL*, 2005.

[3] S. Boagi, D. Chamberlin, M. F. Fernndez, D. Florescu, J. Robie, and J. Simon. XQuery 1.0: An XML Query Language. W3C Candidate Recommendation, June 2006.

[4] A. J. Bonner. Hypothetical datalog: Complexity and expressibility. *Theoretical Computer Science*, 76(1):3–51, 1990.

[5] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. In *SIGMOD*, 1996.

[6] D. Chamberlin, D. Florescu, and J. Robie. XQuery Update. W3C working draft. http://www.w3.org/TR/xqupdate/.

[7] O. Corp. Oracle Database SQL Reference 10g Release 2. Product Documentation.

[8] Y. Diao, P. M. Fischer, M. J. Franklin, and R. To. YFilter: Efficient and scalable filtering of XML documents. In *ICDE*, 2002.

[9] Y. Diao, P. M. Fischer, M. J. Franklin, H. Zhang, and P. M. Fischer. Path sharing and predicate evaluation for high-performance XML filtering. *TODS*, 28(4):467–516, 2003.

[10] W. Fan, C. Y. Chan, and M. Garofalakis. Secure XML querying with security views. In *SIGMOD*, 2004.

[11] M. F. Fernandez, Y. Kadiyska, D. Suciu, A. Morishima, and W. Tan. SilkRoute: A framework for publishing relational data in XML. *TODS*, 27(4):438–493, 2002.

[12] D. Gabbay and U. Reyle. N-prolog: An extension of prolog with hypothetical implications. *J. of Logic Program.*, 1(4):319–355, 1984.

[13] Galax Project: UpdateX. *http://www.galaxquery.org/*.

[14] G. Ghelli, C. R, and J. Simon. XQuery!: an XML query language with side effects. In *EDBT's DataX Workshop*, 2006.

[15] G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. In *VLDB*, 2002.

[16] T. Griffin and R. Hull. A framework for implementing hypothetical queries. In *SIGMOD*, 1997.

[17] A. Gupta and D. Suciu. Stream processing of XPath queries with predicates. In *SIGMOD*, 2003.

[18] H. Jagadish et al. TIMBER: A native XML database. *VLDB Journal*, 2002. *http://www.eecs.umich.edu/db/timber/*.

[19] C. Koch. Efficient processing of expressive node-selecting queries on XML data in secondary storage: A tree automata-based approach. In *VLDB*, 2003.

[20] A. Laux and L. Martin. XUpdate - XML Update Language, 2000. *http://www.xmldb.org/xupdate/xupdate-wd.html*.

[21] P. Lehti. Design and implementation of a data manipulation processor for an XML query processor. Technical report, Technical University of Darmstadt, Diplomarbeit, 2001.

[22] N. Onose, A. Deutsch, Y. Papakonstantinou, and E. Curtmola. Rewriting nested XML queries using nested views. In *SIGMOD*, 2006.

[23] M. Rys. Proposal for an XML data modification language. Microsoft, 2002.

[24] A. Schmidt, F. Waas, M. Kersten, M. Carey, I. Manolescu, and R. Busse. XMark: A benchmark for XML data management. In *VLDB*, 2002.

[25] G. Sur, J. Hammer, and J. Siméon. An XQuery-based language for processing updates in XML. In *PLAN-X*, 2004.

[26] I. Tatarinov, Z. G. Ives, A. Y. Halevy, and D. S. Weld. Updating XML. In *SIGMOD*, 2001.

[27] P. Trinder. *A Functional Database*. PhD thesis, Oxford Univ, 1989.