Schema-Based Independence Analysis for XML Updates

Michael Benedikt
University of Oxford
michael.benedikt@comlab.ox.ac.uk

James Cheney
University of Edinburgh
jcheney@inf.ed.ac.uk

ABSTRACT

Query-update independence analysis is the problem of determining whether an update affects the results of a query. Query-update independence is useful for avoiding recomputation of materialized views and may have applications to access control and concurrency control. This paper develops static analysis techniques for query-update independence problems involving core XQuery queries and updates with a snapshot semantics (based on the W3C XQuery Update Facility proposal). Our approach takes advantage of schema information, in contrast to previous work on this problem. We formalize our approach, sketch a proof of correctness, and report on the performance and accuracy of our implementation.

1. INTRODUCTION

In recent years query and transformation languages for XML data have been studied extensively. The World Wide Web Consortium (W3C) has developed XQuery, a standard XML query language with a detailed formal semantics and type system [9, 14]. Most real-world data changes over time, and so it is also important to be able to update XML documents and XML-based data. However, query languages such as XQuery (and transformation languages such as XSLT) are awkward for writing transformations that update part of the data "in-place" while leaving most of the document alone.

There have been a number of proposals and prototype implementations for XML update languages (see for example [1, 11, 16, 26]). While no clear winner has emerged so far, the W3C has introduced the XQuery Update Facility [10], combining features from several proposals. This is now supported by many XML database implementations and appears well on its way to becoming standard. However, reasoning about updates is challenging; many basic problems, such as the typechecking and static analysis problems for XQuery Update (and for XML updates more generally) remain ill-understood.

One fundamental static analysis problem is that of deciding *query-update independence*, or whether an update *conflicts* with a query [22]. Independence analysis has numerous applications, such as detecting when an integrity constraint needs to be re-validated or a view re-computed after an update occurs. Query-update independence is also related to problems such as access control and concurrency control for XML queries and updates. For example, an access control policy might specify that the result of a particular view must not be altered. Query-update independence implies that a given update satisfies this policy. We will not pursue these further applications in this paper.

Obviously, we can determine at runtime whether an update impacts a query: we simply run the update, then re-run the query, and finally compare the results. However, in practice this *dynamic independence testing* is expensive, especially as the number of constraints or views grows, and it does not save us any work if our ultimate goal is to avoid recomputation. We thus want to compare an approach based on static analysis of independence against an approach based on re-evaluation.

Unfortunately, as we shall show, static query-update independence testing is undecidable in general (interreducible to query equivalence) and Raghavachari and Shmueli [22] showed that it is NP-hard even for XPath-based queries and updates. Therefore, in this paper we study static analyses that *conservatively approximate* the true results. Conservative independence analysis either determines independence or says "unknown".

We distinguish two application scenarios. In the first, we know the typical updates and queries well in advance of their evaluation. In this case, it would suffice to have an *offline analysis* that detects independence; such an analysis might be fairly expensive – for example, taking minutes or hours. If we are only concerned with what happens for a fixed (or rarely changing) set of queries and updates, then we can afford to perform sophisticated and time-consuming analyses, perhaps even ones that provide exact answers (when this is decidable). Previous work on static analysis and optimization of XML updates has focused on such offline scenarios [2].

In the second, *online* scenario, we are given (perhaps a large number of) queries expressing constraints or views, but we do not know the updates in advance. In this case, for the analysis to be useful, it must take (much) less time than full re-evaluation; if the analysis takes a long time but ultimately decides that the query will need to be re-evaluated anyway, then this could impose an unacceptable delay. In this paper, we develop an analysis that is both accurate and fast enough to be useful even for view maintenance settings involving relatively small documents (e.g. around 1MB in size). Of course, this can also be used for offline analysis.

Previously, Ghelli et al. [16] studied update commutativity. Our work differs from theirs in two important respects. First, our language is based on the emerging XQuery Update standard, whereas theirs is based on an different update language with somewhat more complicated semantics. Furthermore, the update commutativity problem resembles, but is not the same as, the query-update in-

VLDB '09, August 24-28, 2009, Lyon, France

dependence problem we study. Second, our approach is based on leveraging *schema information* and Ghelli et al.'s work is based on analysis of paths read or written by the query and update, making no assumptions on the input. Thus, our approach can take advantage of knowledge of the structure of the input.

Of these differences, the second is the more significant, since it does not appear hard to adapt Ghelli et al.'s path-based analysis to handle a different update semantics; in fact, XQuery Update 1.0 is in many ways easier to analyze than their language. As we will show, neither path- nor schema-based analysis is strictly more precise than the other. It seems worthwhile to combine the schema-based and path-based approaches, but in this paper we focus only on the novel schema-based approach.

We illustrate the difference between path and schema-based analysis via the following examples. The examples refer to a common schema S defined as follows:

```
S -> document [A*,B]
A -> a[(B?,C)*]
B -> b[]
C -> c[D]
D -> d[]
```

The schema above is a representation of an XML Schema (in fact, a DTD) in which there are types *S*, *A*, *B*, *C*, and *D*, while the production rules specify the tags and child content of each type. For example, the first rule says that a node of type *S* is associated with tag `document`, and the types of its children must match the regular expression `A* B`. Below we will assume a context in which variable $*doc* points to a node of type *S* in the schema above.

EXAMPLE 1. *Consider the XPath query Q_0 that returns all children of the variable which are labeled* `b`*:*

```
$doc/b
```

and the update U_0 that deletes all d *nodes with parent* c*, grandparent* a*, and great-grandparent the root:*

```
delete $doc/a/c/d
```

Clearly, U_0 cannot "impact" Q_0, so we do not need to re-compute Q_0 when U_0 is applied. This is true for any input document, and both path-based analysis and schema-based analysis can determine this.

EXAMPLE 2. *Consider the same query Q_0 as in the previous example, and the update U_1 that deletes all* `d` *nodes lying below the variable:*

```
delete $doc//d
```

In this case, path-based analysis cannot ensure that Q_0 and U_1 are independent, since the query and update are not independent on an arbitrary document. But note that the nodes returned by the query must have type B*, while the nodes deleted by U_1 must be of type* D*, and no* B *nodes lie beneath* D *nodes. Thus we can see that Q_0 and U_1 are independent on all documents matching the schema.*

EXAMPLE 3. *Consider the XQuery query Q_2:*

```
for $x in $doc/a/b
return <c>$x</c>
```

and the update U_2 that deletes all `b` *immediately below the variable:*

```
delete $doc/b
```

In this case, path-based analysis will easily determine that Q_2 and U_2 are independent: it will determine that the update will delete nodes having path $\texttt{document}/b$*, while the query is concerned with paths of the form* $\texttt{document}/a/b$*. But our schema-based analysis will not detect independence (at least, not with respect to this schema). The reason is that our approach uses the same type name* B *to refer to both the nodes read by Q_2 and those deleted by U_2, and does not employ any path or context information.*

EXAMPLE 4. *As a final example, we consider a query and update whose independence neither path-based nor schema-based analysis can verify. Consider Q_3 that returns all of the nodes matching $\$doc/a/b$ except those under the first* a*:*

```
for $x in $doc/a[position() <> first()]/b
return <c>$x</c>
```

and the update U_3 that deletes the b *nodes under the first* a*:*

```
for $x in $doc/a[position()=first()]/b
return delete nodes $x
```

Clearly, Q_3 and U_3 are independent. However, a path-based analysis like that of Ghelli et al. [16] cannot detect this because it does not take position information into account. Since Q_3 reads from some nodes matching $\$doc/a/b$ path and U_3 impacts some nodes matching $\$doc/a/b$, we must conservatively conclude that they may interfere. Similarly, our schema-based analysis cannot prove that these queries are independent either, since it will statically observe that Q_3 may access nodes of type B *whereas U_3 may impact nodes of type* B*.*

This last example illustrates the inevitable trade-off between the complexity and completeness of a static analysis. We know we cannot have both, so it is of interest to find efficient techniques that are incomplete but nevertheless practically useful.

In this paper, we study schema-based independence analysis for XQuery Updates. Our approach will employ many of the ideas used in the previous literature on XML query and update analysis (see Section 5 for a comparison). In particular, we adapt the notion of the "accessed nodes" of a query used in works such as [18, 6, 16]. Our version must take into account a schema, and is tailored to the update operations available in the XQuery Update Facility – prior notions have been either in the context of queries rather than updates, or in a schema-less setting. While this combination of features adds complexity to our problem, the "snapshot semantics" of the XQuery Update Facility simplifies things considerably compared to several other prior language proposals (e.g. [16]). We feel that this simplicity allows us to isolate some of the key intuitions behind the notion of accessed nodes which are more difficult to extract in the context of a complex language.

The main contributions are thus:

- We give a sound analysis that will detect when an XQuery query and XQuery Update Facility update are independent. Our analysis employs a powerful abstraction of XML schemas, with the expressiveness of arbitrary tree automata, and handles all XPath axes.
- We provide experimental evidence of the efficacy of our analysis, both in terms of performance and accuracy.

For ease of exposition, we consider independence analysis for a limited "core" XQuery language that nevertheless suffices for most of the XMark and XPathMark benchmark queries. We also leave out XQuery Update's "transform" query expression and "replace value of" update operation [10]. We omit proofs and standard definitions; these are placed in the companion technical report [3].

Judgment	Meaning	See
$\sigma \models_{\mathsf{S}} \mathtt{l} : \mathtt{T}$ $\sigma \models_{\mathsf{S}} \gamma : \Gamma$	Validation	Section 2, [3]
$\sigma, \gamma \models \mathtt{q} \Rightarrow \sigma', \mathtt{L}$ $\sigma \models \omega \rightsquigarrow \sigma'$ $\sigma, \gamma \models \mathtt{u} \Rightarrow \sigma', \omega$ $\sigma, \gamma \models \mathtt{u} \rightsquigarrow \sigma'$	Evaluation	Section 2, [3]
$\mathsf{S} \vdash \mathcal{A}/ax{::}\phi \stackrel{\mathsf{step}}{\Rightarrow} \mathcal{A}'$	XPath step typing	Section 3, [3]
$\mathsf{S}; \Gamma \vdash \mathtt{q} : \mathcal{A}$	Type inference	Section 3, Figure 1
$\mathsf{S}; \Gamma \vdash \mathtt{u}$ impacts $\mathcal{A}$	Impacted nodes	Section 3, Figure 2
$\mathsf{S}; \Gamma \vdash_{\mathsf{SAC}} \mathtt{q} : \mathcal{A}$	Static access cover	Section 3, Figure 3
$\mathsf{S} \vdash \mathtt{T} \sqcap \mathtt{T}'$	Aliasing	Section 3

Table 1: Judgments used in the paper

Outline. The rest of this paper is structured as follows: Section 2 reviews core query, update, and schema languages we will use. Section 3 presents the main components of our analysis. Section 4 discusses our implementation and gives experimental results. Section 5 discusses related and future work and Section 6 concludes.

2. BACKGROUND

In this paper we employ a number of different relations defining the semantics and static analysis of XQuery and XQuery Updates. These notations are summarized in Table 1 for easy reference, along with pointers to the parts of the paper or companion technical report where they are discussed or defined.

Stores and Dynamic Environments. Following [12] we employ a simplified data model and query language where we do not consider node attributes. An instance σ of the data model (or simply, a *store*) is an ordered labeled forest, whose nodes $\mathtt{l}, \mathtt{l}', \mathtt{m}$ (also referred to as a *locations*) are either element nodes or text nodes. An element node has a label, while a text node has an associated string. In addition to its label or string, each node has an identifier, which is assumed to be unique within a store.

A *(dynamic) variable environment* is a mapping γ taking a finite set of expression variables to sequences of locations within a store. We often write location sequences as $\mathtt{L}, \mathtt{L}', \mathtt{L}''$.

Schemas. In this paper we employ an abstraction of XML Schema that generalizes DTDs, corresponding in expressiveness to specifications in Relax NG [20]. Our schema formalism consists of an alphabet Σ of element tags, a collection T of *type names* (or *types*), a function mapping type names to elements, and a set of *rules* that associate to each type name a regular expression over type names. There is also a special type `text` which can appear in regular expressions, but has no associated rule. A schema may also optionally have a subcollection of types that are designated as *root types*. In [21] these are called *specialized DTDs*. They can also be considered a normal form for *regular expression types* [17]. We will use capital letters for types and lower-case letters for tags, while using regular expression type syntax, which combines the type name with the regular expression. In the example from the introduction, our type names include *A*, *B*, *C*, etc. while our tags will include a, b, and c. The rule `A -> a[(B?,C)*]` states that type name *A* is associated with tag `a`, and with the regular expression (*B*,*C*)* . A DTD is a special case of our formalism where type names are the same as tags.

A valid typing for S on a store σ is an assignment λ of nodes to types such that a) every text node gets mapped to the special type `text`, while every root node is mapped to a root type, and b) if a node is assigned type `T` by λ and `T -> a[e]` is a rule of the schema, then the label of the node must equal `a`, and there must be a sequence of types matching the regular expression e such that the i^{th} child of `l` is assigned by λ to the i^{th} type in the sequence. The notion of a node `l` in a document *satisfying* or *matching* a type `T` in a schema S (written $\sigma \models_{\mathsf{S}} \mathtt{l} : \mathtt{T}$) is that there is a typing λ that assigns `l` to `T`.

In this paper we will use a simplification of the standard XQuery type system that ignores node order within sequences returned by queries. A *static environment* is a mapping from expression variables to sets of types in a schema S. A variable environment γ for store σ is *consistent* with a static environment Γ for schema S (written $\sigma \models_{\mathsf{S}} \gamma : \Gamma$) if for every variable $x \in dom(\gamma)$, all nodes in $\gamma(x)$ match some type in $\Gamma(x)$.

Queries. We will use a simple core language for XQuery expressions:

$$
\begin{array}{rcl}
\mathtt{q} & ::= & x \mid () \mid \mathtt{q}, \mathtt{q}' \mid \langle a \rangle \mathtt{q} \langle /a \rangle \mid \mathtt{s} \mid x/step \\
& \mid & \mathtt{let}\ x := \mathtt{q}\ \mathtt{in}\ \mathtt{q}' \mid \mathtt{if}\ \mathtt{q}\ \mathtt{then}\ \mathtt{q}_1\ \mathtt{else}\ \mathtt{q}_2 \\
& \mid & \mathtt{for}\ x \in \mathtt{q}\ \mathtt{return}\ \mathtt{q}' \\
step & ::= & ax :: \phi \mid text() \\
ax & ::= & \mathtt{self} \mid \mathtt{child} \mid \mathtt{descendant} \\
& \mid & \mathtt{desc-or-self} \mid \mathtt{follsib} \mid \mathtt{precsib} \\
& \mid & \mathtt{parent} \mid \mathtt{ancestor} \mid \mathtt{anc-or-self}
\end{array}
$$

The $()$, $\langle a \rangle \mathtt{q} \langle /a \rangle$ and $\mathtt{q}, \mathtt{q}'$ and `s` expressions build XML values. The constant string expression `s` builds the fixed text node given by the string `s`. Variables and let-bindings are standard; conditionals branch depending on whether their first argument is nonempty. The expression $x/text()$ retrieves any text node lying below x, while $x/ax{::}\phi$ performs an XPath step starting from x, where ax is one of the standard XPath axes and ϕ is an XPath node test (either $*$ or an element tag a). In this paper we consider only a representative selection of the axes; it is straightforward to extend our results to other axes. The iteration expression $\mathtt{for}\ x \in \mathtt{q}\ \mathtt{return}\ \mathtt{q}'$ evaluates $\mathtt{q}$, and for each node l in the result evaluates $\mathtt{q}'$ with x bound to l, concatenating the results in order. Other axes, such as `following`, can be built up from these using composition.

We model the operational semantics of queries using a judgement $\sigma, \gamma \models \mathtt{q} \Rightarrow \sigma', \mathtt{L}$. Note that the input store σ may grow during evaluation of a query, for example in evaluating expressions of the form $\langle a \rangle \mathtt{q} \langle /a \rangle$ that require new nodes to be allocated; however, the values of nodes in σ are always preserved in σ'. The rules defining this (standard) semantics are given in [3].

A *selection query* is one that does not use the element node construction operation $\langle a \rangle \mathtt{q} \langle /a \rangle$ or string formation `s`. This restriction implies that selection queries always return nodes already present in the input and do not construct new nodes.

Atomic updates. We consider atomic updates of the form:

$$
\begin{array}{rcl}
\iota & ::= & \mathtt{ins}(\mathtt{L}, \mathtt{d}, l) \mid \mathtt{del}(l) \mid \mathtt{repl}(l, \mathtt{L}) \mid \mathtt{ren}(l, a) \\
\mathtt{d} & ::= & \leftarrow \mid \rightarrow \mid \downarrow \mid \swarrow \mid \searrow
\end{array}
$$

Here, the direction `d` indicates whether to insert before ($\leftarrow$), after ($\rightarrow$), or into the child list in first ($\swarrow$), last ($\searrow$) or arbitrary position ($\downarrow$). Moreover, we consider sequences of atomic updates ω with the empty sequence written ϵ and concatenation written $\omega; \omega'$. In [3] we define the semantics of atomic updates as a relation $\sigma \models \iota \rightsquigarrow \sigma'$.

Updating expressions. We now define the syntax of *updating expressions*, based roughly on those of the W3C XQuery Update

proposal.

$$
\begin{array}{rcl}
\mathtt{u} & ::= & () \mid \mathtt{u}, \mathtt{u}' \mid \mathtt{let}\ x := \mathtt{q}\ \mathtt{in}\ \mathtt{u} \\
& \mid & \mathtt{if}\ \mathtt{q}\ \mathtt{then}\ \mathtt{u}_1\ \mathtt{else}\ \mathtt{u}_2 \mid \mathtt{for}\ x \in \mathtt{q}\ \mathtt{return}\ \mathtt{u} \\
& \mid & \mathtt{insert}\ \mathtt{q}\ \mathtt{d}\ \mathtt{q}_0 \mid \mathtt{replace}\ \mathtt{q}_0\ \mathtt{with}\ \mathtt{q} \\
& \mid & \mathtt{rename}\ \mathtt{q}_0\ \mathtt{as}\ a \mid \mathtt{delete}\ \mathtt{q}_0
\end{array}
$$

The XQuery Update proposal re-uses existing query syntax for updates. The () expression is a "no-op" update, u, u' is sequential composition, and let-bindings, conditionals, and for-loops are also included. There are four atomic update expressions: *insertion* $\mathtt{insert}\ \mathtt{q}\ \mathtt{d}\ \mathtt{q}_0$, which says to insert a copy of $\mathtt{q}$ in position $\mathtt{d}$ relative to the value of $\mathtt{q}_0$; *deletion* $\mathtt{delete}\ \mathtt{q}_0$, which says to delete the value of $\mathtt{q}_0$; *renaming* $\mathtt{rename}\ \mathtt{q}_0\ \mathtt{as}\ a$, which says to rename the value of $\mathtt{q}_0$ to a; and finally *replacement* $\mathtt{replace}\ \mathtt{q}_0\ \mathtt{with}\ \mathtt{q}$, which says to replace the value of $\mathtt{q}_0$ with a copy of $\mathtt{q}$. In each case, the target expression $\mathtt{q}_0$ is expected to evaluate to a single node; if not, evaluation fails at run time (as specified by the W3C draft [10]).

Updates have a multi-phase semantics. First, the updating expression is *evaluated*, resulting in a *pending update list* ω. We model this phase using an update evaluation judgement $\sigma, \gamma \models \mathtt{u} \Rightarrow \sigma', \omega$. The rules for these judgements are presented in [3]. Recall that the store may grow as a result of allocation; however, but the values of existing locations in σ do not change in this phase. Next, ω is *sanity-checked* to ensure, for example, that all update targets are mutable nodes in σ and no node is the target of multiple rename or replace instructions. We do not model the sanity check step explicitly here. Finally, the pending updates are *applied* to the store. The formal semantics for update application is given in [4].

One natural-seeming semantics for update application is simply to apply the updates in ω in (left-to-right) order. However, this naive semantics is not what the W3C proposal actually specifies. In the W3C proposal [10], updates are *reordered*; inserts and renames are performed first, followed by replacements, and finally by deletions. For the purposes of this paper, we will conservatively assume that atomic updates may be performed in *any* order. A static analysis that is sound with respect to this semantics will also be sound with respect to any more restrictive semantics, including the W3C proposal.

We use the notation $\sigma, \gamma \models \mathtt{u} \rightsquigarrow \sigma'$, for the judgement that holds iff update expression u generates a pending update list on store σ in context γ such that the resulting pending update list is valid, and when applied (in some order) yields store σ'.

The sequential composition of an update $\mathtt{u}$ and a query $\mathtt{q}$ is written $\mathtt{u}; \mathtt{q}$, and $\sigma, \gamma \models \mathtt{u}; \mathtt{q} \Rightarrow \sigma_3, L$ is an abbreviation for $\exists \sigma_2 . \sigma, \gamma \models \mathtt{u} \rightsquigarrow \sigma_2 \wedge \sigma_2, \gamma \models \mathtt{q} \Rightarrow \sigma_3, L$.

We emphasize here that updates need not preserve the original schema of a document. It is nontrivial to calculate the effect of an update on a schema; we have also investigated this problem in a separate paper [4].

2.1 Equivalence and Independence

A query $\mathtt{q}$ is independent of an update $\mathtt{u}$ if, intuitively, the result of applying $\mathtt{q}$ after $\mathtt{u}$ is "the same" as the result of performing $\mathtt{q}$. But what does it mean for the results to be the same? Clearly, the nodes resulting from performing $\mathtt{q}$ after $\mathtt{u}$ should be allowed to differ from those resulting from performing $\mathtt{q}$ on the original store in inessential ways: for example, they can disagree on node identifiers. We capture the precise notion of equivalence in the following definitions.

DEFINITION 1 (VALUE EQUIVALENCE). *Given stores σ, σ_2 and sequences $\mathtt{L} \subseteq \mathrm{dom}(\sigma), \mathtt{L}' \subseteq \mathrm{dom}(\sigma_2)$, we say $\sigma, \mathtt{L}$ and $\sigma_2, \mathtt{L}'$ are* value equivalent *$(\sigma, \mathtt{L} \cong_V \sigma_2, \mathtt{L}')$ provided $\mathtt{L} = \mathtt{l}_1, \ldots, \mathtt{l}_n$ and $\mathtt{L}' = \mathtt{l}'_1, \ldots, \mathtt{l}'_n$ and for each $i \in \{1, \ldots, n\}$, the subtree with root $\mathtt{l}_i$ in σ is isomorphic to the subtree with root $\mathtt{l}'_i$ in σ_2.*

Value equivalence captures the idea that two programs return the same XML document given the same input, even using different node identifiers. For example, if q is a query that generates a XML tree that is sent to an external application, then we only care about this value, not the identities of the nodes generated by q.

DEFINITION 2 (QUERY-UPDATE INDEPENDENCE). *Given store σ and environment γ, we say that query $\mathtt{q}$ and update $\mathtt{u}$ are independent on (σ, γ) if:*

whenever $\sigma, \gamma \models \mathtt{q} \rightsquigarrow \mathtt{L}_1, \sigma_1$ holds, there exist $\sigma_2, \mathtt{L}_2$ satisfying $\sigma_1, \mathtt{L}_1 \cong_V \sigma_2, \mathtt{L}_2$ and $\sigma, \gamma \models (\mathtt{u}; \mathtt{q}) \rightsquigarrow \mathtt{L}_2, \sigma_2$ holds, and vice versa.

Given a query q, and a update u, we say that they are independent if the above holds for every σ and γ.

EXAMPLE 5. *We consider some examples, written using a variant of the XML Update Facility syntax. The query:*

$$\mathtt{for}\ \$x\ \mathtt{in}\ \$y/foo\ \mathtt{return}\ \langle a \rangle \$x \langle /a \rangle$$

is independent of update

$$\mathtt{for}\ \$x\ \mathtt{in}\ \$y/bar\ \mathtt{return}\ \mathtt{delete}\ \$x$$

because nothing the update does has an observable effect on the result of the query. Any changes made by the update are only to parts of the document the query does not access, so the result of the query will not change.

A more involved example: query

$$\mathtt{for}\ \$x\ \mathtt{in}\ \$y/foo\ \mathtt{return}\ \langle a \rangle \$x \langle /a \rangle$$

is independent of update

$$\mathtt{for}\ \$x\ \mathtt{in}\ \$y/foo\ \mathtt{return}\ \mathtt{insert}\ bar[42]\ \mathtt{after}\ \$x$$

because again the inserted nodes are not visible to the query.

On the other hand, query

$$\mathtt{for}\ \$x\ \mathtt{in}\ \$y/foo\ \mathtt{return}\ \langle a \rangle \$x \langle /a \rangle$$

is not independent of update

$$\mathtt{for}\ \$x\ \mathtt{in}\ \$y//bar\ \mathtt{return}\ \mathtt{insert}\ foo[42]\ \mathtt{after}\ \$x$$

because for some inputs the insertion will lead to additional nodes being visible in the result of the query.

An update is independent of a query relative to a schema if, roughly, the results of evaluating $\mathtt{u}; \mathtt{q}$ and $\mathtt{q}$ are value-equivalent for all stores satisfying the schema. To make this precise, we use a schema and static environments to constrain the stores we consider:

DEFINITION 3 (INDEPENDENCE RELATIVE TO A SCHEMA). *A query q, and update u, are independent relative to S, Γ if Definition 2 holds for every σ and every environment γ consistent with Γ.*

EXAMPLE 6. *Recall query*

$$\mathtt{for}\ \$x\ \mathtt{in}\ \$y/foo\ \mathtt{return}\ \langle a \rangle \$x \langle /a \rangle$$

and update

$$\mathtt{for}\ \$x\ \mathtt{in}\ \$y//bar\ \mathtt{return}\ \mathtt{insert}\ foo[42]\ \mathtt{after}\ \$x$$

Although this query–update pair is not independent in general, a schema might tell us that there are actually no immediate children of $\$y$ with element label bar, and this query-update pair is independent with respect to such a schema.

The query-update independence problem is undecidable for any realistic query and update language. For example, for full XQuery queries and XML Update facility updates, the problem is undecidable even when the query or update is fixed: this follows easily from a reduction to the satisfiability problem for first order logic over data trees.

For restricted cases independence is decidable, but at an enormous cost:

THEOREM 1. *For boolean XQuery queries and updates as given by the grammar in Section 2, the Query-Update independence problem is decidable, as well as the schema-based Query-Update independence problem. However, even for a fixed update and schema the problem is non-elementary.*

Both of these problems are proven by using a connection between the independence problem and the equivalence problem for XUpdate, which can in turn be analyzed using a combination of expressiveness results in [5], which relate XML query languages to logics, and known results about the complexity of equivalence of logical formulas on trees. Details are in the technical report [3].

3. STATIC ANALYSIS

3.1 Static analysis based on schemas

The intuition behind our independence analysis is similar to that of [16]: we want to show that for any input document and variable environment, the nodes affected by an update expression are disjoint from those returned or accessed by a query. There is already a standard notion of static typing that can be used to approximate the nodes returned by a query, and we first review a simplified static type system for XQuery. We will then define the runtime notions of read and updated nodes, and show how to statically approximate these as well.

In our analysis, we abstract input documents by schemas, sets of nodes by sets of type names, and dynamic environments by static environments.

Static type analysis. For queries we define a typechecking judgement that calculates the possible return types for nodes returned by the query when run in static environment Γ. In our analysis of queries we do not analyze the results of node construction; we restrict our attention to the possible nodes returned in the input document, where each node satisfies some type in the input schema. A more refined analysis would create a new "external" type to represent the presence of constructed nodes in the output (analogous to the approach taken in [16] in the context of path-based analysis). Currently we do not make this distinction, and have a judgement $\mathsf{S};\Gamma \vdash \mathtt{q} : \mathcal{A}$, where $\mathcal{A}$ is a set of type names in S. The rules are simplifications of the standard XQuery typing rules, and are found in Figure 1. These rules can be read as a (nondeterministic, partial) function that takes a schema S, static environment Γ, and query expression $\mathtt{q}$ and returns a set of types $\mathcal{A}$.

The key rules with respect to previous work are those for node construction and XPath axis steps, respectively. These rules make use of an auxiliary judgement $\mathsf{S} \vdash \mathcal{A}/ax::\phi \overset{\mathsf{step}}{\Rightarrow} \mathcal{A}'$ to model static typechecking for XPath steps. For the purposes of the soundness proofs, one needs only that this judgement over-approximates the set of types of nodes that can be reached by applying the axis step $ax::\phi$ to a node satisfying a type name in $\mathcal{A}$. Our implementation computes exactly this set of types (see Subsection 4.1).

REMARK 1. *This analysis is (intentionally) simplistic: unlike XQuery's static type system (or more sophisticated type systems such as that of Colazzo et al. [12]), we discard the regular expression structure of the data, since all we need for independence analysis is a set of type names. Our step judgement is as refined as possible given that we only deal with type names, but an analysis that treated the query result as an ordered set can be much more precise. On the other hand, our analysis does handle all XPath steps, whereas XQuery gives the results of ancestor or sibling axis steps the most general possible type, and Colazzo et al. do not handle these axes.*

$$\frac{}{\mathsf{S};\Gamma \vdash \mathtt{s} : \emptyset}$$

$$\frac{}{\mathsf{S};\Gamma \vdash x : \Gamma(x)}$$

$$\frac{}{\mathsf{S};\Gamma \vdash () : \emptyset}$$

$$\frac{\mathsf{S};\Gamma \vdash \mathtt{q}_1 : \mathcal{A}_1 \quad \mathsf{S};\Gamma \vdash \mathtt{q}_2 : \mathcal{A}_2}{\mathsf{S};\Gamma \vdash \mathtt{q}_1, \mathtt{q}_2 : \mathcal{A}_1 \cup \mathcal{A}_2}$$

$$\frac{\mathsf{S};\Gamma \vdash \mathtt{q}_1 : \mathcal{A}_1 \quad \mathsf{S};\Gamma \vdash \mathtt{q}_2 : \mathcal{A}_2}{\mathsf{S};\Gamma \vdash \mathtt{if\ q\ then\ q_1\ else\ q_2} : \mathcal{A}_1 \cup \mathcal{A}_2}$$

$$\frac{\mathsf{S};\Gamma \vdash \mathtt{q}_1 : \mathcal{A}_1 \quad \mathsf{S};\Gamma, x : \mathcal{A}_1 \vdash \mathtt{q}_2 : \mathcal{A}_2}{\mathsf{S};\Gamma \vdash \mathtt{let}\ x := \mathtt{q}_1\ \mathtt{in}\ \mathtt{q}_2 : \mathcal{A}_2}$$

$$\frac{\mathsf{S} \vdash \Gamma(x)/ax::\phi \overset{\mathsf{step}}{\Rightarrow} \mathcal{A}}{\mathsf{S};\Gamma \vdash x/ax::\phi : \mathcal{A}}$$

$$\frac{\mathsf{S} \vdash \Gamma(x)/\mathtt{desc-or-self}::* \overset{\mathsf{step}}{\Rightarrow} \mathcal{A}}{\mathsf{S};\Gamma \vdash x/text() : \mathcal{A}}$$

$$\frac{\mathsf{S};\Gamma \vdash \mathtt{q}_1 : \mathcal{A} \quad \mathsf{S};\Gamma, x : \mathcal{A} \vdash \mathtt{q}_2 : \mathcal{A}'}{\mathsf{S};\Gamma \vdash \mathtt{for}\ x \in \mathtt{q}_1\ \mathtt{return}\ \mathtt{q}_2 : \mathcal{A}'}$$

$$\frac{}{\mathsf{S};\Gamma \vdash \langle a \rangle \mathtt{q} \langle /a \rangle : \emptyset}$$

Figure 1: Input type inference rules

For selection queries, the correctness of this judgement is easy to state and prove. In the presence of node-construction, the correctness criterion is a bit subtle:

THEOREM 2 (TYPE SOUNDNESS). *Suppose* $\sigma \models_{\mathsf{S}} \gamma : \Gamma$ *and* $\mathsf{S};\Gamma \vdash \mathtt{q} : \mathcal{A}$. *If* $\sigma, \gamma \models \mathtt{q} \Rightarrow \sigma_2, \mathtt{L}$ *then for every* $\mathtt{l}$ *in* $\mathtt{L}$ *such that* $\mathtt{l} \in \mathrm{dom}(\sigma)$, *there exists a type* $\mathtt{T} \in \mathcal{A}$ *such that* $\sigma_2 \models_{\mathsf{S}} \mathtt{l} : \mathtt{T}$.

The proof is outlined in [3].

Update impact analysis. We next turn to the problem of statically approximating the behavior of the update. In previous work [4] we developed a complicated analysis that approximates the set of possible pending update lists generated by an update. However, here we will simplify matters by approximating only a set of nodes "impacted" by an update.

DEFINITION 4 (IMPACTED NODES). *Given a store* σ, *we say a node in* σ *is* impacted *by an atomic update sequence* ω *on* σ *if it is a target of a* `rename` *or* `insert into` *command, or the* parent *of a target of a* `delete`, `replace`, `insert before` *or* `insert after`

command. Similarly, given a store σ and variable environment γ, a node is impacted *by an update expression u if it is impacted by the atomic update sequence generated by u.*

Intuitively, the impacted nodes of an update are the nodes whose label or child sequence is changed by the update.

The *impacted types* for an update u schema S and static environment Γ is a set of type names $\mathcal{A}$ of S, provided that, in any σ, γ consistent with S, Γ, each impacted element node of u on σ, γ satisfies a type in $\mathcal{A}$, and each impacted text node has a parent that satisfies a type in $\mathcal{A}$.

We use a judgement:

$$\mathsf{S}; \Gamma \vdash \mathtt{u} \text{ impacts } \mathcal{A}$$

to infer the impacted types. The judgement is given in Figure 2. Note that the impact analysis judgement makes use of the query type inference judgement in the rules for `for`, `let`, and atomic updates.

The soundness of this judgement is stated as follows:

THEOREM 3 (IMPACT SOUNDNESS). *Suppose $\sigma \models_{\mathsf{S}} \gamma : \Gamma$ and $\mathsf{S}; \Gamma \vdash \mathtt{u}$ impacts $\mathcal{A}$. If $\sigma, \gamma \models \mathtt{u} \Rightarrow \sigma_2, \omega$ then for every node $\mathtt{l} \in \mathrm{dom}(\sigma)$ that is impacted by ω, there exists a type $\mathtt{T} \in \mathcal{A}$ such that $\sigma \models_{\mathsf{S}} \mathtt{l} : \mathtt{T}$.*

The proof follows easily from the definition of impact set, plus the soundness of type inference. We discuss a few cases.

- `let` and `for` are handled similarly (in rules (`ILet`) and (`IFor`)), given our set-based abstraction. The types of nodes in the input store that can be returned by the query are determined, using a call to the type-inference judgement in Figure 1. The static context is then expanded by assigning this set of types to the newly-bound variable x.
- Because `insert into` commands impact the target of the insert, we statically approximate the impact set by the types of these targets (in rule (`IInsInto`)). The types are likewise calculated by a call to the type-inference judgement. The same comment applies to `rename` (in rule (`IRename`)).
- `insert before` and `insert after` commands impact the parent of the target. The types of such parents are approximated by first estimating the target types using type-inference, and then tracing their parents in the schema using the step-judgement (in rule (`IInsSib`)). The same approach is used for `replace` and `delete` commands (in rules (`IReplace`) and (`IDel`)).

Access Set Analysis. To determine whether an update interacts with a query, we need an abstraction of the nodes the query "accesses" (or those on which the query "depends"). This is similar to the concepts of the "accessed nodes" [16] or the "projection" of a query [18]. As pointed out in both these works, the notion of accessed nodes is subtle; we will begin by looking at the corresponding runtime notion. Intuitively, if the nodes accessed and returned are disjoint from those modified by an update, this should imply that the query and update are independent.

DEFINITION 5 (I-SIMILARITY AND N-SIMILARITY). *For a set of node identifiers* I *we say two stores σ and σ_2 are* I*-similar (written $\sigma \simeq_{\mathtt{I}} \sigma_2$) provided that for every identifier* i *in* I*, we have*

1. *there are nodes* l *in σ and* $\mathtt{l}'$ *in σ_2 with identifier* i*, and these nodes have the same label.*
2. *if* l *and* $\mathtt{l}'$ *are as above, then there is a bijection from the children of* l *to the children of* $\mathtt{l}'$ *preserving node identifiers and sibling order.*

For a set of element nodes N *in σ, we say $\sigma \simeq_{\mathtt{N}} \sigma_2$ iff $\sigma \simeq_{I(\mathtt{N})} \sigma_2$ where $I(\mathtt{N})$ is the set of identifiers of* N*.*

Thus if two stores are I-similar then the children and labeling of locations in I are indistinguishable. From now on, we will generally identify a node with its identifier, and if $\sigma \simeq_{\mathtt{N}} \sigma_2$ we will say that the nodes in N and their children are "still in σ_2", when technically we mean that there are nodes with the same identifiers in σ_2.

Our notion of N being a set of "accessed nodes" for a query q will be in terms of N-similarity preserving q. In the case of a selection query, we require that the set of accessed nodes be such that: if two stores agree on them, then the query returns the same list of locations. In the case of general queries, we require that the list being returned is "the same up to renaming constructed or copied nodes".

DEFINITION 6 (DYNAMIC ACCESS COVER). *Let* q *be a query, σ_1 an input store, and γ an environment. Suppose $\sigma_1, \gamma \models \mathtt{q} \Rightarrow \mathtt{L}, \sigma_2$ with $\mathtt{L} = \mathtt{l}_1 \ldots \mathtt{l}_k$.*

If q *is a selection query, we say that* N *is a dynamic access cover for* q *on σ, γ provide that for any σ_1' containing all locations in γ with $\sigma_1' \simeq_{\mathtt{N}} \sigma$, we have $\sigma_1', \gamma \models \mathtt{q} \Rightarrow \mathtt{L}, \sigma_2'$ for some σ_2'.*

$$\frac{}{\mathsf{S}; \Gamma \vdash () \text{ impacts } \emptyset} \ (\mathtt{IEmp})$$

$$\frac{\mathsf{S}; \Gamma \vdash \mathtt{u}_1 \text{ impacts } \mathcal{A}_1 \quad \mathsf{S}; \Gamma \vdash \mathtt{u}_2 \text{ impacts } \mathcal{A}_2}{\mathsf{S}; \Gamma \vdash \mathtt{u}_1, \mathtt{u}_2 \text{ impacts } \mathcal{A}_1 \cup \mathcal{A}_2} \ (\mathtt{ISeq})$$

$$\frac{\mathsf{S}; \Gamma \vdash \mathtt{q} : \mathcal{A} \quad \mathsf{S}; \Gamma, x : \mathcal{A} \vdash \mathtt{u} \text{ impacts } \mathcal{A}'}{\mathsf{S}; \Gamma \vdash \mathtt{let}\ x := \mathtt{q}\ \mathtt{in}\ \mathtt{u} \text{ impacts } \mathcal{A}'} \ (\mathtt{ILet})$$

$$\frac{\mathsf{S}; \Gamma \vdash \mathtt{u}_1 \text{ impacts } \mathcal{A}_1 \quad \mathsf{S}; \Gamma \vdash \mathtt{u}_2 \text{ impacts } \mathcal{A}_2}{\mathsf{S}; \Gamma \vdash \mathtt{if}\ \mathtt{q}\ \mathtt{then}\ \mathtt{u}_1\ \mathtt{else}\ \mathtt{u}_2 \text{ impacts } \mathcal{A}_1 \cup \mathcal{A}_2} \ (\mathtt{ICond})$$

$$\frac{\mathsf{S}; \Gamma \vdash \mathtt{q} : \mathcal{A} \quad \mathsf{S}; \Gamma, x : \mathcal{A} \vdash \mathtt{q}' \text{ impacts } \mathcal{A}'}{\mathsf{S}; \Gamma \vdash \mathtt{for}\ x \in \mathtt{q}\ \mathtt{return}\ \mathtt{q}' \text{ impacts } \mathcal{A}'} \ (\mathtt{IFor})$$

$$\frac{\mathtt{d} \in \{\downarrow, \swarrow, \searrow\} \quad \mathsf{S}; \Gamma \vdash \mathtt{q}' : \mathcal{A}}{\mathsf{S}; \Gamma \vdash \mathtt{insert}\ \mathtt{q}\ \mathtt{d}\ \mathtt{q}' \text{ impacts } \mathcal{A}} \ (\mathtt{IInsInto})$$

$$\frac{\mathtt{d} \in \{\leftarrow, \rightarrow\} \quad \mathsf{S}; \Gamma \vdash \mathtt{q}' : \mathcal{A} \quad \mathsf{S} \vdash \mathcal{A}/\mathtt{parent} :: * \overset{\mathsf{step}}{\Rightarrow} \mathcal{A}'}{\mathsf{S}; \Gamma \vdash \mathtt{insert}\ \mathtt{q}\ \mathtt{d}\ \mathtt{q}' \text{ impacts } \mathcal{A}'} \ (\mathtt{IInsSib})$$

$$\frac{\mathsf{S}; \Gamma \vdash \mathtt{q} : \mathcal{A}}{\mathsf{S}; \Gamma \vdash \mathtt{rename}\ \mathtt{q}\ \mathtt{as}\ a \text{ impacts } \mathcal{A}} \ (\mathtt{IRename})$$

$$\frac{\mathsf{S}; \Gamma \vdash \mathtt{q} : \mathcal{A} \quad \mathsf{S} \vdash \mathcal{A}/\mathtt{parent} :: * \overset{\mathsf{step}}{\Rightarrow} \mathcal{A}'}{\mathsf{S}; \Gamma \vdash \mathtt{replace}\ \mathtt{q}\ \mathtt{with}\ \mathtt{q}' \text{ impacts } \mathcal{A}'} \ (\mathtt{IReplace})$$

$$\frac{\mathsf{S}; \Gamma \vdash \mathtt{q} : \mathcal{A} \quad \mathsf{S} \vdash \mathcal{A}/\mathtt{parent} :: * \overset{\mathsf{step}}{\Rightarrow} \mathcal{A}'}{\mathsf{S}; \Gamma \vdash \mathtt{delete}\ \mathtt{q} \text{ impacts } \mathcal{A}'} \ (\mathtt{IDel})$$

Figure 2: Update impact rules

For q *a general query, we say* N *is a dynamic access cover if for any σ_1' as above, we have $\sigma_1', \gamma \models \mathtt{q} \Rightarrow \mathtt{L}', \sigma_2'$, where $\mathtt{L}' = \mathtt{l}_1' \ldots \mathtt{l}_k'$, and there is a bijection f from the range of* L *to the range of* $\mathtt{L}'$ *such that:*

- $\forall i \leq k.\ \mathtt{l}_i' = f(\mathtt{l}_i)$,
- *f preserves node identifiers on σ_1,*
- *for every node n, the isomorphism type of n within its connected component is the same as the isomorphism type of $f(n)$ within its component.*

Notice that if N is a dynamic access cover for a selection query q on σ_1, γ, and we update σ_1 to get store σ_2 without touching N, then we know only that the *locations* in σ_1 returned by q are unchanged. However, the labels of these locations, as well as locations in the subtrees underneath these nodes may still change. Thus for an update to be independent of a query, we will need to know a bit more than the fact that it does not update anything in an access cover. For example if $\mathtt{q} = \$doc/\mathtt{child}::a$, then an access cover for q on σ_1 would include the nodes pointed to by $\$doc$ and their children. An update to to σ_1 that changes a grandchild of $\$doc$ may not be independent of q, even though such an update does not impact the access cover for q.

Of course, we also want a static notion of access cover that approximates the dynamic one.

DEFINITION 7 (STATIC ACCESS COVER). *Given schema* S*, selection query* q *and static environment* Γ*, a Static Access Cover is a set of type names $\mathcal{A}$ from* S *such that whenever σ, γ is consistent with* S, Γ *and D is all the elements nodes in σ that can be assigned to a type in $\mathcal{A}$ in σ, then D is a Dynamic Access Cover for* q *on σ, γ.*

The judgement $\mathsf{S}; \Gamma \vdash_{\mathsf{SAC}} \mathtt{q} : \mathcal{A}$ allows us to compute, given a schema S, static environment Γ, and query q, a set of type names $\mathcal{A}$ in S that is a Static Access Cover. The rules are shown in Figure 3. Formally, the desired correctness property is:

THEOREM 4 (ACCESS SOUNDNESS). *If* $\mathsf{S}; \gamma \vdash_{\mathsf{SAC}} \mathtt{q} : \mathcal{A}$ *then $\mathcal{A}$ is a static access cover for* q *in σ.*

We will explain the most interesting cases below, assuming for the moment that q is a selection query.

- **Rule (Var)** The empty set of types is a static cover for a variable access, because the empty set of nodes is a dynamic cover. This is because if a variable x points to location l in a store σ, and we "update σ" – change it to some σ' that still has location l in it – the locations returned by the query x are the same. Renamings may change the label of l, deletes may detach l from its parent, but the query will still return l, which is all we require for an access cover.
- **Rule (Text)** The corresponding runtime claim is that given a store σ and environment where variable x points to a location l, if N is the set of all element descendants of l, then N forms a dynamic access cover for $x/text()$. If we modify σ to get a store σ_2 N-similar to σ, then the set of element descendants of l *and their children* will be the same (by the definition of N-similarity). Hence the collection of text nodes returned will be the same.
- **Rules (Self1) - (Self2)** The runtime claim for the label test version is that given σ and variable x pointing to a location l

$$\frac{}{\mathsf{S}; \Gamma \vdash_{\mathsf{SAC}} x : \emptyset}\ (\mathtt{Var})$$

$$\frac{}{\mathsf{S}; \Gamma \vdash_{\mathsf{SAC}} () : \emptyset}\ (\mathtt{Empty})$$

$$\frac{\mathsf{S}; \Gamma \vdash_{\mathsf{SAC}} \mathtt{q}_1 : \mathcal{A}_1 \quad \mathsf{S}; \Gamma \vdash_{\mathsf{SAC}} \mathtt{q}_2 : \mathcal{A}_2}{\mathsf{S}; \Gamma \vdash_{\mathsf{SAC}} \mathtt{q}_1, \mathtt{q}_2 : \mathcal{A}_1 \cup \mathcal{A}_2}\ (\mathtt{Concat})$$

$$\frac{\mathsf{S}; \Gamma \vdash_{\mathsf{SAC}} \mathtt{q} : \mathcal{A} \quad \mathsf{S}; \Gamma \vdash_{\mathsf{SAC}} \mathtt{q}_1 : \mathcal{A}_1 \quad \mathsf{S}; \Gamma \vdash_{\mathsf{SAC}} \mathtt{q}_2 : \mathcal{A}_2}{\mathsf{S}; \Gamma \vdash_{\mathsf{SAC}} \mathtt{if\ q\ then\ q_1\ else\ q_2} : \mathcal{A} \cup \mathcal{A}_1 \cup \mathcal{A}_2}\ (\mathtt{IfThen})$$

$$\frac{\mathsf{S}; \Gamma \vdash_{\mathsf{SAC}} \mathtt{q}_1 : \mathcal{A}_1 \quad \mathsf{S}; \Gamma \vdash \mathtt{q}_1 : \mathcal{A}_2 \quad \mathsf{S}; \Gamma, x : \mathcal{A}_2 \vdash_{\mathsf{SAC}} \mathtt{q}_2 : \mathcal{A}_3}{\mathsf{S}; \Gamma \vdash_{\mathsf{SAC}} \mathtt{let}\ x := \mathtt{q_1}\ \mathtt{in}\ \mathtt{q_2} : \mathcal{A}_1 \cup \mathcal{A}_3}\ (\mathtt{Let})$$

$$\frac{\mathsf{S} \vdash \Gamma(x)/\mathtt{descendant}::* \stackrel{\mathsf{step}}{\Rightarrow} \mathcal{A}}{\mathsf{S}; \Gamma \vdash_{\mathsf{SAC}} x/text() : \Gamma(x) \cup \mathcal{A}}\ (\mathtt{Text})$$

$$\frac{}{\mathsf{S}; \Gamma \vdash_{\mathsf{SAC}} x/\mathtt{self}::a : \Gamma(x)}\ (\mathtt{Self1})$$

$$\frac{}{\mathsf{S}; \Gamma \vdash_{\mathsf{SAC}} x/\mathtt{self}::* : \emptyset}\ (\mathtt{Self2})$$

$$\frac{ax \text{ sibl. axis} \quad \mathsf{S} \vdash \Gamma(x)/ax::* \stackrel{\mathsf{step}}{\Rightarrow} \mathcal{A} \quad \mathsf{S} \vdash \Gamma(x)/\mathtt{parent}::* \stackrel{\mathsf{step}}{\Rightarrow} \mathcal{A}'}{\mathsf{S}; \Gamma \vdash_{\mathsf{SAC}} x/ax::a : \mathcal{A} \cup \mathcal{A}'}\ (\mathtt{Sib1})$$

$$\frac{ax \text{ sibl. axis} \quad \mathsf{S} \vdash \Gamma(x)/\mathtt{parent}::* \stackrel{\mathsf{step}}{\Rightarrow} \mathcal{A}}{\mathsf{S}; \Gamma \vdash_{\mathsf{SAC}} x/ax::* : \mathcal{A}}\ (\mathtt{Sib2})$$

$$\frac{ax \text{ parent or ancestor axis} \quad \mathsf{S} \vdash \Gamma(x)/ax::* \stackrel{\mathsf{step}}{\Rightarrow} \mathcal{A}}{\mathsf{S}; \Gamma \vdash_{\mathsf{SAC}} x/ax::\phi : \mathcal{A}}\ (\mathtt{Up})$$

$$\frac{\mathsf{S} \vdash \Gamma(x)/\mathtt{child}::* \stackrel{\mathsf{step}}{\Rightarrow} \mathcal{A}}{\mathsf{S}; \Gamma \vdash_{\mathsf{SAC}} x/\mathtt{child}::a : \Gamma(x) \cup \mathcal{A}}\ (\mathtt{Child1})$$

$$\frac{}{\mathsf{S}; \Gamma \vdash_{\mathsf{SAC}} x/\mathtt{child}::* : \Gamma(x)}\ (\mathtt{Child2})$$

$$\frac{\mathsf{S} \vdash \Gamma(x)/\mathtt{descendant}::* \stackrel{\mathsf{step}}{\Rightarrow} \mathcal{A}}{\mathsf{S}; \Gamma \vdash_{\mathsf{SAC}} x/\mathtt{descendant}::\phi : \Gamma(x) \cup \mathcal{A}}\ (\mathtt{Desc})$$

$$\frac{\mathsf{S}; \Gamma \vdash_{\mathsf{SAC}} x/\mathtt{descendant}::\phi : \mathcal{A} \quad \mathsf{S}; \Gamma \vdash_{\mathsf{SAC}} x/\mathtt{self}::\phi : \mathcal{A}'}{\mathsf{S}; \Gamma \vdash_{\mathsf{SAC}} x/\mathtt{desc-or-self}::\phi : \mathcal{A} \cup \mathcal{A}'}\ (\mathtt{DOS})$$

$$\frac{\mathsf{S}; \Gamma \vdash_{\mathsf{SAC}} x/\mathtt{ancestor}::\phi : \mathcal{A} \quad \mathsf{S}; \Gamma \vdash_{\mathsf{SAC}} x/\mathtt{self}::\phi : \mathcal{A}'}{\mathsf{S}; \Gamma \vdash_{\mathsf{SAC}} x/\mathtt{anc-or-self}::\phi : \mathcal{A} \cup \mathcal{A}'}\ (\mathtt{AOS})$$

$$\frac{\mathsf{S}; \Gamma \vdash_{\mathsf{SAC}} \mathtt{q}_1 : \mathcal{A}_1 \quad \mathsf{S}; \Gamma \vdash \mathtt{q}_1 : \mathcal{A}_2 \quad \mathsf{S}; \Gamma, x : \mathcal{A}_2 \vdash_{\mathsf{SAC}} \mathtt{q}_2 : \mathcal{A}_3}{\mathsf{S}; \Gamma \vdash_{\mathsf{SAC}} \mathtt{for}\ x \in \mathtt{q_1}\ \mathtt{return}\ \mathtt{q_2} : \mathcal{A}_1 \cup \mathcal{A}_3}\ (\mathtt{For})$$

$$\frac{\mathsf{S}; \Gamma \vdash \mathtt{q} : \mathcal{A} \qquad \mathsf{S}; \Gamma \vdash_{\mathsf{SAC}} \mathtt{q} : \mathcal{A}'' \quad \mathsf{S} \vdash \mathcal{A}/\mathtt{desc-or-self} :: * \stackrel{\mathsf{step}}{\Rightarrow} \mathcal{A}'}{\mathsf{S}; \Gamma \vdash_{\mathsf{SAC}} \langle a \rangle \mathtt{q} \langle / a \rangle : \mathcal{A}' \cup \mathcal{A}''}\ (\mathtt{EltCon})$$

$$\frac{}{\mathsf{S}; \Gamma \vdash_{\mathsf{SAC}} \mathtt{s} : \emptyset}\ (\mathtt{StrCon})$$

Figure 3: Access Cover Algorithm

then l itself forms a dynamic access cover for $x/\texttt{self}::a$. If σ_2 is $\{l\}$-similar to σ, then the label of l in σ_2 is the same, hence the label test will return the same in σ_2 as in σ. The wildcard version $\texttt{self}::*$ is the same as the variable case in Rule (Var), and hence also accesses nothing.

- **Rules** (Sib1) **-** (Sib2) Consider the label-test version $ax::a$ in Rule (Sib1). The runtime claim is that given σ and variable x pointing to a location l then if N contains the parent of l unioned with the set of nodes resulting from applying this sibling axis step to l, then N is a dynamic access cover. If σ_2 is N-similar to σ, then since the parent of l is in N, the collection of siblings of l will be the same in σ_2 as in σ, and have the same sibling order. Furthermore, the labels of the siblings in the direction given by ax will be unchanged. Hence the label test will return the same in σ_2 as in σ.

  For the wildcard version in Rule (Sib2), note that we no longer require that the labels of the siblings remain the same. Hence in the argument above, we do not need N to contain the siblings.

- **Rules** (Child1) **-** (Child2) Consider the label-test version $\texttt{child}::a$ in Rule (Child1). The runtime claim is that given σ and variable x pointing to a location l then a set N containing l and all its children, is a dynamic access cover. If σ_2 is N-similar to σ, then since l itself is in N, the set of children of l is the same in σ_2 as in σ, since N-similarity requires preservation of children. Since the children of l are in N, the labels of the children are all preserved, and hence the set of children passing the label test is unaffected as well.

  For the wildcard version, $\texttt{child}::*$ in Rule (Child2), note that we no longer require that the labels of children remain the same, and hence we do not need N to contain the children.

In the discussion above, we have ignored the presence of node construction. Node construction is a subtle issue for the schema-based approach, since the new documents that result do not satisfy the input schema. A fine-grained analysis would analyze the structure of the constructed nodes (e.g. inferring a new schema), and then track navigation within them. In our approach, we do not do such tracking, but rather assume that the constructed document is immediately navigated in its entirety. The rule (EltCon) states that the nodes accessed by $\langle a\rangle \mathtt{q} \langle/a\rangle$ are those accessed by q plus all the non-strict descendants of nodes in the input document returned by q. That is, when we copy a node into the new document, the result may now be impacted by any changes below the node.

In the presence of node construction, formally what we calculate is a set of type names $\mathcal{A}$ in the input document such that: for any store σ satisfying the schema and environment γ, for any two extensions σ_2, γ' and σ_3, γ' which may have arbitrary nodes added to variable assignments, but only nodes disconnected from those in σ, where σ_3 and σ_2 agree on all nodes in $\mathcal{A}$ as well as all nodes outside of σ, then the results of q on σ and σ_2 are equivalent – i.e. satisfy the conclusion of Definition 6. (Note that updates such as node deletion can also invalidate the schema but this is irrelevant since Theorem 4 concerns only queries).

Aliasing. To obtain a safe analysis, we need to know when two types may or must not "alias". We say that T and $\mathtt{T}'$ may alias (with respect to S) provided that for some σ and $\mathtt{l} \in \mathrm{dom}(\sigma)$, we have $\sigma \models_{\mathsf{S}} \mathtt{l} : \mathtt{T}$ and $\sigma \models_{\mathsf{S}} \mathtt{l} : \mathtt{T}'$. There is a tractable exact algorithm for determining non-aliasing of types T and $\mathtt{T}'$: convert the schema to a non-deterministic tree automaton A [25] in which types T and $\mathtt{T}'$ will each correspond to states. In case there are root types, these correspond to the final states; otherwise all states are final. In the product A^2 perform reachability analysis to see if the product state $(\mathtt{T}, \mathtt{T}')$ can be inhabited by a run that reaches a final state. A similar analysis is done in [23]. For the purposes of this paper we assume that we are given a procedure $\mathsf{S} \vdash \mathtt{T} \sqcap \mathtt{T}'$ such that if T and $\mathtt{T}'$ may alias we have $\mathsf{S} \vdash \mathtt{T} \sqcap \mathtt{T}'$.

Independence Testing. Finally, we assemble the components of this section to give an independence test. The algorithm is summarized in Algorithm 3.1. As per the preceding discussion, it is not sound to simply test that the static access cover of the query is disjoint from the impact set of the update — this is necessary, but not sufficient. We must also ensure that the update cannot modify any of the tree structure under the nodes returned by the query. Thus, for update u and query q to be independent, it suffices that u does not update any type accessed by q or any type below something returned by q. We formalize this as stating the following independence test:

THEOREM 5. *For a schema* S *and static environment* Γ*, suppose that*

- $\mathcal{A}$ *is a Static Access Cover for selection query* q *and* Γ*,*
- $\mathcal{A}'$ *is such that* $\mathsf{S}; \Gamma \vdash \mathtt{q} : \mathcal{A}'$
- $\mathcal{A}''$ *is such that* $\mathsf{S} \vdash \mathcal{A}'/\texttt{desc} - \texttt{or} - \texttt{self}::* \overset{\mathsf{step}}{\Rightarrow} \mathcal{A}''$
- $\mathcal{A}'''$ *is the set of impacted types for update* u *and* Γ*.*

Then if no type in either $\mathcal{A}$ *or* $\mathcal{A}''$ *aliases a type in* $\mathcal{A}'''$*, then* u *and* q *are independent.*

The theorem proves the soundness of Algorithm 3.1.

Algorithm 3.1 Sound Test for Independence

(Independence Test)
Input*: A schema* S*, static environment* Γ*, query* q*, and update* u
Output*: yes if* q *and* u *are found to be independent on* S, Γ *false otherwise*

Calculate $\mathcal{A}$ *such that* $\mathsf{S}; \Gamma \vdash_{\mathsf{SAC}} \mathtt{q} : \mathcal{A}$ *using Figure 3*
Calculate $\mathcal{A}'$ *such that* $\mathsf{S}; \Gamma \vdash \mathtt{q} : \mathcal{A}'$ *using Figure 1*
Calculate $\mathcal{A}''$ *such that* $\mathsf{S} \vdash \mathcal{A}'/\texttt{desc} - \texttt{or} - \texttt{self}::* \overset{\mathsf{step}}{\Rightarrow} \mathcal{A}''$
Calculate $\mathcal{A}'''$ *such that* $\mathsf{S}; \Gamma \vdash \mathtt{u}$ *impacts* $\mathcal{A}'''$ *using Figure 2*
If $\exists \mathtt{T} \in \mathcal{A} \cup \mathcal{A}''.\ \exists \mathtt{T}' \in \mathcal{A}'''.\ \mathsf{S} \vdash \mathtt{T} \sqcap \mathtt{T}'$ **then return** *false*
Else return *true*

Complexity of the analysis. The most expensive step of the static analysis is the step calculation $\mathsf{S} \vdash \mathtt{T}/ax::\phi \overset{\mathsf{step}}{\Rightarrow} \mathcal{A}'$. Our implementation uses a straightforward syntactic analysis of regular expressions to determine this relation for the sibling axes, child axis, and parent axis. We then perform a fixed-point iteration to handle the transitive vertical axes, resulting in a worst-case time of $O(|\mathsf{S}|^3)$. This relation takes at worst $O(|\mathsf{S}|^2)$ space and can be precomputed once and for all. The aliasing relation, which takes at most quadratic time and space, can likewise be precomputed. The input type inference algorithm makes only one call to each subformula, with the size of the context argument being bounded linearly in the size of the schema: it thus runs in time $O(|\mathsf{S}'| + |\mathtt{q}|)$, where S' is the size of the reachability relation precomputed from the schema. The update impact algorithm is likewise in $O(|\mathsf{S}'| + |\mathtt{u}|)$. The access cover algorithm of Figure 3 runs in time $O((|\mathsf{S}'| + |\mathtt{q}|)^2)$, since at most two recursive calls are made to each subquery. The final algorithm makes only linearly many calls to each of these components, and hence runs in time $O((|\mathsf{S}'| + |\mathtt{q}|)^2 + |\mathtt{u}|) \leq O((|\mathsf{S}|^2 + |\mathtt{q}|)^2 + |\mathtt{u}|)$.

4. EXPERIMENTAL EVALUATION

4.1 Implementation

We implemented our independence analysis in OCaml. The prototype currently handles the core fragment of XQuery discussed in this paper, plus some syntactic sugar (including the `following` and `preceding` axes, which are compositions of the basic axes in this paper). The XMark and XPathMark queries we used can all be translated to this fragment.

Our experiments only involved DTDs, for which alias analysis is trivial: two type names can alias if and only if they are equal (since each type has a unique element tag and no types can be empty in a DTD). Therefore we used the obvious constant-time alias test instead of the more general quadratic test that would be needed for XML Schemas or general tree automata.

Our implementation employs a schema data structure that precomputes the sets of possible children, parents, following siblings, and preceding siblings of each type name. These sets are easy to compute once at the beginning of computation. We do not precompute the other axes because these are more expensive and less frequently needed. Instead, we compute them on-the-fly only as needed.

4.2 Benchmarks and Experimental Setup

Our measurements were performed on an Intel Pentium D (3.0 Ghz) running Ubuntu Linux 8.10. We used the XMark random data generator to generate test documents of sizes 1.1MB, 2.3MB, 5.7MB and 11MB. We used a standard installation of Galax 1.1 to measure query and update processing times. Galax 1.1 supports the W3C XQuery Update Facility 1.0 via a command-line option, and we used this option to run the updates.

We constructed a view maintenance benchmark using all of the XMark [24] queries and some of the XPathMark [19] queries (A1–8 and B1–8). All of these queries operate on the XMark data, for which there is a standard schema available (auction.dtd). The queries exercise all of the features of our XQuery core language, including all XPath axes, the $text()$ node test, and element node construction, as illustrated by Table 2. We also included a trivial query $Q_0 = ()$ that has no effect and a trivial update $U_0 = \$auction$ that returns the input document unmodified. We observed that Galax always fully parses its input by default and so these trivial queries and updates can be used to determine (and adjust for) the fixed common costs of loading data and (for updates) saving the results.

We regard all of these queries as possible materialized views on the data, and we also used the XPathMark queries as the basis of updates. For each XPathMark query p named A_i, B_i, we define updates UA_i or UB_i respectively to be the deletion updates of the form `delete` p. We only considered deletion in the experiments because our update analysis ignores (almost) all information about the type of update performed.

Moreover, since deletion always *decreases* the amount of data, the time to perform a deletion is generally a lower bound on the time needed to perform other kinds of updates, and similarly the time needed to re-evaluate a query after a deletion is a lower bound on the time needed to re-evaluate after other kinds of updates. Thus, if our analysis is effective when used with deletion-only updates, then it will likely be competitive with updates performing other operations or performing a mixture of operations.

4.3 Experimental Results

Validity and Precision. For each update and query pair (U, Q), we checked whether U and Q are (dynamically) independent with respect to the fixed (1.1MB) document. We also checked independence statically for each such pair. Table 3 shows the results. In Table 3, S indicates that static independence check succeeded, and and D indicates that the query and update were dynamically independent on the 1.1MB document.

Table 2: Features used by queries and updates. The updates are based on the XPathMark queries A1–A8 and B1–B8 and so their rows are combined.

Query#	child	text()	node	descendant	parent	ancestor	sibling
Q0							
(U)A1	X						
(U)A2	X			X			
(U)A3	X			X			
(U)A4	X						
(U)A5	X			X			
(U)A6	X						
(U)A7	X						
(U)A8	X						
(U)B1	X				X		
(U)B2	X					X	
(U)B3	X						X
(U)B4	X						X
(U)B5	X				X	X	X
(U)B6	X				X	X	X
(U)B7	X			X			
(U)B8	X						X
Q1	X	X					
Q2	X	X	X				
Q3	X	X	X				
Q4	X	X	X				
Q5	X	X					
Q6	X			X			
Q7	X			X			
Q8	X	X	X				
Q9	X	X	X				
Q10	X	X	X				
Q11	X	X	X				
Q12	X	X	X				
Q13	X	X	X				
Q14	X	X	X				
Q15	X	X		X			
Q16	X	X	X				
Q17	X	X	X				
Q18	X						
Q19	X	X	X	X			
Q20	X		X				

Update evaluation time. We measured the time needed to evaluate each of the updates on XMark documents of varying sizes. For each update, we measured the time needed by Galax to load the document, perform the update, and store the updated document. We also measured the time Galax needed just to load and store the document without making any changes. The difference between these two times is reported as the update processing time in Figure 4.

View maintenance. For each update, we measured the cost of maintaining the views using independence analysis to avoid recomputing views that are independent of the update. We measured the time needed to perform independence checks (t_c^{ind}) and the time needed to recompute views that could not be certified independent of the update (t_r^{ind}). Table 4 shows these measurements, along with the total independence-based maintenance time, t_m^{ind}.

The "Saved" and "Save%" columns of Table 4(a) and (b) show the total time saved (in seconds) and the percentage improvement over the naive approach, for the 1.1MB and 2.3MB documents respectively. Both figures are negative in some cases, indicating that checking independence took (slightly) more time than was saved through avoiding recomputation.

Table 3: Query-update independence results. "D" indicates dynamic independence on the 1.1MB document; "S" indicates static analysis was able to verify independence. Note that the static analysis algorithm is sound but incomplete (at least on this document).

	U0	UA1	UA2	UA3	UA4	UA5	UA6	UA7	UA8	UB1	UB2	UB3	UB4	UB5	UB6	UB7	UB8
Q0	DS	DS	DS	DS	DS	DS	DS	DS	DS	DS	DS	DS	DS	DS	DS	DS	DS
A1	DS				D	D	DS	DS	DS	DS	D	DS	DS	DS	DS	DS	DS
A2	DS				D	D	D	D	D	D		D	D	D	D	D	D
A3	DS				D	D	D	D	D	D		D	D	D	D	D	D
A4	DS						DS	DS	DS	DS	D	DS	DS	DS	DS	DS	DS
A5	DS						D	D	D	D		D	D	D	D	D	D
A6	DS	DS	DS	DS	DS	DS				D	DS	DS	DS	D	D	D	DS
A7	DS	DS	DS	DS	DS	DS				D	DS	DS	DS	D	D	D	DS
A8	DS	DS	DS	DS	DS	DS				D	DS	DS	DS	D	D	D	DS
B1	DS	D	DS	DS	DS	DS	D	D	D		DS	DS	DS			D	DS
B2	DS	D			D	D	D	D	D	D		D	D	D	D	D	D
B3	DS	DS	DS	DS	D	D	DS	DS	DS	DS	DS			DS	DS	DS	D
B4	DS	DS	DS	DS	D	D	DS	DS	DS	DS	DS			DS	DS	DS	D
B5	DS	D	D	D	D	D	D	D	D		D	D	D			D	D
B6	DS	D	D	D	D	D	D	D	D		D	DS	DS			D	DS
B7	DS	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D
B8	DS	DS	DS	DS	D	D	DS	DS	DS	DS	DS			DS	DS	DS	
Q1	DS	DS	DS	DS	DS	DS				D	DS	DS	DS	D	D	D	DS
Q2	DS	DS	DS	DS	D	D	DS	DS	DS	DS	DS			DS	DS	DS	D
Q3	DS	DS	DS	DS	D	D	DS	DS	DS	DS	DS			DS	DS	DS	D
Q4	DS	DS	DS	DS	D	D	DS	DS	DS	DS	DS	D	D	DS	DS	DS	D
Q5	DS	DS	DS	DS	D	D	DS	DS	DS	DS	DS	DS	DS	DS	DS	DS	DS
Q6	DS	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D
Q7	DS	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D
Q8	DS	D	D	D	D	D				D	D	D	D	D	D	D	D
Q9	DS	D	DS	DS	D	D				D	DS	DS	DS	D	D	D	DS
Q10	DS	DS	DS	DS	DS	DS	D	D	D	D	DS	DS	DS	D	D	D	DS
Q11	DS	DS	DS	DS	D	D				D	DS	D	D	D	D	D	D
Q12	DS	DS	DS	DS	D	D	D	D	D	D	DS	D	D	D	D	D	D
Q13	DS	DS	DS	DS	D	D	D	D	D	D	DS	D	D	D	D	D	D
Q14	DS	D	D	D	DS	DS	D	D	D	D		DS	DS			D	DS
Q15	DS	D	D	D	D	D	D	D	D		D	D	D			D	D
Q16	DS	D			D	D	DS	DS	DS	DS	D	DS	DS	DS	DS	DS	DS
Q17	DS	DS	DS	DS	DS	DS				D	DS	DS	DS	D	D	D	DS
Q18	DS	DS	DS	DS	D	D	DS	DS	DS	DS	DS	D	D	DS	DS	DS	D
Q19	DS	D	D	D	D	D	D	D	D		D	DS	DS			D	DS
Q20	DS	DS	DS	DS	DS	DS	D	D	D	D	DS	D	D	D	D	D	D

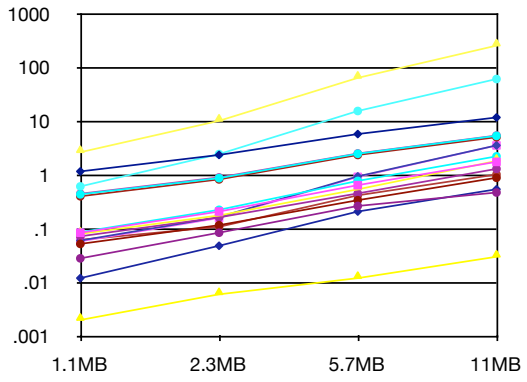

Figure 4: Times needed for benchmark updates (UA1–8, UBA–8)

4.4 Evaluation

The qualitative results in Table 3 show that there are significant opportunities for avoiding recomputation through independence analysis. Since our test is sound, there are (as expected) no query-update pairs in Table 3 for which static analysis predicts independence but the query and update conflict dynamically. Concerning completeness, it is difficult to draw conclusions. About 44% of the query/update pairs that are dynamically independent on the 1.1MB document are found to be statically independent — however, it is important to note that dynamic independence on a single document does not imply static independence, and some of the remaining 60% might conflict on a different valid document. Our analysis was successful in identifying nontrivial independence pairs in the presence of each of the features in Table 2. Certain kinds of queries seem to be inherently hard for our independence analysis to deal with; for example, queries B2, B5 and B7 involving sibling, ancestor and descendant axes and we were not able to prove any updates independent of this query. This is unfortunate because these queries are also among the most expensive.

Our experimental results show that query-update independence analysis is both precise and fast enough to be effective for view maintenance on the relatively small 1.1MB and 2.3MB documents. First, we observe that static analysis for a single query–update pair typically took under 12 milliseconds; almost all updates take longer for the 2.3MB document (see Figure 4). This implies that query-update independence analysis could be performed in parallel with update application without harming latency, as long as enough cores are available to process the independence checks in parallel with

Table 4: A comparison of naive and independence analysis-based view maintenance for (a) the 1.1MB document and (b) the 2.3MB document. t_r^{naive} is the naive recomputation time. t_r^{ind} is recomputation time using static independence checks. t_c^{ind} is time to perform independence analysis. t_m^{ind} is $t_r^{\text{ind}} + t_c^{\text{ind}}$. The next two columns show the amount of time saved (in seconds and as a percentage of the naive time). Each row summarizes execution times for maintaining all 37 queries. All times are in seconds.

(a)

Upd#	t_r^{naive}	t_r^{ind}	t_c^{ind}	t_m^{ind}	Saved	Save%
U0	9.04	0.00	0.36	0.36	8.68	96%
UA1	8.90	7.32	0.39	7.71	1.19	13%
UA2	8.95	6.57	0.42	6.99	1.96	22%
UA3	8.93	6.53	0.40	6.93	2.00	22%
UA4	8.91	8.52	0.39	8.91	0.00	0%
UA5	8.94	8.65	0.39	9.04	-0.10	-1%
UA6	8.91	8.35	0.39	8.74	0.17	2%
UA7	8.88	8.50	0.39	8.89	-0.01	0%
UA8	8.95	8.60	0.39	8.99	-0.04	0%
UB1	8.90	8.59	0.38	8.97	-0.07	-1%
UB2	8.86	6.55	0.42	6.97	1.89	21%
UB3	7.89	6.06	0.39	6.45	1.44	18%
UB4	7.89	6.11	0.38	6.49	1.40	18%
UB5	8.92	8.49	0.39	8.88	0.04	0%
UB6	8.92	8.32	0.38	8.70	0.22	2%
UB7	8.96	8.39	0.41	8.80	0.16	2%
UB8	8.98	7.23	0.39	7.62	1.36	15%

(b)

Upd#	t_r^{naive}	t_r^{ind}	t_c^{ind}	t_m^{ind}	Saved	Save%
U0	27.77	0.00	0.35	0.35	27.42	99%
UA1	27.66	23.09	0.38	23.47	4.19	15%
UA2	28.21	20.69	0.42	21.11	7.10	25%
UA3	27.77	21.05	0.40	21.45	6.32	23%
UA4	27.87	27.53	0.38	27.91	-0.04	0%
UA5	27.79	27.47	0.39	27.86	-0.07	0%
UA6	27.71	27.39	0.39	27.78	-0.07	0%
UA7	27.49	27.18	0.38	27.56	-0.07	0%
UA8	27.94	27.31	0.39	27.70	0.24	1%
UB1	27.61	27.30	0.38	27.68	-0.07	0%
UB2	27.25	20.59	0.41	21.00	6.25	23%
UB3	25.05	19.95	0.38	20.33	4.72	19%
UB4	25.06	19.88	0.38	20.26	4.80	19%
UB5	27.59	27.06	0.39	27.45	0.14	1%
UB6	27.65	26.91	0.38	27.29	0.36	1%
UB7	27.96	27.48	0.41	27.89	0.07	0%
UB8	27.87	22.50	0.38	22.88	4.99	18%

the update.

Even in a sequential setting, however, our experiments show that independence analysis is generally beneficial. In some cases, the total time needed by independence-based maintenance was slightly longer than the naive approach. However, the added expense of independence analysis is negligible even in comparison to the time needed to re-compute queries on the small, 1.1MB document. Indeed, since the static checking time is fixed, the asymptotic worst-case overhead is zero as the size of the database increases (for queries that take more than constant time).

Conversely, our experiments also show that the potential benefits of static independence checking are substantial (up to 22% for the 1.1MB document), and actually increase (to a maximum of 25% for the 2.3MB document) as the data size increases. In particular, note that almost all of the time savings percentages in Table 4 are slightly higher for the 2.3MB document than for the 1.1MB document. This is again because the costs of query re-evaluation grow in proportion to the size of the data, whereas the cost of static analysis is dependent only on the query and update.

Galax is not the performance leader among XQuery engines; we chose it for its support of the standard. However, for larger documents (e.g. tens or hundreds of megabytes) the overhead of our analysis is negligible compared with the querying times of the faster engines. For example, only two of the 20 XMark queries can be answered in less than 20 milliseconds for a 110MB document by any of the engines measured in the current Qizx[1] benchmarks, .

5. RELATED AND FUTURE WORK

To our knowledge, Raghavachari and Shmueli [22] were the first to study query-update independence problems. They studied conflicts between read, insert and delete operations based on downward XPath expressions, described special cases that are solvable in polynomial time, and proved NP-hardness results for several XPath fragments; however they do not present an implementation or experimental validation. In contrast, we give a sound, but incomplete technique that works for general XQuery queries and updates involving all XPath axes.

There is a growing literature on typechecking for XML queries. Our set-based type system is a simplification of the standard XQuery type system; Colazzo et al. [12] have studied more sophisticated regular expression type systems for XML queries and Cheney [11] extended this approach to a simple XML update language. More recently, Benedikt and Cheney [4] have developed typechecking techniques for W3C XQuery Update Facility 1.0 updates. Besides being intrinsically useful, update type analysis may lead to more accurate techniques for query-update or update-update independence problems.

Static analysis problems besides typechecking have also been studied for XML or object query/update languages. Bierman [7] developed an effect analysis that tracks object-identifier generation side-effects in OQL queries. Benedikt et al. [1, 2] presented offline static analyses for optimizing updates in UpdateX, a precursor to XQuery Update. Marian and Siméon [18] deal with the problem of *projecting* an XML document on a query; this involves statically finding the paths that may be accessed by the query. Our access set analysis is similar to projection analysis. However, for our independence analysis we need to consider changes that may insert, delete, replace or rename nodes, whereas projection analysis only considers deletions. Benzaken et al. [6] investigated schema-based projection of queries, including a more sophisticated type system for XPath steps that may also be useful in improving the accuracy of our independence analysis.

The closest work to ours is that of Ghelli, Rose and Siméon [16]. They study the commutativity problem for a different update language, where side-effects can be applied immediately in the course of evaluation. The algorithms of [16] take an approach similar to that of [18]: they find the paths associated with nodes accessed by the queries in the input, along with those paths modified by the update – a sufficient condition for commutativity is that these sets do not overlap. In contrast, while our work adapts some of the ideas of [16] to the independence analysis setting, it is based on schema information rather than path information. Combining schema-based and path-based techniques is an interesting direction

[1] `http://www.xmlmind.com/qizx/speed.html`

for future work.

Incremental view maintenance of XQuery expressions is considered in [15, 13]. Queries are converted into an algebra, and as queries are evaluated some metadata is recorded. Subsequent update expressions are propagated using the metadata to avoid unnecessary recomputation. These works deal with a simpler update language, with no control structures; they also do not account for the presence of schemas. Björklund et al. [8] also investigate incremental maintenance for Boolean XPath queries. Our work complements, but does not replace efficient incremental view maintenance. It may be interesting to compare static independence analysis with efficient incremental view maintenance techniques or to develop combined static and dynamic techniques.

6. CONCLUSIONS

Query-update independence analysis is useful for avoiding view maintenance or recomputation costs. In this paper we have given the (to our knowledge) first *schema-based* query-update independence analysis. We have also implemented and experimentally validated our approach, and shown that it offers significant performance improvements for an online view maintenance scenario based on typical XMark and XPathMark queries and updates using Galax. Even for a relatively small 1.1MB XMark document, we found that the cost of independence analysis is negligible and can lead to significant ($20\% - 25\%$) savings from avoiding query recomputation. The costs of query and update evaluation typically grow in proportion to the size of the data, whereas the costs of static analysis do not, so query-update independence analysis is inherently scalable.

We have identified a number of possible directions for future work. While our analysis already provides significant benefits, there is much room for improvement of features such as descendant, ancestor and sibling axes. Accuracy might be improved further by tracking more detailed static approximations of the behavior of the queries and updates. We also believe it would be worthwhile to combine our approach with complementary path-based analyses or incremental view maintenance techniques. Finally, it would be of interest to test our approach using more realistic benchmarks involving schemas, queries and updates gathered from real-world settings.

Acknowledgment. We would like to thank Avinash Vyas and Dinesh Venkataramanaidu for comments on an early draft of this work. Michael Benedikt is supported in part by EPSRC EP/G004021/1 (the Engineering and Physical Sciences Research Council, UK). James Cheney is supported by a Royal Society University Research Fellowship and EPSRC grant EP/F028288/1.

7. REFERENCES

[1] Michael Benedikt, Angela Bonifati, Sergio Flesca, and Avinash Vyas. Adding updates to XQuery: Semantics, optimization, and static analysis. In Daniela Florescu and Hamid Pirahesh, editors, *XIME-P*, 2005.

[2] Michael Benedikt, Angela Bonifati, Sergio Flesca, and Avinash Vyas. Verification of tree updates for optimization. In *CAV*, 2005.

[3] Michael Benedikt and James Cheney. Schema-based independence analysis for XML updates. `http://web.comlab.ox.ac.uk/people/Michael.Benedikt/papers/tr.pdf`.

[4] Michael Benedikt and James Cheney. Types, effects, and schema evolution for XML Updates. In *DBPL*, 2009.

[5] Michael Benedikt and Christoph Koch. Interpreting tree-to-tree queries. In *ICALP*, 2006.

[6] Véronique Benzaken, Giuseppe Castagna, Dario Colazzo, and Kim Nguyên. Type-based xml projection. In *VLDB*, pages 271–282. VLDB Endowment, 2006.

[7] G. M. Bierman. Formal semantics and analysis of object queries. In *SIGMOD*, 2003.

[8] Henrik Björklund, Wouter Gelade, Marcel Marquardt, and Wim Martens. Incremental xpath evaluation. In Ronald Fagin, editor, *ICDT*, volume 361 of *ACM International Conference Proceeding Series*, pages 162–173. ACM, 2009.

[9] Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. XQuery 1.0: An XML query language. W3C Recommendation, January 2007. `http://www.w3.org/TR/xquery`.

[10] Don Chamberlin and Jonathan Robie. XQuery update facility 1.0. W3C Candidate Recommendation, August 2008. `http://www.w3.org/TR/xquery-update-10/`.

[11] James Cheney. FLUX: FunctionaL Updates for XML. In *ICFP*, 2008.

[12] Dario Colazzo, Giorgio Ghelli, Paolo Manghi, and Carlo Sartiani. Static analysis for path correctness of XML queries. *J. Funct. Program.*, 16(4-5):621–661, 2006.

[13] Katica Dimitrova, Maged El-Sayed, and Elke A. Rundensteiner. Order-sensitive view maintenance of materialized XQuery views. In *ER*, 2003.

[14] Denise Draper, Peter Fankhauser, Mary Fernández, Ashok Malhotra, Kristoffer Rose, Michael Rys, Jérôme Siméon, and Philip Wadler. XQuery 1.0 and XPath 2.0 formal semantics. W3C Recommendation, January 2007. `http://www.w3.org/TR/xquery-semantics/`.

[15] J. Nathan Foster, Ravi Konuru, Jérôme Siméon, and Lionel Villard. An algebraic approach to view maintenance for XQuery. In *PLAN-X*, 2008.

[16] Giorgio Ghelli, Kristoffer Rose, and Jérôme Siméon. Commutativity analysis for XML updates. *ACM Trans. Database Syst.*, 33(4):1–47, 2008.

[17] Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. Regular expression types for XML. *ACM Trans. Program. Lang. Syst.*, 27(1):46–90, 2005.

[18] Amélie Marian and Jérôme Siméon. Projecting XML documents. In *VLDB*, 2003.

[19] M.Francechet. XPathMark: an XPath benchmark for XMark generated data. In *XSYM*, 2005.

[20] Makoto Murata. "Relax". http://www.xml.gr.jp/relax/.

[21] Yannis Papakonstantinou and Victor Vianu. Type inference for views of semistructured data. In *PODS*, 2000.

[22] Mukund Raghavachari and Oded Shmueli. Conflicting XML updates. In *EDBT*, 2006.

[23] Mukund Raghavachari and Oded Shmueli. Efficient revalidation of XML documents. *IEEE Trans. on Knowl. and Data Eng.*, 19(4):554–567, 2007.

[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] Thomas Schwentick. "Automata for XML – A Survey". *Journal of Computer and Systems Science*, 73:289–315, 2007.

[26] Gargi Sur, Joachim Hammer, and Jérôme Siméon. UpdateX - an XQuery-based language for processing updates in XML. In *PLAN-X*, 2004.