Adding Updates to XQuery: Semantics, Optimization, and Static Analysis

Michael Benedikt[1] Angela Bonifati[2] Sergio Flesca[3] Avinash Vyas [1]

[1]Bell Laboratories
Lucent Technologies
{benedikt,vyas}@research.bell-labs.com

[2]Icar CNR
Via P. Bucci 41C, 87036 Rende, Italy
bonifati@icar.cnr.it

[3]D.E.I.S., University of Calabria
Via P. Bucci 41C, 87036 Rende, Italy
flesca@deis.unical.it

Abstract

The need to extend XQuery to support updates has been recognized both in the research and the standards community. Several language proposals and prototype implementations have been put forward, and update language requirements are being defined within the W3C. Most proposals center around the use of update primitives applied to XQuery expressions, along with a variant of the FLWR loop construct binding variables within a block of basic update statements. In defining a precise semantics for such statements a number of issues arise: one must decide how conflicts among updates are to be resolved, and how query evaluation interacts with update application. In this work we provide a framework for defining alternative semantics for updates, and identify within this space what is (thus far) the consensus choice: that semantics involves a two-stage execution process, in which query evaluation is performed first, after which a generated sequence of concrete updates is applied in a fixed order determined by query output. This results in a clean deterministic semantics which facilitates analysis. A drawback is that the evaluation of the language can be inefficient. One would prefer to perform updates eagerly before further evaluation, or to re-order the update operations. We focus on an optimization of the "standard semantics", in which updates are performed as soon as they are generated. We present a static analysis for determining when this optimization can be exploited. Experiments on the implementation of this analysis, implemented on top of Galax, show that the overhead is minimal.

1 Introduction

Specification and processing of bulk updates is a critical data management task for XML documents. While updates to XML have long been implementable in node-at-a-time fashion within navigational interfaces such as DOM [Dom98], languages for specifying bulk updates are emerging. Several language proposals based on extensions of XQuery have been put forward [WS02, WS00, SIGMOD01, PlanX04], and the World Wide Web consortium is well underway to extend XQuery [XQuery04] with the capability of expressing updates over XML data. Two sample declarative update programs, in the syntax of [PlanX04] are shown below:

```
U1:update
   for $i in //openauction
    insert $i/initial/text() into $i/current
    delete $i/bidder[@increase=3000]

U2:update
   for $i in /site//item
    insert <incategory/> into $i
    replace $i//mail/to/text()
      with $i//mail/from/text()
```

These examples are both built upon an instance document conforming to the XMark [VLDB02] DTD. A document satisfying this DTD is shown in Figure 4.

Informally, example U1 states that after the update each $open\_auction$ element should have the value of $current$ equal to the value of $initial$, and that after the update each $bidder$ element having an increase of three thousand and lying below a $open\_auction$ element should be deleted. The effect of this program over an instance, is shown in Figure 1(a) and (b).

Previous update language proposals differ in many details, but they agree on a critical semantic issue regarding how program evaluation is to be ordered. These proposals generally center upon the *generation-order, snapshot semantics* [PlanX04], which specifies the use of two logical phases of processing: in the first all evaluation of XPath and XQuery expressions is done, yielding an ordered set of point updates. In the second phase the sequence of updates is applied in the specified order. In example U1 above, a sequence consisting of alternating inserts and deletes will be generated, based on the document order of the results of $i,$i/initial/text(), $i/bidder, and this sequence will be applied in that order to the document.

This semantics has a number of attractive features; it is more consistent with the semantics of declarative relational update languages such as SQL, and it averts the possibility of ill-formed reads arising at runtime. We shall see further that the use of the semantics allows one to do reasoning on programs that is impossible in a more iterative semantics. The main drawback of the semantics is that the naive implementation of it is very inefficient. In a straightforward implementation the intermediate results must all be materialized before any writes are performed, rather than a more pipelined chaining of reads to subsequent writes. Secondly, the standard semantics requires application of the updates in the ordering generated by the queries, while other orders may result

Second International Workshop on XQuery Implementation, Experience and Perspectives (XIME-P 2005),
Baltimore, Maryland USA 2005

in more locality of access and more efficient renumbering of indices.

We consider several optimizations to the standard evaluation. The first optimization revolves around changing the evaluation order in which concrete updates are applied, and instead doing generated deletes before inserts. This optimization is not necessarily sound, and we state some results from [TR05] concerning the feasibility of checking soundness. The second optimization involves relaxing the requirement that evaluation is done before application of updates, and doing instead an *eager evaluation*. The eager evaluation is also not sound for all updates. We review the technique of [CAV05] for determining soundness of eager evaluation, which involves checking for "conflicts" between evaluation and update. We have implemented this soundness check on top of the Galax update engine, and we present experiments to show that our analysis can detect optimization opportunities that arise in many common updates. Since bulk updates are often defined well in advance of their use, and can be used to specify computing-intensive modifications to data that may take minutes or even hours on current update processors, the use of offline analysis is particularly attractive. In example U1, our analysis detects that both order-optimization and eager-optimization are applicable, while in U2 it detects that the eager-optimization is applicable.

Organization. Section 2 presents the update language studied in this paper, a variant of the language of [PlanX04], and then presents the semantics in three steps: primitive updates, single-step update reduction, and evaluation order. Section 3 gives optimizations based on re-ordering and eager optimization, and then overviews the static analysis that checks their soundness. Section 4 describes the implementation of our static analysis and runtime engine and presents experimental results both on the static analyses and the impact of optimization. Section 5 gives conclusions and ongoing research, while also reviewing related work.

2 XML and XML Update Languages

In this section, we briefly review the syntax of the XML update language we study in the paper and discuss its semantics.

2.1 Update Language for XML

We review here the syntax of the XML update language UpdateX we use throughout the paper, based on [PlanX04]. The top-level update-specific constructs we deal with are presented in Table 1.

ComplexUpdate ::= **FLWUpdate** | **ConditionalUpdate**

FLWUpdate ::= "UPDATE" (**ForClause** | **LetClause**)+ **WhereClause**? **SimpleUpdate**+

ConditionalUpdate ::= "UPDATE" "IF (" **XQueryExpr** ") THEN" **SimpleUpdate** "ELSE" **SimpleUpdate**

Table 1: Update statements syntax

where **XQueryExpr**, **ForClause**, **LetClause** and **WhereClause** refer to productions[40, 43, 45, 46] drawn from the XQuery syntax specification [XQuery04], while the non-terminal **SimpleUpdate** is defined in Table 2.

```
SimpleUpdate ::=  ( "insert" cExpr
                       ("after" | "before") tExpr ) |
                  ( "insert" cExpr ("as first" |
                       "as last")? "into" tExpr ) |
                  ( "delete" tExpr ) |
                  ( "replace" tExpr "with" cExpr )
```

Table 2: Simple updates

In the table, **tExpr** is an XPath expression, while **cExpr** is an XQuery expression. Intuitively, **tExpr** computes the target location where the update is taking place, while **cExpr** constructs a new document fragment which is to be inserted or replaced at the target of the update.

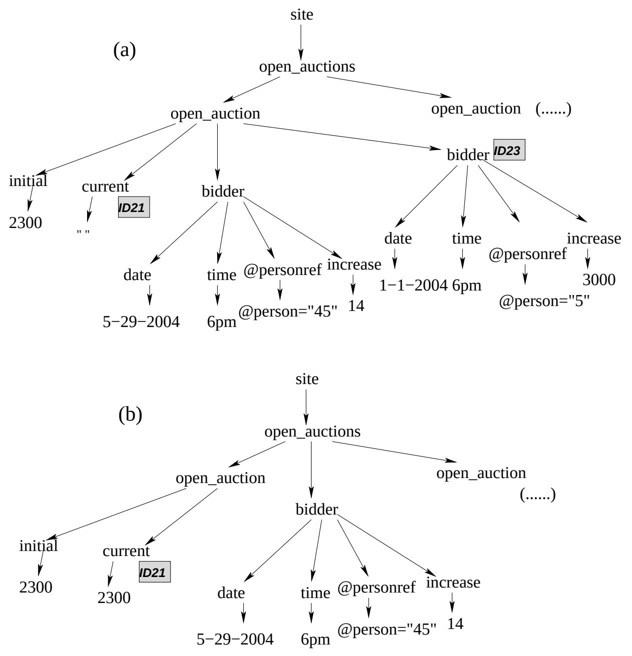

Figure 1: An example XML document, before (a) and after (b) the update U1.

2.2 Semantics of XML Updates

We reformulate the semantics of updates as proposed in [PlanX04, SIGMOD01] both to establish notation and to highlight some details that were not emphasized in earlier work. We need this formalization both in order to rigorously describe the existing semantics, and as a basis for exploring optimizations and alternatives.

The semantics of all existing update proposals [SIGMOD01, WS02, PlanX04, ICDE01] consists first of a description of how individual updates apply, and secondly of how query evaluation is used to generate an ordered sequence of individual updates. Following this pattern, we will first introduce our concrete update operations, and then discuss the order in which they are generated. This second discussion will itself consist of two components, one defining single transition steps on complex updates, and the second defining the evaluation strategy in which steps are executed.

Concrete update API. Concrete updates correspond at a more abstract level to the data model update primitives presented in [PlanX04] and mirrored in the Galax data model update API interface. Let D be an XML document, f be a forest (ordered sequence of XML documents), and n a node identifier, then a *concrete update* u, applied to the XML document D, is one of the following operations.

- $u = \text{InsAft}(n, f)$ or $u = \text{InsBef}(n, f)$: the operation returns a new document, such that, if $n \in D$, each tree in f is inserted immediately after (before) the node with id n in its parent node, in the same order as in the forest f.
- $u = \text{InsInto}(n, f)$: the operation returns a new document such that, if $n \in D$, the trees in f are inserted after the last child of the node given by n.
- $u = \text{Del}(n)$: if $n \in D$, the operation returns a new document obtained from D by removing the sub-document rooted in the node associated with n.
- $u = \text{Replace}(n, f)$: the operation returns a new document D_n, such that, if $n \in D$, the trees in f replace the sub-document rooted in the node of n (in the ordering given by f).

To complete the semantic definition above, we need to address what to do if $n \notin D$ in each case. Previous proposals differ on this issue (or leave it unspecified). One possibility, which we call the *lenient API* has each of the operations be the identity in this case. A second possibility, which we call the *strict API*, aborts the update in this case. We can consider this to mean that a particular value "abort" is returned; when our UpdateX programs generate an API call that returns this value, they will be required to return abort as well. The Galax 0.3.5 implementation chooses this strict option, while 0.4.0 and 0.5.0 use the lenient API. Of course, there are a range of possibilities in between. One plausible middle ground is the API in which delete operations with non-referring nodeIds have no effect, but insert or replace operations on such nodeIds abort. We refer to this as the *standard API* (for lack of a better term – it is certainly not a standard) and we will use it as the default here.

Each concrete update implicitly entails several operations that keep the consistency of the output document. For example, since node insertions and replacements are performed under the assumption that the sets of nodeIds in the inserted and replace documents are disjoint – if this is not the case an assignment of fresh nodeIds to nodes must be performed.

Single-step processing of update statements. Generating a sequence of concrete updates from either a simple or complex update program defined in Table 1 is built upon transitions that evaluate expressions to produce a sequence of concrete updates. We now define these transitions.

Our single-step transitions transform the state of the program during processing . We can characterize this state by means of a pair consisting of the current document and a sequence of partially-evaluated updates still to be processed. An *expression binding* for an update u is a mapping associating a set of tuples to occurrences of XPath expressions in u. A tuple will be either a nodeId in the original document or an XML tree constructed from the original document (e.g. a copy of the subdocument below a node).

A *bound update* is a pair (b, u) where u is a non-terminal **UpdateStatement** and b is an expression binding for u.

We are now ready to define the *update reduction operator* $[\cdot]$, which takes a bound update and produces either a sequence of bound updates and concrete updates, or aborts. We refer to such a sequence as above as a *pending update sequence*.

- $[p]$ for a bound update $p = (b, u)$ is defined as follows. If u is a FLWUpdate of the form for var in $E\ u'$ (that is, for var in$E$ is the initial loop in the outermost ForClause), we form $[p]$ by evaluating E to get nodes $n_1 \ldots n_k$, and returning the sequence whose i^{th} element is (b_i, u') , where b_i extends b by assigning var to n_i. There is a corresponding reduction step for evaluating the where and let clauses, as well as the conditional expressions, which we omit for brevity.
- If u is a block of SimpleUpdates $u_1 \ldots u_l$, then $[p]$ returns $(b, u_1) \ldots (b, u_l)$.
- If u is a single simple update and some expressions in u do not yet have bindings, $[p]$ is formed by first evaluating the XQuery expressions in u (in case of replace or insert) to get a forest, and evaluating the target expressions in u to get one or more target node identifiers. We then proceed as follows:
  - for an insert or replace if the target expression evaluates to at most one node, then $[p]$ is (b', u), where b' extends b by binding the remaining variables according to the evaluation just performed. What happens when the target expression evaluates to multiple nodes is also a question of strictness. Following [PlanX04], we say that the result of $[p]$ is abort in this case. If the lenient API is used, it would be more natural to have the reduction operator return the empty sequence in this case.
  - For a delete, let nodeIds $n_1 \ldots n_l$ be the result of evaluation of the target expression. $[p]$ is $(b_1, u) \ldots (b_1, u)$, where b_i extends b by assigning the target expression of the delete to n_i.
- Finally, if $p = (b, u)$ is a bound update in which u is simple and every expression is already bound, then $[p]$ is simply the concrete update formed by replacing the expressions in u with the corresponding nodeId or forest given by the bindings.

Note that the update reduction operator is concerned only with reducing a single update statement to a sequence of simpler ones; it does not actually apply any updates.

Processing order for complex updates. We are now ready to define the semantics of programs, using two kinds of transitions acting on a *program state*, which consists of either a document and a pending update sequence, or the keyword abort (for the standard or strict API).

An *evaluation step* on a program state $(D, \text{us} = p_1 \ldots p_n)$ is a transition to (D, us') where the new sequence us' is formed by picking a pair $p = (b, u)$ and replacing p by $[p]$ in the sequence. If ps is the program state before such an evaluation step and ps' is the result of the step, we write $\text{ps} \rightsquigarrow^e_p \text{ps}'$.

For example, the processing of the update u = `for $x in A/B insert $x/C into /B` on document D at program state $ps_0 = D, \{p_0 = (\emptyset, u)\}$ would proceed as follows.

We start with transition $ps_0 \rightsquigarrow^e_{p_0} ps_1$, with program state ps_1 being:

$D, \quad \{p_1 = (\langle \$x : i_1 \rangle, \texttt{insert \$x/C into /B});$
$\qquad p_2 = (\langle \$x : i_2 \rangle, \texttt{insert \$x/C into /B})\ \}$

where the nodeids $\{i_1, i_2\}$ are the result of evaluating A/B. For ease of readability, in the remainder we substitute for each occurrence of a variable $\$x$ the nodeids $\{i_1, i_2\}$ and similarly for path expressions rooted in variables. We would then have the transition:

$ps_1 \rightsquigarrow^e_{p_1} ps_2$ with program state ps_2 equal to:
$D, \quad \{p_1 = (\texttt{insert}\ j_1\ \texttt{into}\ j_3\ );$
$\qquad p_2 = (\texttt{insert}\ i_1\ \texttt{/C into /B})\ \}$

where j_3 is the result of $/B$ and j_1 is the result of evaluating $\$x/C$ with $\$x = i_1$.

The next transition to fire is $ps_2 \rightsquigarrow^e_{p_2} ps_3$ where ps_3 is:
$D, \quad \{p_1 = (\texttt{insert}\ j_1\ \texttt{into}\ j_3\ );$
$\qquad p_2 = (\texttt{insert}\ j_2\ \texttt{into}\ j_3)\ \}$

An *application step* simply consumes a concrete update u and replaces the document D by the result of applying u to D. If ps$'$ is the result of an application step applying p to program state ps under the standard API, we write ps $\rightsquigarrow^a_p$ ps$'$. Under the standard or strict API we also specify that if u aborts then the application step returns abort as well.

In the above example we have:
$ps_3 \rightsquigarrow^a_{p_1} D', \{(p_2 = (\texttt{insert}\ j_2\ \texttt{into}\ j_3)\}$
where D' is the result of applying the concrete update $(\texttt{insert}\ j_1\ \texttt{into}\ j_3)$ to D.

The processing of the update u would conclude with the application step:
$\rightsquigarrow^a_{p_2} D'', \emptyset$
where D'' is the result of applying the concrete update $(\texttt{insert}\ j_2\ \texttt{into}\ j_3)$ to D'.

An *evaluation sequence* is any sequence of steps $\rightsquigarrow^e$ and $\rightsquigarrow^a$ as above, leading from the initial document and update statement to some document with empty pending update sequence. The final document D'' is the *output of the sequence*.

In general, different evaluation sequences may produce distinct outputs. This is where we need the final component of the semantics: a *strategy* for deciding which steps to execute. As mentioned in the introduction, all existing proposals use a semantics which restricts to evaluation sequences such that:

(i) (snapshot rule) all available evaluation transitions $\rightsquigarrow^e$ must be applied before any application step $\rightsquigarrow^a$ is performed,

(ii) (generation order rule) the application steps $\rightsquigarrow_a$ must then be applied in exactly the order given in the pending update sequence - that is, we always perform $\rightsquigarrow^a_p$ starting at the initial concrete update in the sequence.

It is easy to see that this results in a unique output for each update. We say $(D, U) \rightsquigarrow D'$ if $(D, \{(\emptyset, U)\})$ rewrites to $(D', \emptyset)$ via a sequence of $\rightsquigarrow^e$ and $\rightsquigarrow^a$ transitions, using the standard API subject to the two conditions above. Under this semantics, the processing of program U1 is given in Figure 2.

3 Optimizations

For the rest of this paper we consider the semantics using the standard API and the snapshot and generation-order rules as the "official semantics". Naturally, an implementation of this semantics will differ from the conceptual version above: e.g. multiple evaluation or concrete update steps can be folded into

$D, \{p_0 = (\emptyset, U1)\}$
$\rightsquigarrow^e_{p_0}$

$D, \{p_1 = (\texttt{insert}\ i_1\texttt{/initial/text() into}\ i_1\texttt{/current});$
$\quad p_2 = (\texttt{delete}\ i_1\texttt{/bidder[@increase=3000]});$
$\quad p_3 = (\texttt{insert}\ i_2\texttt{/initial/text() into}\ i_2\texttt{/current});$
$\quad p_4 = (\texttt{delete}\ i_2\texttt{/bidder[@increase=3000]})\ \}$
$\rightsquigarrow^e_{p_1}$

$D, \{p_1 = (\texttt{insert}\ t_1\ \texttt{into}\ c_1);$
$\quad p_2 = (\texttt{delete}\ i_1\texttt{/bidder[@increase=3000]});$
$\quad p_3 = (\texttt{insert}\ i_2\texttt{/initial/text() into}\ i_2\texttt{/current});$
$\quad p_4 = (\texttt{delete}\ i_2\texttt{/bidder[@increase=3000]})\ \}$
$\rightsquigarrow^e_{p_2} D, \ldots \rightsquigarrow^e_{p_3} D, \ldots \rightsquigarrow^e_{p_4}$

$D, \{p_1 = (\texttt{insert}\ t_1\ \texttt{into}\ c_1);$
$\quad p_2 = (\texttt{delete}\ b_1);$
$\quad p_3 = (\texttt{insert}\ t_2\ \texttt{into}\ c_2);$
$\quad p_4 = (\texttt{delete}\ b_2)\ \}$
$\rightsquigarrow^a_{p_1}$

$D_1, \{p_2 = (\texttt{delete}\ b_1);$
$\quad p_3 = (\texttt{insert}\ t_2\ \texttt{into}\ c_2);$
$\quad p_4 = (\texttt{delete}\ b_2)\ \}$
$\rightsquigarrow^a_{p_2}$

$D_2, \{p_3 = (\texttt{insert}\ t_2\ \texttt{into}\ c_2);$
$\quad p_4 = (\texttt{delete}\ b_2)\ \}$
$\rightsquigarrow^a_{p_3} D_3, \ldots \rightsquigarrow^e_{p_4}$

$D_4, \emptyset$

Figure 2: The snapshot evaluation for the update U1; we abbreviate the nodeId corresponding to i_1`/initial/text()` (i_2`/initial/text()`) with t_1 (t_2) and similarly for c_1, c_2, b_1 and b_2.

one, and cursors over intermediate tables containing query results will be used, rather than explicit construction of the pending update sequence. However, even a sophisticated implementation may be inefficient if it respects (i) and (ii) above.

3.1 Re-ordering Optimizations

We first consider relaxing the generation order rule (ii) by indexing concrete updates in an order other than generation order, and then using this index to choose which application step $\rightsquigarrow^a_p$ to apply. In Example U1, we may wish to index deletes before inserts, and thus apply the delete operations $\rightsquigarrow^a_{p_2}$ and $\rightsquigarrow^a_{p_4}$ first. That is, we change rule (ii) so that we apply $\rightsquigarrow^a$ for the first concrete delete in the pending update sequence, and then apply other operations in generation order. We call this the *delete-first* rule. An "early delete" strategy can reduce space usage, in addition to decreasing the number of times indices must be re-numbered, for example when an indexing scheme based on document order is required. Note that under *every* possible semantics considered here, we cannot switch the ordering among concrete insert operations which have the same target, unless the two inserted trees are isomorphic.

To perform the "early delete" optimization, we have to know that it is sound. Whether or not this is always the case depends on the issue of strictness. It is easy to verify that if the lenient API is used, the order-based optimization above is always sound. If some operations abort, then the alternative orderings above may produce different results. Under the standard or strict API, different results can occur depending on the order, because a sequence of concrete updates produced contains two updates whose targets are in an ancestor/descendant relationship.

If we know at specification time that for program U, the delete-first evaluation can always be used, then we can simply index delete operations as they are generated separately from other operations and then evaluate using the delete-first strategy. We say that an update U is *Order Independent* (OI) if *delete-first* gives the same output as *generation-order* for any document. In [TR05], we show that for updates based on navigational *XPath* and navigational XQuery (we call these *navigational updates*), OI is decidable. The decision procedure makes use of a simple reduction to satisfiability of a small subset of XPath 2.0, along with a satisfiability test for this subset. This can be used as an approximate test for the full UpdateX language, by using an abstraction taking an arbitrary update to a navigational one.

3.2 Eager Optimization

We now turn to optimizations weakening the snapshot rule rather than the generation-order rule. The snapshot rule (i) forces the evaluation of all embedded expressions to occur prior to update application, which can be inefficient for a number of reasons. In particular, it may be more efficient to apply delete operations as soon as they are generated, since this will reduce space consumption and can dramatically reduce the processing time in further evaluations.

The *eager optimization* of an update u is the evaluation formed by dropping rule (i) and replacing it with the requirement (i') perform either an application step or an evaluation step on the first element of the pending update sequence. It is easy to see that i') also guarantees that there is at most one outcome of every evaluation. We denote the corresponding rewriting relation by $\leadsto^{eager}$.

The eager optimization is not necessarily sound, regardless of the API. A simple example of an update for which it is not valid is:

```
U3 : update
    for $i in //openauction,for $ j in //openauction,
        insert $i into $j
```

A simple argument shows that U3 behaves differently under eager optimization than under the snapshot semantics. Under eager optimization, the initial evaluation step will produce a sequence of identical copies of the inner loop. After this, the first copy of the loop will be evaluated and its generated updates applied, then the second copy, and so forth. One can see that this process can increase the size of the document exponentially. In contrast, under the standard semantics U3 can produce only a polynomial increase.

We turn to the problem of statically verifying that the eager optimization is sound. We say that an update u is *Binding Independent* (BI) if any evaluation sequence for u satisfying requirement (ii) produces the same output modulo an isomorphism that preserves any nodeIds from the original document D. Note that if T_2 and T_1 are strongly isomorphic, then no user query can distinguish between them.

Unfortunately, one cannot hope to decide whether an arbitrary update in UpdateX is BI, even for updates based on navigational XPath and XQuery. That is, the problem of deciding whether the eager optimization is sound is undecidable, even for navigational updates [CAV05]. Despite this, [CAV05] gives an algorithm giving a conservative test checking whether an update is Binding Independent, a test that can be relativized to check independence relative to documents that satisfy a given schema or DTD. We sketch the key idea of this algorithm below.

Intuitively, an update is BI if performing concrete updates that are generated from a program does not impact the evaluation of other XPath expressions. For the example U1, we can see that in order to verify BI it suffices to check that: i) for each a_0 in //openauction, the update ($i:a$_0$,insert $i/initial/text into $i/current) does not change the value of the expression $i/initial/text, $i/current, or $i/bidder, where in these expressions $i can be bound to any other a_1 in //openauction, ii) for each a$_0$ in //openauction, ($i:a$_0$, delete $i/bidder) does not effect $i/bidder, $i/initial/text, or $i/current for any i\neq$ a_0. The above suffices to show BI, since it implies that performing any evaluation step after an update gives the same result as performing the evaluation before the update.This observation is the basis for the key notion of a *non-interfering update program*: one for which any generated concrete update cannot effect the output of any query involved in evaluation of the updates. Any update program that is non-interfering is BI, and furthermore the number of transition steps used to evaluate such a program under the eager semantics will be no more than the number of transitions to evaluate it under the standard semantics.

[CAV05] gives an algorithm to test whether an update operation can interfere with an XPath query (where both the operation and the query can have parameters), and this in turn gives a conservative test for Binding Independence. As with the OI decision procedure, this algorithm relies upon a reduction to the satisfiability problem for XPath expressions. From a program we generate several *non-interference checks*, that assert that an update does not affect the value of an XPath expression; from each assertion we generate a set of XPath 2.0 expressions whose unsatisfiability will confirm the assertion. Finally, for each of these XPath expressions, we run a satisfiability test to determine that they are all unsatisfiable. This algorithm for non-interference checking may be of interest in its own right, for use static analyses related to XML concurrency control.

4 Static Analysis Implementation and Experiments

In order to investigate the feasibility of these optimizations, we have implemented the soundness test for eager optimization. In addition, we have implemented both the eager and order-based optimization on top of Galax as an adjunct external Java module. The result of the static analysis for eager optimization is a boolean flag passed together with the update program. If the flag states that the update is BI the eager evaluation is used.

The general picture of the static analysis and its components is shown in Figure 3. At verification time, a program is parsed and then goes through the conservative analysis for soundness, which includes the generation of non-interference tests, which in turn generate calls to our XPath satisfiability test. If all non-interference tests return positive, then the program is verified to be BI. If some test is negative, than nothing can be concluded about the program. At runtime, the program is processed by our modification of the Galax-based UpdateX engine.

The verification algorithms are implemented completely in Java, while the runtime, like the rest of Galax engine, is in OCAML. The runtime optimizations are currently imple-

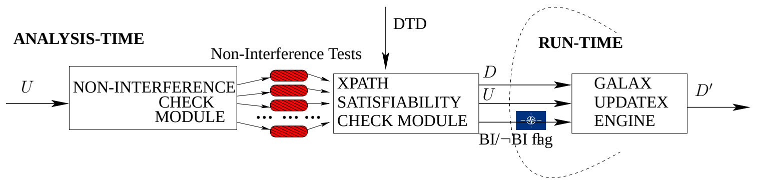

Figure 3: Architecture of Static Analysis-time and Run-time in Galax.

mented only on Galax 0.3.5, by modifying the module implementing FLWUpdates. It is more challenging to integrate eager optimization into Galax 0.4.0 and higher, where updates are fully integrated with querying. At the moment, our analysis is a "global one" that decides the choice of the entire evaluation strategy for the update; it is not clear how this can be used within a general algebraic optimization framework.

We ran experiments on a testbed of sample updates, built upon XMark [VLDB02] DTD, whose snippet is illustrated in Figure 4. To measure the static analysis time, we chose a collection of 16 sample updates, divided into *delete-intensive updates*, *insert-intensive updates*, to indicate whether the generated deletes will outnumber inserts or viceversa. These updates exhibit various numbers of nested for clauses, ranging *one to three*. The program text for each example appears in the Appendix.

U	*Update Descr.*	*Operations*	*AT*	*Res*
U_1	U with 1for	1del	700	BI
U_2	U with 2for	1ins,1del	600	BI
U_3	U with 1for	1ins,1del	610	BI
U_4	U with 2for	1ins,1del	450	BI
U_5	U with 2for	1del	420	BI
U_6	U with 3for	1ins,1rep	9030	BI
U_7	U with 2for	1ins	380	BI
U_8	U with 2for	1ins,1del	550	NotBI
U_9	U with 2for	1ins	360	BI
U_{10}	U with 2for	1ins	460	NotBI
U_{11}	U with 2for	1rep	780	NotBI
U_{12}	U with 2for	1ins,1del	470	NotBI
U_{13}	U with 1for	1ins,1del	420	BI
U_{14}	U with 1for	1ins,1rep	1950	BI
U_{15}	U with 1for	1ins	400	NotBI
U_{16}	U with 2for	1rep	1170	BI

Table 3: U (Update Nr.); AT (Analysis Time in ms); Res (Result).

In each case, it can be noted that the verification runs in at most seconds. At this time we can make no definitive statement on the runtime impact of the optimization. The eager optimization decreases update times by about 30% in execution time and 50% in memory usage in average for the queries in the delete-intensive category in the Table above, for the datasets ranging from 30MB to 100MB. Although this gives an indication that the optimization is sometimes useful, these updates reflect an extreme case where both time and space usage are affected most significantly by the use of eager evaluation. In addition, the underlying update API in the Galax versions we tested on was not tuned for performance, and hence we do not believe a convincing test of the effect of evaluation-strategy optimization can emerge from testing on top of Galax 0.3.5 or 0.4.0.

5 Conclusions and Related Work

One contribution of this paper is an exploration of semantic issues around XQuery updates. We hope that formalizing evaluation in terms of concrete updates, single-step operators, and ordering policies will make it easier to look at alternative semantics, and to study optimizations of the standard semantics. The optimizations we present in Section 3 are obviously only a few that can be considered for XQuery updates, but they give a feel for the difficulties that arise in defining sound optimizations. But because many applications define bulk updates well before their use, an optimization requiring computationally-intensive static analysis to check soundness may well be feasible. Certainly our current experimental results on the overhead of analysis time are encouraging. Of course, these analyses address the question of whether a particular optimization can be done soundly at all. A question for future work is what sort of cost model is needed to determine whether a sound optimization should be done. The key component of our analysis is an algorithm for determining statically whether an update can affect the value of an XPath expression. As future work, we plan to investigate other uses of this algorithm, for transaction management and access-control.

Related Work. A detailed description of the algorithm verifying soundness of the eager optimization was presented in [CAV05], in the context of a simple tree update language based on navigational XPath. Some of the experiments of Section 4 and the undecidability proof of Binding Independence are also from [CAV05]. Most of the XQuery-specific discussion, including semantics, strictness-issues in the API, and ordering optimization, is not contained in [CAV05].

The *snapshot* semantics for XML updates has been studied in [WS00, WS02, SIGMOD01, PlanX04] and earlier for semistructured-data in [ABS99, IJDL97]. [SIGMOD01] considers an implementation of the snapshot semantics which relies on mapping them to SQL updates. Throughout this paper, we consider a restriction of the syntax and the snapshot semantics of XML Updates as described in [PlanX04], since the language of [PlanX04] is both the one adopted in Galax [VLDB03, ECOOP03] and being discussed at W3C [W3C04]. Some preliminary hints on read-write conflicts and write-write conflicts are sketched in the master thesis by P.Lehti [WS02] and a discussion on update semantics is included in the W3C draft on XQuery Update Language [W3C04] and in a proprietary proposal [MS02]. A suitable extension of SQL with XQuery-like data manipulation primitives is implemented as part of the XML:DB Initiative Project [EEXTT02]. A suitable extension of SQL with XQuery-like data manipulation primitives is implemented as part of the XML:DB Initiative Project [EEXTT02]. Major RDBMS vendors [Oracle03, ICDE01, SQLServer05] and native XML vendors are providing their own ad-hoc solutions for XML updates. One interesting approach is that of MS SQL Server 2005 [ICDE01, SQLServer05], where *updategrams* are used to perform transformations from a before-state to an after-state.

Past work on relational updates and non-deterministic/deterministic transactional languages is very pertinent to our work [PODS88, TODS01, AI91, Infix98]. [TODS01] investigates the problem (shown to be decidable for a language fragment) of applying update methods to a set of objects in an arbitrary order. [AI91] considers the bounded-iteration construct **for each** x **in** R | p **do**, whose execution is the concatenation in arbitrary order of transaction constituted by update statements on relational tuples, and investigates conditions for which a procedural implementation reduces to a declarative one. Work on static analysis of rule conflict in active databases (e.g. [TODS00]) does not directly address optimization of high-level updates, but has similar aims to the non-interference algorithm used in our soundness test. Given the significant differences between XML and relational updates, these results cannot be applied directly to the XQuery setting. In addition, the existence of decision procedures for satisfiability of navigational XPath on ordered trees makes static analysis of navigational updates much more feasible than their relational counterparts [PODS05].

There has been considerable work on static analysis of other XML query and transformation languages. A good summary, focusing on the type-checking problem, can be found in [ICDT05]. [VLDB2003] focuses on static analysis for optimization, giving a compile time DTD-based analysis of XQuery and an optimization based on the analysis output. Because XQuery cannot perform destructive updates, this kind of approximate analysis is much different than ours. Other analyses work at the level of query plans or transaction plans. An example that has a similar flavor to our work on ordering optimization is [DBPL03], which analyzes query plans to remove unneeded ordering operations. Our notion of non-interfering updates has some similarity to work on analysis of concurrency control for XML [ISADS03]. This work (and its predecessor papers) analyze at the level of transaction schedules, not the update language.

Acknowledgements: We thank Jérôme Siméon for many helpful discussions on update semantics, and Andrea Pugliese for his support in implementation of the static analysis routines. We also thank Wenfei Fan for valuable comments on the draft.

References

[ABS99] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann Publishers, San Francisco, California, 1999.

[AI91] X. Qian. The expressive power of the bounded-iteration construct. *Acta Informatica*, 28(7):631–656, 1991.

[CAV05] M. Benedikt, A. Bonifati, S. Flesca, and A. Vyas. Verification of Tree Updates for Optimization. In *(To appear) Proc. of the Int. Conf. on Computer Aided Verification (CAV)*, 2005. Technical Report:http://www.icar.cnr.it/angela/updates.pdf.

[CCS03] M. Murata, A. Tozawa, M. Kudo, and S. Hada. XML access control using static analysis. In *Proc. of CCS*, 2003.

[Dom98] V. Apparao et al. *Document Object Model (DOM) Level 1 Specification*. W3C Recommendation, Oct. 1998. http://www.w3.org/TR/REC-DOM-Level-1.

[DBPL03] J. Hidders and P. Michiels Avoiding Unnecessary Ordering Operations in XPath In *DBPL*, 2003.

[ECOOP03] M. Fernandez and J. Sim´eon. Growing XQuery. In *Proceedings of ECOOP*, pages 405–430, 2003.

[EEXTT02] D. Obasanjo and S. B. Navathe. A Proposal for an XML Data Definition and Manipulation Language. In *Proc. of EEXTT*, 2002.

[ICDE01] M. Rys. Bringing the Internet to Your Database: Using SQLServer 2000 and XML to Build Loosely-Coupled Systems. In *Proc. of ICDE*, pages 465–472, 2001.

[ICDT05] M. Schwartzbach and A. Moller. The design space of type checkers for XML transformation languages. In *Proc. of ICDT*, 2005.

[IJDL97] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The lorel query language for semistructured data. *Int. J. on Digital Libraries*, 1(1):68–88, 1997.

[ISADS03] S. Dekeyser, J. Hidders and J. Paredaens. Instance-independent Concurrency Control for Semi-structured Data. In *Italian Symposium on Advanced Database Systems*, 2003.

[Infix98] M. Rys. *Materialisation and Parallelism in the Mapping of an Object Model to a Relational Multi-processor System*. Infix Verlag, St. Augustin, Germany, 1998.

[MS02] M. Rys. *Proposal for an XML Data Modification Language*. Microsoft Corp., Redmond (WA), Proposal May 2002.

[Oracle03] Oracle9i. Web Site, 2003. http://www.oracle.com.

[PlanX04] G. Sur, J. Hammer, and J. Sim´eon. An XQuery-Based Language for Processing Updates in XML. In *PLAN-X*, 2004. See also: http://www.cise.ufl.edu/research/mobility.

[PODS88] S. Abiteboul and V. Vianu. Procedural and declarative database update languages. In *Proc. of ACM PODS*, 1988.

[PODS05] M. Benedikt and W. Fan and F. Geerts. XPath Satisfiability in the Presence of DTDs. To appear in *Proc of ACM PODS*, 2005.

[SIGMOD01] I. Tatarinov, Z. Ives, A. Halevy, and D. Weld. Updating XML. In *Proc. of ACM SIGMOD*, 2001.

[SQLServer05] XML Support in Microsoft SQL Server 2005. http://msdn.microsoft.com/xml/default.aspx?pull=/library/en-us/dnsql90/html/sql2k5xml.asp.

[TODS00] E. Baralis and J. Widom An algebraic approach to static analysis of active database rules. *ACM TODS*, 25(3):269–332,2000.

[TODS01] M. Andries, L. Cabibbo, J. Paredaens, and J. V. den Bussche. Applying an update method to a set of receivers. *ACM TODS*, 26(1):1–40, 2001.

[TR05] M. Benedikt, A. Bonifati, S. Flesca, and A. Vyas. Semantics and Optimization of XML Updates Technical Report 2005.

[TSE04] C. Kirkegaard, A. Moller, and M. I. Schwartzbach. Static analysis of xml transformations in java. *IEEE Transactions on Software Engineering*, 2004.

[VLDB02] A. Schmidt, F. Waas, M. Kersten, M. Carey, I. Manolescu, and R. Busse. XMark: A benchmark for XML data management. In *Proc. of VLDB*, 2002.

[VLDB03] M. Fernandez, J. Sim´eon, B. Choi, A. Marian, and G. Sur. Implementing XQuery 1.0: The Galax Experience. In *Proc. of VLDB*, 2003.

[VLDB2003] A. Marian and J. Sim´eon. Projecting XML Documents. In *Proc. of VLDB*, 2003.

[W3C04] Updates in XQuery: working notes of W3C XQueryX , 2004.

[WS00] A. Laux and L. Martin. XUpdate - XML Update Language., 2000. http://www.xmldb.org/xupdate/xupdate-wd.html.

[WS02] P. Lehti. Design and Implementation of a Data Manipulation Processor. Diplomarbeit, Technische Universitat Darmstadt, 2002. `http://www.lehti.de/beruf/diplomarbeit.pdf`.

[WS04] M. Olesen. Static validation of XSLT, 2004. `http://www.daimi.au.dk/ madman/xsltvalidation`.

[XQuery04] Website. XQuery 1.0: An XML Query Language, 2004. `http://www.w3.org/TR/xquery`.

6 Appendix

The DTD for our sample is shown in Figure 4.

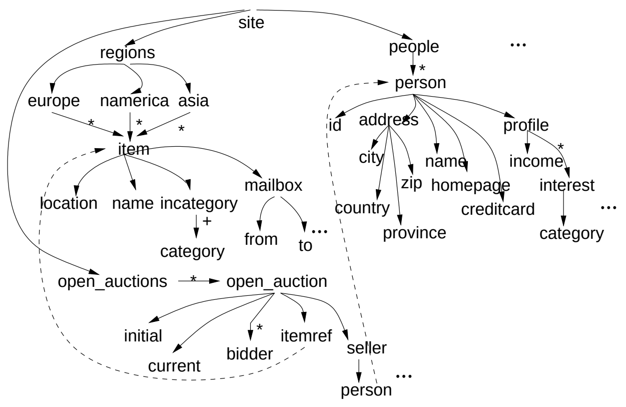

Figure 4: XMark auction DTD

6.1 Experimental Data

The text of update statements used in experiments is shown in the following:

```
Q1 : update
    for $person in //site/people/person
        delete $person
Q2 : update
    for $person in //site/people/person,
      $watches in $person/watches
        insert ⟨watch/⟩ into $ watches
        delete $person/emailaddress
Q3 : update
    for $person in //site/people/person,
        insert ⟨homepage/⟩ into $person
        delete $person/profile
Q4 : update
    for $people in //site/people,
      $person in $people/person
        insert $person into $people
        delete $person
Q5 : update
    for $o_auction in //site/open_auctions/open_auction,
      $c_auction in //site/closed_auctions/closed_auction
        delete $c_auction
```

$Q_6, \ldots, Q_{16}$ can be found at the following url: *http://www.icar.cnr.it/angela/examples.html*