Destabilizers and Independence of XML Updates

Michael Benedikt
Computing Laboratory, Oxford University

James Cheney
LFCS, University of Edinburgh

ABSTRACT

Independence analysis is the problem of determining whether an update affects the result of a query, e.g. a constraint or materialized view. We develop a new, modular framework for static independence analysis that decomposes the problem into two orthogonal subproblems: approximating the *destabilizer*, that is, a finite representation of the set of updates that can change the result of the query, and testing whether the update and destabilizer overlap via an intersection analysis. Focusing on XML queries as the view language and the XQuery Update Facility as the update language, we present a syntactic query rewriting algorithm for translating queries to destabilizers, and show that intersection checking can be reduced to satisfiability problems for which efficient checkers already exist. We present an implementation based on an expressive tree satisfiability checker and a Satisfiability Modulo Order package, and give experiments confirming that the resulting analysis is both fast and effective.

1. INTRODUCTION

View maintenance – that is, recomputing integrity constraints or derived data as base data is updated – is a classical and well-studied problem for relational databases. It is much less well-understood for other data models, particularly XML databases. View maintenance is just as important for XML as for relational databases and is an active area of study [2, 27, 26, 7, 12].

View recomputation is avoidable if the query and update are statically *independent*, that is, there is no possible source data for which the update can affect the result of the query. In this common case, full or incremental recomputation might require time proportional to the size of the data just to determine that no work needs to be done. Query and update expressions are usually fairly small, so even a relatively expensive static analysis may be competitive with full recomputation or incremental techniques if it offers the hope of avoiding dynamic recomputation cost proportional to data size.

Static independence analysis of queries and updates has been explored both in the context of relational databases [8, 23] and XML [2, 27, 26]. For SQL, a crude independence analysis often suffices, checking whether the relation and field names in the update are disjoint from those in the query. For XML there is no such "obvious analysis" – queries and updates generally run over the same document tree, and variables may not have a syntactically transparent scope. Prior techniques either require a schema [2] or apply only to downward fragments of XPath [27, 26].

Proceedings of the VLDB Endowment, Vol. 3, No. 1

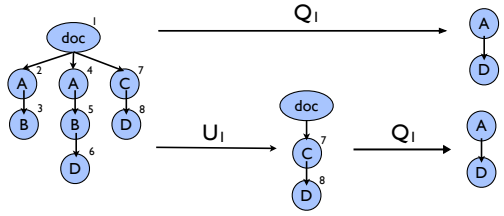

Figure 1: Document, view and an independent update

We develop a new, modular framework for static independence analysis that decomposes the problem into two orthogonal subproblems: (1) identifying a *destabilizer*, that is, a finite representation of a set of run-time updates that includes all updates that could change the result of the query, and (2) statically determining whether the destabilizer and the update are disjoint. Destabilizers are a generalization of ideas such as statically approximating the set of *accessed nodes* found in some prior work [24, 6, 18]. However, the destabilizer framework provides increased flexibility and modularity, easing exploration of the tradeoff between precision and performance. Our approach, in principle, is applicable to independence analysis in any data model and query/update language; here, we apply it to XQuery/XQuery Updates.

To illustrate our approach, consider the following examples:

Example 1. Consider the situation depicted in Figure 1, where we have a view

$$Q_1 = \texttt{for } \$x \in \$doc/C \texttt{ return } \langle A\rangle \$x/D \langle /A\rangle \ .$$

Consider the set of all atomic update sequences that could affect the result of Q_1. Any set containing this collection of sequences is called a *subtree value destabilizer* for Q_1. In this example, the set of all update sequences that include an update to $\{1, 2, 4, 7, 8\}$ is a value destabilizer. We can be even more precise and say that for an update sequence to change the value of Q_1 it must contain either a deletion or replace on one of $\{1, 7, 8\}$, a rename on one of $\{2, 4, 7, 8\}$, or an insert into one of $\{1, 7, 8\}$, etc. Our analysis can determine that the update expression $U_1 = \texttt{delete } \$doc/A$ can never perform any of the updates above, so it is statically independent of Q_1, as suggested in Figure 1. $\diamond$

Example 2. Consider the XPath query $Q_2 = \$doc/A/B$ where variable $\$doc$ again refers to the root of the tree shown in Figure 1.

We view Q_2 as a Boolean constraint asserting that there must always be a node matching path $/A/B$. To ensure the constraint is maintained, we would like to statically determine whether an update can change the result of Q_2 from nonempty (true) to empty (false). A *negative Boolean destabilizer* is a set including all update sequences that can invalidate Q_2. In this example, the set of all sequences containing updates to one of $\{1, 2, 3, 4, 5\}$ is a negative Boolean destabilizer. If we take the update operation into account, then for an update sequence to change Q_2 to false, it must contain a delete or replace to one of $\{1, 2, 3, 4, 5\}$ or a rename to one of $\{2, 3, 4, 5\}$: insert operations can never play a role in making the result of Q_2 empty. Our analysis can determine that update $U_2 = \texttt{insert}\ \langle A\rangle\langle/A\rangle\ \texttt{into}\ \$doc$ can never invalidate Q_2, since it can only perform inserts. $\diamond$

Example 3. Consider a query and update

$$\begin{aligned} Q_3 &= \texttt{if}\ \$doc/A\ \texttt{then}\ \langle E\rangle\langle/E\rangle\ \texttt{else}\ () \\ U_3 &= \texttt{if}\ \$doc/A\ \texttt{then}\ ()\ \texttt{else}\ \texttt{delete}\ \$doc/C \end{aligned}$$

Clearly, for the example document in Figure 1, Q_3 and U_3 are independent; in fact they are independent for any input, since U_3 cannot change the Boolean value of the test $\$doc/\texttt{child} :: A$. Our analysis can determine this by describing a destabilizer and then showing that the updates generated by U_3 can not intersect the destabilizer via reduction to a decidable tree logic. $\diamond$

The destabilizer framework thus works in several steps: we symbolically describe the sequences of updates that can change the query result, and also represent the sequences that can be generated by the update expression, and then use satisfiability testing to see that they cannot overlap. Though this is a clean and deceptively simple approach, there are nontrivial obstacles that must be overcome to turn it into a working, useful analysis. Calculating exact (static or dynamic) destabilizers is not feasible for our core XML query language; instead, we present a query-rewriting technique that provides a useful, sound approximation. Given an approximate destabilizer, it then suffices to establish that none of the updates that U can perform are in Q's destabilizer. This can be reduced to solving intersection problems for XPath or XQuery expressions, but again, this is nontrivial in practice: for example, intersection testing for even simple path expressions is NP-hard [21].

Contributions. The main contributions of this work are:

- We introduce a framework for independence analysis based on the notion of destabilizer, which is, in principle, applicable to any data model and query/update language.
- We introduce two varieties of destabilizer for XQuery, called *generic* and *operation(op) -sensitive*. We provide algorithms that efficiently rewrite queries to approximate destabilizers.
- We develop novel techniques for efficiently solving the disjointness problems that arise through destabilizers, using a combination of heuristics and reduction to SMT solvers.
- We present experiments showing that independence analysis using destabilizers is fast and effective.

Organization: We begin in Section 2 by reviewing the XML query and update languages considered here. In Section 3 we give our destabilizer representations and algorithms. In Section 4 we apply these to independence analysis. Section 5 presents our experiments. Section 6 reviews related work, and Section 7 Proofs and many routine details are omitted or placed in appendices.

2. PRELIMINARIES

As is conventional in most work on XML databases (e.g. [24, 6, 26, 11, 16, 2, 18, 27]) we employ a simplified XML data model and query language, focusing on tree structure and ignoring attributes and some other features. Generalizing to handle these features is straightforward. Throughout this paper, we use the symbol $=$ for mathematical equality, not one of XQuery's equality operations.

A *store* σ (sometimes called an *instance*, *document* or *database*) is an ordered labeled forest, whose nodes $\texttt{l}, \texttt{l}', \texttt{m}$ (also referred to as *locations*) are either element nodes or text nodes. An element node has a label, while a text node has an associated string; text nodes have no children. In addition to its label or string, each node has an identifier, which is assumed to be unique within a store. We often write location sequences as $\texttt{L}, \texttt{L}', \texttt{L}''$.

Queries. We will use a simple XQuery-like core language, denoted XQ:

$$\begin{aligned} \texttt{q} &::= s \mid () \mid \texttt{q}, \texttt{q}' \mid \langle a\rangle\texttt{q}\langle/a\rangle \mid \$x \mid \$x/step \\ &\mid \texttt{if q then q}_1\ \texttt{else q}_2 \mid \texttt{for}\ \$x \in \texttt{q}\ \texttt{return q}' \\ step &::= ax :: \phi \qquad \phi ::= a \mid * \mid \texttt{text}() \\ ax &::= \texttt{self} \mid \texttt{child} \mid \texttt{parent} \mid \texttt{foll\_sib} \mid \texttt{prec\_sib} \\ &\mid \texttt{desc} \mid \texttt{desc\_or\_self} \mid \texttt{anc} \mid \texttt{anc\_or\_self} \end{aligned}$$

XQuery expressions include variables $\$x, \$y, \$z, \ldots \in Var$. The expression s is a test literal (constructing a text node from a string), expression $()$ denotes the empty sequence; the expression $\langle a\rangle\texttt{q}\langle/a\rangle$ builds an XML tree with root a and content $\texttt{q}$; and $\texttt{q}, \texttt{q}'$ concatenates sequences of XML values. Conditionals $\texttt{if q then q}_1\ \texttt{else q}_2$ branch depending on whether their first argument is nonempty. The iteration expression $\texttt{for}\ \$x \in \texttt{q}\ \texttt{return q}'$ evaluates $\texttt{q}$, and for each node $\texttt{l}$ in the result evaluates $\texttt{q}'$ with $\$x$ bound to $\texttt{l}$, concatenating the results in order. Without loss of generality, we omit XQuery's `let`-binding construct; since we consider a recursion-free language, all occurrences of `let` can be inlined.

The expression $\$x/ax::\phi$ performs an XPath step starting from $\$x$, where ax is one of the standard XPath axes and ϕ is an XPath node test. The XPath axes allow navigation in the document tree. The `self` axis is the identity, and the `child` and `parent` axes link nodes to their parents and children. The `desc` and `anc` axes are the transitive closures of `child` and `parent`, and the `desc_or_self` and `anc_or_self` are their reflexive, transitive closures. The axes `foll_sib` and `prec_sib` select the following or preceding siblings of a node, respectively; other axes, such as `following`, can be built up from these using composition. The semantics of XPath axes is standard; see for example [17, 13]. The expressions $\$x/\texttt{child} :: \phi$ and $\$x/\texttt{desc\_or\_self} :: */\texttt{child} :: \phi$ are often abbreviated $\$x/\phi$ and $\$x//\phi$, respectively.

Updates. The W3C XQuery Update proposal [9] defines *atomic updates* that describe concrete changes to XML trees::

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

Here, the symbols $\texttt{d}$ stand for the qualifiers on inserts (before, after, into, as first into and as last into, respectively). Atomic updates describe concrete changes to the tree using node identifiers; for example $\texttt{delete}(n)$ or $\texttt{ins}(\texttt{L}, \downarrow, n)$ where n is a target node and $\texttt{L}$ is a node sequence addressing data that will be inserted into the child list of a target node. Atomic updates can be classified by the following *operations*:

$$op ::= \texttt{insert}(\texttt{d}) \mid \texttt{delete} \mid \texttt{replace} \mid \texttt{rename}$$

where the insert, delete, replace and rename operations match the corresponding atomic updates in the obvious way.

The W3C proposal also defines high-level *updating expressions* $\texttt{u}$ that declaratively describe updates without referring to node identifiers. Update expressions include forms such as $\texttt{delete q}_0$ and

`insert q into` $\mathtt{q}_0$ that evaluate to sequences of atomic updates. In our implementation, we handle a core language for XQuery Updates similar to that formalized in [3]. Due to space limits, in the body of the paper we discuss only updates targeting XPath queries. More details are presented in Appendix A.

Semantics. Due to space limits, we cannot give the full formal semantics of queries and updates. We outline a simplified semantics that illustrates the basic ideas. Our results hold for the full query and update language described in Appendix A. The semantics of a query is given by a relation $\sigma, \gamma \models \mathtt{q} \Rightarrow \sigma', \mathtt{L}$ where σ is an input store, γ is an environment mapping variables to nodes in σ, σ' is an output store, and L is a list of nodes. Our semantics explicitly models nondeterministic creation of element nodes. Node construction queries allocate new trees in the store, whereas all other query constructs leave the contents of the store alone; hence, σ is always a subforest of σ'.

The semantics of atomic updates to XML trees is formalized in several places, e.g. [3], and we include a formalization in Appendix A. One subtlety relevant to our work is that we model a delete operation as detaching a node from its parent, creating a disconnected store; downward axis steps from the detached node can be evaluated as usual after the deletion occurs. We can easily modify our algorithms to account for alternative semantics for deletion.

We assume (purely for convenience) that each store has a distinguished root $root(\sigma)$, and we define a *document query* or *document update* to be a query or update with a single distinguished free variable $\$doc$ which is bound to the root of the store.

We also define a sublanguage of XQ called $SelXQ$, consisting of *selection queries* that exclude the element node construction operation $\langle a \rangle q \langle /a \rangle$. This restriction implies that queries always return nodes already present in the input and never construct new nodes. We can interpret document selection queries as maps from stores to sets of nodes as follows:

$$\mathtt{Sel}[\![q]\!]\sigma = \{\mathtt{l} \in \mathtt{L} \mid \sigma, [\$doc := root(\sigma)] \models \mathtt{q} \Rightarrow \sigma, \mathtt{L}\}$$

Equivalence. As discussed in the introduction, we are interested in several different interpretations of the results of a query, including Boolean, node, and subtree equivalence. We capture these notions precisely in the following definitions.

DEFINITION 1 (QUERY OUTPUT EQUIVALENCES). *Given a document query* q *and stores* $\sigma, \hat{\sigma}$*, let* $\sigma_1, \mathtt{L}_1$ *be the result of evaluating* q *on* σ *and* $\sigma_2, \mathtt{L}_2$ *be the result of evaluating* q *on* $\hat{\sigma}$*. Then we say that* $\sigma_1, \mathtt{L}_1$ *and* $\sigma_2, \mathtt{L}_2$ *are* Boolean equivalent *(*$(\sigma_1, \mathtt{L}_1) \cong_{bool} (\sigma_2, \mathtt{L}_2)$*) if either both* $\mathtt{L}_1$ *and* $\mathtt{L}_2$ *are empty or both are nonempty. Similarly, we say that* $\sigma_1, \mathtt{L}_1$ *and* $\sigma_2, \mathtt{L}_2$ *are* subtree value equivalent *(*$\sigma_1, \mathtt{L}_1 \cong_{ns} \sigma_2, \mathtt{L}_2$*) if* $\mathtt{L}_1$ *and* $\mathtt{L}_2$ *are isomorphic (as hedges, that is, sequences of trees).*

Finally, we say that $\sigma_1, \mathtt{L}_1$ *and* $\sigma_2, \mathtt{L}_2$ *are* node equivalent *if* $\mathtt{L}_1 = \mathtt{l}_1, \ldots, \mathtt{l}_n$ $\mathtt{L}_2 = \hat{\mathtt{n}}_1, \ldots, \hat{\mathtt{n}}_n$ *such that*

- *for every* $i \leq n$ *if* $\mathtt{l}_i$ *is in* σ *or* $\hat{\mathtt{n}}_i$ *is in* $\hat{\sigma}$ *then* $\mathtt{l}_i = \hat{\mathtt{n}}_i$
- *for every* $i \leq n$ *if* $\mathtt{l}_i$ *is not in* σ *then the isomorphism type of* $\mathtt{l}_i$ *within its connected component in* σ_1 *is the same as the isomorphism type of* $\hat{\mathtt{n}}_i$ *within its connected component in* σ_2*.*

We write $\sigma_1, \mathtt{L}_1 \cong_{node} \sigma_2, \mathtt{L}_2$ *in this case, although the equivalence depends on the original stores* $\sigma, \hat{\sigma}$ *as well.*

In the above definition, node equivalence says that the resulting lists return exactly the same nodes in the original store, and the new nodes that are created via node construction are isomorphic. It is easy to see that if $\sigma_1, \mathtt{L}_1 \cong_{node} \sigma_2, \mathtt{L}_2$, then the same holds for any other $\mathtt{L}_1'$ that can result from q on σ, since these will only vary on the identities of nodes outside of the input store. The term "isomorphism type" is used above with its standard meaning in model theory, regarding the store as a finite structure.

Independence. Informally, independence means that an update cannot change the result of a query. However, since both queries and updates can be nondeterministic, we need to define this carefully. We formalize this as follows.

DEFINITION 2. *Let* $\equiv$ *be one of* $\cong_{ns}, \cong_{node}, \cong_{bool}$*. Given an update* u *and a query* q*, we say* q *is* independent of u *(modulo* $\equiv$*) on input* σ_1 *provided that for all* σ_2 *that can result from applying* u *to* σ_1*, the possible behaviors of* q *on* σ_1 *are the same as those on* σ_2 *(modulo* $\equiv$*). If* q *and* u *are independent on all input stores, then we say they are* statically independent*.*

3. DESTABILIZERS FOR XML QUERIES

As discussed in the previous section, we can view the result of a query in a number of ways. We may only care about the Boolean value of a query, for example, if the query is only used as a Boolean test in a conditional, or as a constraint. We may care about the sequence of nodes produced (up to isomorphism). Or we may care about the complete XML tree values produced.

DEFINITION 3. *A* destabilizer *is a collection of update sequences containing all those sequences that change the result of a query in one of the above senses. More specifically, a* Boolean destabilizer *for* q *on* σ *is a collection of atomic update sequences that includes every update sequence that changes the result of* q *on* σ *modulo* $\cong_{bool}$*. A positive (negative) destabilizer is a collection containing every update sequence changing the result from false to true (respectively from true to false). A* node destabilizer *is a collection of update sequences containing every sequence changing the result of* q *on* σ *modulo* $\cong_{node}$*, while a* subtree value destabilizer *contains every sequence changing the result modulo* $\cong_{ns}$*.*

In this paper we will be most concerned with value destabilizers, but Boolean and node destabilizers are also of practical interest. Moreover, to calculate useful value destabilizers we will need to calculate positive, negative and node destabilizers as well.

3.1 Statically approximating destabilizers

For a given update sequence ω and input σ it is possible to determine dynamically whether ω destabilizes q on σ, by applying ω and rerunning q. However, this cost is exactly what we wish to avoid by a static independence analysis. It is not feasible to deal with destabilizers statically as concrete sets of sequences. Instead, we want to work with destabilizers indirectly, at a symbolic level.

Selection queries are a natural way to represent sets of nodes statically; a set of nodes S in turn can be considered as a representation of the collection of update sequences that modify a node in S. For a set of nodes S, let $\mathtt{SeqTargets}(S)$ be the set of update sequences that contain some update with target in S. A *static value destabilizer* for q is a selection query δ such that for each σ, $\mathtt{SeqTargets}(\mathtt{Sel}[\![\delta]\!]\sigma)$ is a value destabilizer, and analogously define positive (respectively negative, node, Boolean) static destabilizers. Unwinding the definition, this says that the query returns a set of nodes S such that every update sequence that destabilizes q on σ modifies a node in S.

The generic static destabilizer defined above is frequently a significant over-approximation of the runtime destabilizer. For example, a query may be sensitive to update sequences that can only affect a node via a particular update operation. Precision can be increased substantially using finer-grained representations of collections of update sequences. We exhibit one such refinement, using

$$
\begin{array}{rcl}
\Delta(()) & = & () \\
\Delta(\$x) & = & \Delta(\$x/\texttt{self} :: *) \\
\Delta(\mathtt{q}, \mathtt{q}') & = & \Delta(\mathtt{q}), \Delta(\mathtt{q}') \\
\Delta^{\mathtt{b}+}(\langle A\rangle \mathtt{q} \langle /A\rangle) & = & () \\
\Delta^{\mathtt{b}-}(\langle A\rangle \mathtt{q} \langle /A\rangle) & = & () \\
\Delta^{\mathtt{n}}(\langle A\rangle \mathtt{q} \langle /A\rangle) & = & \Delta^{\mathtt{v}}(\mathtt{q}) \\
\Delta^{\mathtt{v}}(\langle A\rangle \mathtt{q} \langle /A\rangle) & = & \Delta^{\mathtt{v}}(\mathtt{q}) \\
\Delta(\texttt{if q then q' else q''}) & = & \texttt{if q then } (\Delta^{\mathtt{b}-}(\mathtt{q}), \Delta(\mathtt{q}')) \\
 & & \texttt{ else } (\Delta^{\mathtt{b}+}(\mathtt{q}), \Delta(\mathtt{q}'')) \\
\Delta^{\mathtt{b}+}(\texttt{for } \$x \in \mathtt{q} \texttt{ return } \mathtt{q}') & = & \Delta^{\texttt{for}}(\mathtt{q}, \$x, \texttt{if } \neg\mathtt{q}' \texttt{ then } \Delta^{\mathtt{b}+}(\mathtt{q}')) \\
\Delta^{\mathtt{b}-}(\texttt{for } \$x \in \mathtt{q} \texttt{ return } \mathtt{q}') & = & \Delta^{\texttt{for}}(\mathtt{q}, \$x, \texttt{if } \mathtt{q}' \texttt{ then } \Delta^{\mathtt{b}-}(\mathtt{q}')) \\
\Delta^{\mathtt{n}}(\texttt{for } \$x \in \mathtt{q} \texttt{ return } \mathtt{q}') & = & \Delta^{\texttt{for}}(\mathtt{q}, \$x, \Delta^{\mathtt{n}}(\mathtt{q}')) \\
\Delta^{\mathtt{v}}(\texttt{for } \$x \in \mathtt{q} \texttt{ return } \mathtt{q}') & = & \Delta^{\texttt{for}}(\mathtt{q}, \$x, \Delta^{\mathtt{v}}(\mathtt{q}')) \\
\Delta^{\texttt{for}}(\mathtt{q}, \$x, \mathtt{q}') & = & \Delta^{\mathtt{n}}(\mathtt{q}), \texttt{for } \$x \in \mathtt{q} \texttt{ return } \mathtt{q}'
\end{array}
$$

Figure 2: Query rewriting for destabilizers (non-axis steps)

operations to classify updates (as defined in Section 2). We then statically represent the update sequences that include an update with a particular target and a particular operation. For a set of nodes S, let $\texttt{SeqTargets}_{op}(S)$ be the set of update sequences that contain some update with operation op and a target in S. An indexed family of queries $(\delta_{op})_{op \in \texttt{Ops}}$ is a *static value op-destabilizer* if for each σ, $\bigcup_{op} \texttt{SeqTargets}_{op}(\texttt{Sel}[\![\delta]\!]\sigma)$ is a value destabilizer, and we define positive, negative, node, and Boolean op-destabilizers analogously. Equivalently, every update sequence that destabilizes $\mathtt{q}$ on σ must contain an update of operation op that modifies a node returned by δ_{op}.

In developing a static destabilizer, we have a trade-off between the precision (how close it is to the dynamic analog) and the complexity of analyzing the resulting query. Ideally, one would calculate a query that always returns the best possible approximation of the runtime destabilizer. Formally, a *pointwise minimal* static destabilizer (or respectively op-destabilizer) is a query that returns on any document a minimal collection of nodes having the property that every update sequence that changes the query result contains an update (respectively an update of operation op) with the given node as target. However, the following result shows that minimal static boolean destabilizers can not be calculated efficiently. The proof is in Appendix B.

THEOREM 1. *There is no elementary time algorithm for constructing a pointwise minimal static destabilizer.*

On the other extreme, there is always a trivial, non-minimal destabilizer including *all* update sequences to the store σ. Of course, this is useless for static independence analysis.

Our approach is to compute a static destabilizer by a simple inductive rewriting algorithm. This is algorithmically feasible, while providing high precision in practice. We define four functions $\Delta^{\mathtt{b}+}$, $\Delta^{\mathtt{b}-}$, $\Delta^{\mathtt{n}}$ and $\Delta^{\mathtt{v}}$ simultaneously by mutual recursion on the structure of expressions, which will return static positive, negative, node, and value destabilizers. We also define functions $\Delta^{\mathtt{b}+}_{op}$, $\Delta^{\mathtt{b}-}_{op}$, $\Delta^{\mathtt{n}}_{op}$ and $\Delta^{\mathtt{v}}_{op}$ for each update operation op; the vector of queries over each op will give a op-destabilizer.

Figure 2 shows the definitions of $\Delta^{\mathtt{b}+}$, $\Delta^{\mathtt{b}-}$, $\Delta^{\mathtt{n}}$ and $\Delta^{\mathtt{v}}$ for all query constructs except for $\$x/step$. We use Δ to denote any one of $\Delta^{\mathtt{b}+}$, $\Delta^{\mathtt{b}-}$, $\Delta^{\mathtt{n}}$ and $\Delta^{\mathtt{v}}$. In the case of these query constructs, the very same rules can be used for each element of the static op-destabilizer (e.g. for every $op \in \texttt{Ops}$, $\Delta^{\mathtt{b}+}_{op}(\mathtt{q}, \mathtt{q}') = \Delta^{\mathtt{b}+}_{op}(\mathtt{q}), \Delta^{\mathtt{b}+}_{op}(\mathtt{q}')$, and similarly for the other complex constructs).

We now discuss the rules in Figure 2 starting from the top. The case for () is obvious: no update sequence can destabilize the query (). Variables $\$x$ are treated the same as $\$x/\texttt{self} :: *$ steps. For

$$
\begin{array}{rcl}
\Delta^{\mathtt{b}+}_{*}(\$x/\texttt{self} :: *) & = & () \\
\Delta^{\mathtt{b}+}_{*}(\$x/\texttt{self} :: A) & = & \$x \\
\Delta^{\mathtt{b}+}_{*}(\$x/\texttt{child} :: \phi) & = & (\$x, \$x/\texttt{child} :: *) \\
\Delta^{\mathtt{b}+}_{*}(\$x/\texttt{parent} :: \phi) & = & (\$x, \$x/\texttt{parent} :: *) \\
\Delta^{\mathtt{b}+}_{*}(\$x/\texttt{desc} :: \phi) & = & (\$x/\texttt{desc\_or\_self} :: *) \\
\Delta^{\mathtt{b}+}_{*}(\$x/\texttt{anc} :: \phi) & = & (\$x/\texttt{anc} :: *) \\
\Delta^{\mathtt{b}+}_{*}(\$x/\texttt{foll\_sib} :: \phi) & = & (\$x, \$x/\texttt{parent} :: *, \$x/\texttt{foll\_sib} :: *) \\
\Delta^{\mathtt{b}-}_{*}(\$x/\texttt{self} :: *) & = & () \\
\Delta^{\mathtt{b}-}_{*}(\$x/\texttt{self} :: A) & = & \$x \\
\Delta^{\mathtt{b}-}_{*}(\$x/\texttt{child} :: \phi) & = & (\$x, \$x/\texttt{child} :: \phi) \\
\Delta^{\mathtt{b}-}_{*}(\$x/\texttt{parent} :: \phi) & = & (\$x, \$x/\texttt{parent} :: \phi) \\
\Delta^{\mathtt{b}-}_{*}(\$x/\texttt{desc} :: \phi) & = & (\$x/\texttt{desc\_or\_self} :: *) \\
\Delta^{\mathtt{b}-}_{*}(\$x/\texttt{anc} :: \phi) & = & (\$x/\texttt{anc\_or\_self} :: *) \\
\Delta^{\mathtt{b}-}_{*}(\$x/\texttt{foll\_sib} :: \phi) & = & (\$x/\texttt{parent} :: *, \$x/\texttt{foll\_sib} :: \phi) \\
\Delta^{\mathtt{n}}_{*}(\$x/\texttt{self} :: *) & = & () \\
\Delta^{\mathtt{n}}_{*}(\$x/\texttt{self} :: A) & = & \$x \\
\Delta^{\mathtt{n}}_{*}(\$x/\texttt{child} :: \phi) & = & (\$x, \$x/\texttt{child} :: *) \\
\Delta^{\mathtt{n}}_{*}(\$x/\texttt{parent} :: \phi) & = & (\$x, \$x/\texttt{parent} :: *) \\
\Delta^{\mathtt{n}}_{*}(\$x/\texttt{desc} :: \phi) & = & (\$x/\texttt{desc\_or\_self} :: *) \\
\Delta^{\mathtt{n}}_{*}(\$x/\texttt{anc} :: \phi) & = & (\$x/\texttt{anc\_or\_self} :: *) \\
\Delta^{\mathtt{n}}_{*}(\$x/\texttt{foll\_sib} :: \phi) & = & (\$x, \$x/\texttt{parent} :: *, \$x/\texttt{foll\_sib} :: *) \\
\Delta^{\mathtt{v}}_{*}(\$x/ax :: \texttt{text}()) & = & \Delta^{\mathtt{n}}_{*}(\$x/ax :: \texttt{text}()) \\
\Delta^{\mathtt{v}}_{*}(\$x/step) & = & \Delta^{\mathtt{n}}_{*}(\$x/step)/\texttt{desc\_or\_self} :: *
\end{array}
$$

Figure 3: Generic destabilizers for axis steps. We omit the cases for `desc_or_self`, `anc_or_self`, and `prec_sib`.

sequential composition, the translation is straightforward. In the output of the destabilizer, order is unimportant, since we will only be considering its node selection semantics. This step could thus be read as saying that we over-approximate the update sequences destabilizing $\mathtt{q}, \mathtt{q}'$ as those that destabilize either $\mathtt{q}$ or $\mathtt{q}'$.

Now consider the static destabilizers for node construction queries on lines 4–7 of Figure 2. The rules for the Boolean cases reflect that no updates can change the Boolean result of this query. For node and value equivalence, we conservatively assume that any change that could affect the subtree value of the subquery $\mathtt{q}$ could also affect the result. Note that it would not be sound to translate $\Delta^{\mathtt{n}}(\langle A\rangle \mathtt{q} \langle /A\rangle)$ to () instead. For example, in a query such as $\texttt{for } \$y \in \langle A\rangle \$x \langle /A\rangle \texttt{ return } \$y$, we would thereby lose track of the fact that changes to $\$x$ can affect the result (modulo $\cong_{ns}$). For conditional queries $\texttt{if q then q' else q''}$, we mimic the structure of $\mathtt{q}$. If $\mathtt{q}$ is nonempty, then an update sequence that destabilizes $\mathtt{q}$ must either change $\mathtt{q}$ to be empty, or destabilize $\mathtt{q}'$. Dually, if $\mathtt{q}$ is empty, then to destabilize $\mathtt{q}$, the update must either change $\mathtt{q}$ to be nonempty, or destabilize $\mathtt{q}''$.

For iteration queries $\texttt{for } \$x \in \mathtt{q}_1 \texttt{ return } \mathtt{q}_2$, we introduce a helper function $\Delta^{\texttt{for}}(\mathtt{q}, \$x, \mathtt{q}')$ that conservatively includes any updates that destabilize the node sequence returned by $\mathtt{q}$, and additionally returns any updates satisfying $\texttt{for } \$x \in \mathtt{q} \texttt{ return } \mathtt{q}'$. For the positive destabilizer, the return sequence can also change from empty to nonempty if some update changes the value returned in some iteration from empty to nonempty. Thus, in this case the third argument to $\Delta^{\texttt{for}}$ returns all updates that can positively destabilize an empty result from some iteration. (In Figure 2, we use $\texttt{if } \neg\mathtt{q} \texttt{ then } \mathtt{q}'$ as shorthand for $\texttt{if q then } () \texttt{ else } \mathtt{q}'$.) The negative destabilizer is symmetric. For node and value destabilizers, we return all nodes that may destabilize the node sequence or the value returned by some iteration of the body of the loop, respectively.

We now turn to the static destabilizer functions for axis steps $\$x/step$. We give only the generic versions in Figure 3; we em-

$$
\begin{array}{rcl}
\Delta^{\mathtt{b}+}_{\mathtt{delete}}(\$x/ax :: \phi) & = & () \\
\Delta^{\mathtt{b}+}_{\mathtt{insert}(\downarrow)}(\$x/\mathtt{child} :: \phi) & = & \$x \\
\Delta^{\mathtt{b}+}_{op}(\$x/step) & = & \Delta^{\mathtt{b}+}_{*}(\$x/step) \\
\Delta^{\mathtt{b}-}_{\mathtt{delete}}(\$x/\mathtt{child} :: \phi) & = & \$x/\mathtt{child} :: \phi \\
\Delta^{\mathtt{b}-}_{\mathtt{insert}(\downarrow)}(\$x/\mathtt{child} :: \phi) & = & () \\
\Delta^{\mathtt{b}-}_{op}(\$x/step) & = & \Delta^{\mathtt{b}-}_{*}(\$x/step) \\
\Delta^{\mathtt{n}}_{\mathtt{delete}}(\$x/\mathtt{child} :: \phi) & = & \$x/\mathtt{child} :: \phi \\
\Delta^{\mathtt{n}}_{\mathtt{insert}(\downarrow)}(\$x/\mathtt{child} :: \phi) & = & \$x \\
\Delta^{\mathtt{n}}_{op}(\$x/step) & = & \Delta^{\mathtt{n}}_{*}(\$x/step) \\
\Delta^{\mathtt{v}}_{op}(\$x/ax :: \mathtt{text}()) & = & \Delta^{\mathtt{n}}_{op}(\$x/ax :: \mathtt{text}()) \\
\Delta^{\mathtt{v}}_{op}(\$x/step) & = & \Delta^{\mathtt{n}}_{op}(\$x/step)/\mathtt{desc\_or\_self} :: *
\end{array}
$$

Figure 4: Kind-sensitive destabilizers for axis steps

phasize this by using notation $\Delta^{\mathtt{b}+}_{*}$, $\Delta^{\mathtt{b}-}_{*}$, $\Delta^{\mathtt{n}}_{*}$ and $\Delta^{\mathtt{v}}_{*}$, the $*$ meaning "generic operation". We will not discuss these rules in detail, but each case is easy to verify. Note that for the negative destabilizer rules, we do often take the node label ϕ into account, while for other rules we cannot do this. For example, only updates to nodes matching $\$x/\mathtt{child} :: A$ can negatively destabilize (change from true to false) a step $\$x/\mathtt{child} :: A$, while replacing or renaming any child of $\$x$ could positively destabilize (change from false to true) this step. This is one reason it is helpful to distinguish between positive and negative Boolean destabilizers.

The rules for these functions are highly conservative because we must handle all update operations, as the previous example illustrates. Similarly, the destabilizers for irreflexive steps such as `desc` need to be made reflexive since an insertion targeting $\$x$ would affect the results. Using the op-destabilizer we can improve accuracy, since we are computing different queries for different classes of updates. Figure 4 shows how to do this for $\mathtt{insert}(\downarrow)$ and `delete` operations (additional op-specific rules are in Appendix C). For deletes, we can take advantage of the fact that the result of a child step can only be affected by deleting one of the selected children, and that there is no way to positively destabilize an XPath step by deleting. Analogously, for inserts into the child sequence of a query, we take advantage of the fact that such updates can only destabilize a child step $\$x/\mathtt{child} :: \phi$ by inserting into $\$x$, and that no insert can negatively destabilize such a query. We also give rules for handling `desc` steps. Similar reasoning can be applied to the other axes and other update operators, but we will restrict attention to these common cases.

Note that the algorithm can produce large expressions — in principle, there can be an exponential blowup resulting from the rewriting steps in Figure 2, due to duplication of expressions in conditionals and for-expressions. However, this is not a problem for the typical queries in our benchmarks.

We state correctness for op-destabilizers; the statement of correctness for generic destabilizers is similar.

THEOREM 2. *For any query* $\mathtt{q}$ *and any update operation* $op \in \mathtt{Ops}$*:*

1. $(\Delta^{\mathtt{b}+}_{op}(\mathtt{q}))_{op\in\mathtt{Ops}}$ *is a static positive op-destabilizer for* $\mathtt{q}$*;*
2. $(\Delta^{\mathtt{b}-}_{op}(\mathtt{q}))_{op\in\mathtt{Ops}}$ *is a static negative op-destabilizer for* $\mathtt{q}$*;*
3. $(\Delta^{\mathtt{n}}_{op}(\mathtt{q}))_{op\in\mathtt{Ops}}$ *is a static node op-destabilizer for* $\mathtt{q}$*; and*
4. $(\Delta^{\mathtt{v}}_{op}(\mathtt{q}))_{op\in\mathtt{Ops}}$ *is a static value op-destabilizer for* $\mathtt{q}$*.*

Finally, recalling examples 1–3 from the introduction, we note that the dynamic destabilizers discussed there are obtained by evaluating the static destabilizers in this section on the example data shown in Fig. 1. For example, the (simplified) subtree-value delete-destabilizer of Q_1 is $(\$doc, \$doc/C, \$doc/C/D/\mathtt{desc\_or\_self} :: *)$, which returns nodes $\{1, 7, 8\}$ when run on the document in Figure 1.

4. INDEPENDENCE ANALYSIS

In this section, we show how to use destabilizers for static independence analysis, by reducing the problem to disjointness analysis, or the problem of determining whether two selection queries can ever select a node in common.

DEFINITION 4. *Let* $\mathtt{q}_1$ *and* $\mathtt{q}_2$ *be selection queries. We say* $\mathtt{q}_1$ *and* $\mathtt{q}_2$ overlap *if there exists an input store* σ *such that* $\mathtt{q}_1$ *and* $\mathtt{q}_2$ *select a node in common when evaluated on* σ*. Conversely,* $\mathtt{q}_1$ *and* $\mathtt{q}_2$ *are* disjoint *if they do not overlap.*

We employ the notation $\mathtt{Targ}_{op}(\mathtt{u})$ to denote the selection query that identifies all targets of atomic updates of operation op that can be produced by $\mathtt{u}$. We also write $\mathtt{Targ}_{*}(\mathtt{u})$ for the query selecting all targets of any kind of update performed by $\mathtt{u}$. The (routine) definition of this function is in Appendix C. We state the main correctness result for our independence analysis for arbitrary operations; the correctness property of the generic analysis is similar:

THEOREM 3. *Query* $\mathtt{q}$ *and update expression* $\mathtt{u}$ *are statically independent*

1. *modulo* $\cong_{bool}$ *if for each* $op \in \mathtt{Ops}$*, the queries* $\mathtt{Targ}_{op}(\mathtt{u})$ *and* $\Delta^{\mathtt{b}+}_{k}(\mathtt{q}), \Delta^{\mathtt{b}-}_{k}(\mathtt{q})$ *are disjoint.*
2. *modulo* $\cong_{node}$ *if for each* $op \in \mathtt{Ops}$*, the queries* $\mathtt{Targ}_{op}(\mathtt{u})$ *and* $\Delta^{\mathtt{n}}_{op}(\mathtt{q})$ *are disjoint.*
3. *modulo* $\cong_{ns}$ *if for each* $op \in \mathtt{Ops}$*, the queries* $\mathtt{Targ}_{op}(\mathtt{u})$ *and* $\Delta^{\mathtt{v}}_{op}(\mathtt{q})$ *are disjoint.*

4.1 Disjointness Analysis

Selection queries can be translated to first-order formulas over the child, descendant and sibling relations in a standard way (see e.g. [5]). Thus, disjointness analysis for $SelXQ$ reduces to satisfiability for first-order formulas over trees $FO(Tree)$, involving the child, descendant, sibling and node test predicates. However it follows from results of [29] that there is a non-elementary lower bound on the complexity of satisfiability.

We investigated a number of practical ways to solve the overlap problems arising in typical XPath or XQuery queries, including both exact algorithms and approximate heuristics that offer better performance on typical examples. Here, we present general techniques that can prove static disjointness for arbitrary pairs of selection queries. Additional heuristic techniques and further details are presented in Appendix C. We should stress here that in our implementation, we use the fast, approximate techniques first, and only if these fail try progressively stronger but more expensive methods.

Exact approaches. The most direct, and expensive, approaches we considered employ off-the-shelf techniques for checking satisfiability of tree logics. We experimented with solving these problems by translation to monadic second order logic over trees ($MSO(Tree)$), which can be solved by the `mona` tool [16, 22]. This approach can decide satisfiability for arbitrary first-order formulas over trees, thus can exactly determine whether two selection queries overlap. We also experimented with an XPath solver based on μ-calculus by Genevès et al. [17].

Reduction to SMT solving. We developed a novel approach that worked particularly well: a translation from existential first-order logic over trees ($EFO(Tree)$) to *satisfiability modulo the theory of linear order*. Concretely, we can take the linear order to be $(\mathbb{N}, <)$, a subtheory of linear arithmetic. Satisfiability modulo theories (SMT) solvers such as `yices` [14], `z3` [10], and `cvc3` [1]

have been developed that can solve such linear arithmetic satisfiability problems very quickly; we obtained the best results with the `yices` and `z3` solvers.

This approach requires that we first approximate selection queries by positive $EFO(Tree)$ formulas. We consider two such approximations: The *existential first-order abstraction* replaces conditionals `if q then q1 else q2` by the monotone overapproximation (`if q then q1`), `q2` and then applies the translation to $FO(Tree)$ from [5]. The *simple path abstraction* approximates a query by overapproximating it with a set of simple path expressions, that is, plain sequences of XPath axis steps. Both of these abstractions are defined and verified in Appendix C.

Once we have approximated the destabilizer and target queries by $EFO(Tree)$ formulas, we translate the satisfiability question to one concerning sentences about a linear order. The translation is based on the well-known interval encoding [11] of XML documents, widely-used for indexing. We represent a tree node by a triple $(pre, post, tag)$ of numbers, representing the starting position, ending position, and tag of x respectively. We constrain each node x to satisfy $0 < x.pre \leq x.post$ and each pair of nodes x, y must satisfy:

(well nesting) $y.pre < x.pre < y.post \Rightarrow x.post < y.post$
(uniqueness) $x.pre = y.pre \iff x.post = y.post$
(tags unique) $x.pre = y.pre \Rightarrow x.tag = y.tag$

Then, given a formula ϕ over $child$, $desc$ and $sibling$, whose free variables are drawn from a finite set X, we can define:

$$\begin{aligned} follows(x,y) &= x.post < y.pre \\ desc(x,y) &= x.pre < y.pre \wedge y.post < x.post \\ btwn(x,y,z) &= desc(x,y) \wedge desc(y,z) \\ child(x,y) &= desc(x,y) \wedge \bigwedge_{z \in X} \neg(btwn(x,z,y)) \\ sibling(x,y) &= \exists z.child(z,x) \wedge child(z,y) \wedge follows(x,y) \end{aligned}$$

Node tests are represented using constraints $x.tag = i$, where i is an integer index surrogate for the actual element tag. We constrain text nodes to be leaves, i.e. $x.pre = x.post$, and to have a special tag not used for any element tag, such as -1. Note that in the definition of $child(x, y)$, we check that none of the nodes described by the variables of the formula ϕ are between x and y. This is correct because it suffices to restrict attention to these nodes when considering satisfiability (we omit a full proof but the details are similar to the satisfiability proofs in [21]).

Additional heuristics. Calling an external SMT solver is expensive, and often we can avoid this cost using heuristics to solve easy common cases of disjointness problems. For downward-only paths, overlap is exactly decidable in quadratic time [20], and we use this algorithm as well as other heuristics, described further in Appendix C.

5. EXPERIMENTS

5.1 Experimental Setup

We have implemented a static independence analyzer in OCaml based on destabilizers. The analysis takes a query and update expression and attempts to determine their independence, first using heuristics, and then optionally by calling an external solver.

We evaluated several analyses of varying degrees of complexity:

1. L_1: Solving via simple path approximations, using only heuristics and an exact algorithm for downward-only paths [20].
2. L_2: Solving via simple path approximations using `yices`.
3. L_3: Solving XQuery disjointness directly using `mona`. We permit `mona` at most one second to find a solution.
4. SCH: For comparison, using the schema-based analysis from our earlier work [2].

Each of the above levels L_i is cumulative, and stops as soon as a definite answer is obtained: e.g. L_2 involves first trying the heuristics and Hammerschmidt et al.'s algorithm [20], then trying `yices`. We also evaluated several other translations and external solvers in place of `yices`, in step L_2.

All external solvers were called as separate processes, and each such call is expensive; this represents an opportunity for improvement. Some of the solvers (`yices`, `mona`, `cvc3`) are available as libraries or in source code form, but others (`z3`, Genevès et al.'s XPath solver [17]) are not. We therefore chose not to try to integrate our implementation more closely with any external solvers in order to permit fair comparisons among the different solvers.

We used queries and updates derived from two standard sets of benchmark queries: twenty XQuery queries comprising the XMark benchmark [28], and sixteen queries from the XPathMark benchmark [15]. Our simplified forms of these queries exercise many aspects of XQuery and were used to evaluate previous work on schema-based independence analysis (e.g. [2]). The XMark queries were all translated to the more-restrictive core language accepted by our system, and we automatically generated rename, delete and insert updates based on each XPathMark query. We refer to these as UA_1–UA_8 and UB_1–UB_8. See Appendix E for further details of the queries and updates. Experiments were conducted on an Intel Pentium D (3.0 Ghz) running Ubuntu Linux 8.10, with 2GB of main memory.

5.2 Experimental Results

Running Time. For each update, we measured the total time taken to analyze its independence with respect to all queries. For each pair, we recorded system time elapsed between the start of execution and the time when the independence result is produced, including all time taken by external solvers and system overhead. We report times for the op-insensitive analysis only; the running times for op-sensitive analysis were similar. Figure 5 shows the total time taken by each analysis for each update to analyze all 36 benchmark queries. We also show the time for the schema-based analysis SCH from [2].

For comparison, we also evaluated the performance of our approach using Genevès et al.'s XPath solver, and using `z3` and `cvc3` (both via the interval encoding). The running times obtained using `z3` were similar to those using `yices`, while `cvc3` and Genevès et al.'s solver were not competitive. (We used a preliminary, unoptimized version of Genevès et al.'s solver; a more efficient version is under development but has not yet been released.)

Analysis Results. We summarize the results of the analysis by giving the percentages of queries that each level of the analysis was able to prove independent of each update, in Figure 6. These results are for the generic version of the analysis, so the operation of the update (insert, delete or rename) was irrelevant; the results are grouped by update numbers UA_i and UB_i. We also report the results of the schema-based analysis SCH and the combination of L_3 and SCH (labeled $L_3 + SCH$).

We also compare the generic analysis with the op-sensitive analysis. These results are summarized in Figure 7, where we show the total number of problems proved independent by the schema-based analysis SCH, destabilizer-based analyses L_1–L_3, and combined analyses $L_3 + SCH$.

Maintenance time. Finally, to evaluate the overall effectiveness of the approach, we measured the savings in view recomputation time for analyses SCH, L_1–L_3, and L_2+SCH. We summarize the results in Table 1 by presenting the total analysis overhead time and

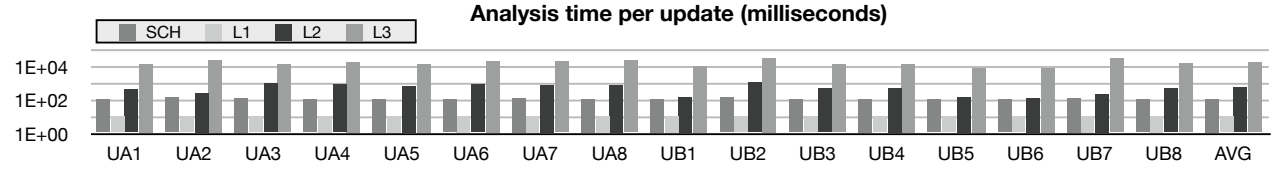

Figure 5: Running times for the generic analysis, in milliseconds (logarithmic scale), broken down by update and analysis level.

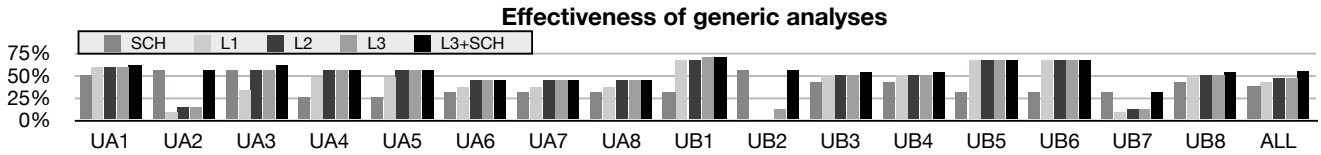

Figure 6: Effectiveness of the generic analysis, expressed as a percentage of query-update pairs determined independent, broken down by update and by analysis level.

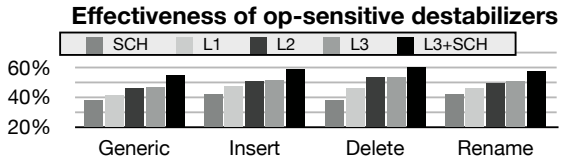

Figure 7: Effectiveness of the generic and op-sensitive analysis, expressed as a percentage of query-update pairs determined independent, by analysis level and update operation.

	SCH	L_1	L_2	L_3	$L_2 + SCH$
Static overhead	1.8	0.16	8.4	265	10
1.1MB	52	53	55	55	66
2.3MB	95	80	86	85	130
5.7MB	375	312	311	311	555

Table 1: Static analysis overheads and maintenance savings across whole benchmark (seconds).

total time saved for each technique on documents of increasing size. The documents are 1.1MB, 2.3MB and 5.7MB XMark data [28]. We used BaseX 6.1, an efficient XQuery engine that implements the full W3C update recommendation.

5.3 Discussion

Efficiency. The running times reported in Figure 5 show that the simple techniques used by L_1 are around ten times faster than SCH; the difference is likely due to the need for processing and analyzing the schema. The more sophisticated analysis level L_2 is generally slower than SCH, and L_3 is generally much slower, even with a 1 second time limit.

Effectiveness. Using the op-insensitive analysis, L_1 is able to determine independence in approximately 41% of cases. L_2 improves this to around 46%. However, the most sophisticated analysis, L_3, is only able to find a handful of additional independent pairs. Using op-sensitive destabilizers led to improvements of 4–6% for each level of the analysis. This demonstrates the value of taking the update operators into account.

We cannot easily measure the completeness of this approach, and do not have a theoretical characterization of its (relative) completeness. Informal inspection indicates that op-sensitive analysis is reasonably precise. For example, we manually constructed counterexamples to independence for all but five of the deletion-based problems that are not proved independent by op-sensitive L_3. Automating this heuristic counterexample-generation process could be an interesting topic for future work.

Figure 6 also gives an idea of how independence of the benchmark update/query pairs depends on a schema. Although our approach is good at detecting schema-free independence, there are certain pairs that can interfere in the absence of a schema but are independent for the XMark schema — e.g. UA_2, UB_2, UB_7 involve the descendant or ancestor axes and can modify data almost anywhere in the tree, so actually do interfere with most queries in the absence of a schema – thus schema-based analysis does better on these updates. Note that our approach can certainly accommodate schemas, simply by using schema-aware satisfiability tests – this is another opportunity for future work.

Impact on View Maintenance. Table 1 shows that all of the techniques save time overall on our benchmark compared to full recomputation, and that the new techniques in this paper find savings that previous approach SCH did not. Moreover, as document size increases, the savings-to-overhead ratio increases; already for a 5.7MB document the cost of analysis is minuscule compared to the time savings obtained. This is promising, but some important caveats apply to these results: we are comparing with full recomputation, not incremental view maintenance, and our benchmark is synthetic. However, dynamic view maintenance techniques do not yet appear to be available in standard XML databases. More work needs to be done to develop realistic view maintenance benchmarks that can be used to evaluate the techniques.

6. RELATED WORK

Due to space limits, we discuss only directly related work. Additional related work (including work on incremental view maintenance for XML) is reviewed in Appendix D.

This work is inspired partly by work on XML projection, where the goal is to identify nodes that can safely be deleted without affecting the result of a query [24, 6]; this is similar to independence problems involving deletion only.

Ghelli et al. [18] developed a static commutativity analysis, using paths to represent the sets of nodes accessed and updated by an ex-

pression. However, their analysis is based on a different XML update language proposal and does not address independence. Raghavachari and Shmueli [26] investigated the complexity of the independence problem for different fragments of downward XPath, while Sawires et al. [27] implemented an independence checker for downward XPath queries and updates, motivated by difficulties with incremental view maintenance in loosely-coupled systems. Benedikt and Cheney [2] developed a schema-based independence analysis for core XQuery over arbitrary XPath axes.

Our approach improves on these techniques in several ways. We handle arbitrary XPath axes and all of the essential features of XQuery. By analyzing operations we gain accuracy over any purely node-based approach, by using a variety of XQuery overlap tests we can gain accuracy on path-based abstraction (as used in [18, 26, 27, 24, 6]) and we do not rely on the presence of a schema (unlike [2]). Moreover, we have evaluated our approach on benchmark queries and updates exercising many features of XPath/XQuery.

Beyond these advantages, the notion of destabilizer itself is a key conceptual contribution of our work, since it factors the analysis problem cleanly into the approximation of the runtime updates by a query and an overlap test. This cleanly generalizes ideas in previous work. It is conceptually straightforward to extend our approach to handle constraints or schemas, simply by testing disjointness with respect to such constraints. Destabilizers may be useful in other data models, and may also be applicable to the dynamic, incremental maintenance problem.

7. CONCLUSIONS

We have introduced the notion of destabilizer, or a representation of the set of updates that may change the result of a given query. The problem of determining whether a query and update are (statically) independent reduces to calculating a destabilizer and testing whether the destabilizer and update are (statically) disjoint. This yields a general, modular and extensible framework for reasoning about independence.

We defined generic and kind-sensitive destabilizers via query rewriting for XQuery queries and updates. We implemented an independence analysis based on these destabilizers, and demonstrated its effectiveness and efficiency. Our approach offers a range of tradeoffs between precision and analysis cost, and can solve independence problems that no prior approach could solve. Even for fairly small documents, the overhead of our approach is much smaller than the savings it obtains. Combining this approach with schema-based analysis yields further savings. Moreover, these savings increase with document size.

There are many interesting directions for future work, such as schema-conscious disjointness analysis, counterexample generation, and combining static and dynamic view maintenance techniques.

Acknowledgments. Benedikt is supported in part by EPSRC EP/G004021/1 (the Engineering and Physical Sciences Research Council, UK) and by FET FP7-ICT-233599 (European Research Consortium). Cheney is supported by a Royal Society University Research Fellowship.

8. REFERENCES

[1] C. Barrett and C. Tinelli. CVC3. In *CAV 2007*, July 2007.

[2] M. Benedikt and J. Cheney. Schema-based independence analysis for XML updates. In *VLDB*, 2009.

[3] M. Benedikt and J. Cheney. Semantics, types and effects for XML Updates. In *DBPL*, 2009.

[4] M. Benedikt, W. Fan, and F. Geerts. XPath satisfiability in the presence of DTDs. In *PODS*, 2005.

[5] Michael Benedikt and Christoph Koch. From XQuery to relational logics. *ACM TODS*, 34(4):A25, 2009.

[6] V. Benzaken, G. Castagna, D. Colazzo, and K. Nguyên. Type-based XML projection. In *VLDB*, 2006.

[7] H. Björklund, W. Gelade, M. Marquardt, and W. Martens. Incremental XPath evaluation. In *ICDT*, 2009.

[8] J. Blakeley, N. Coburn, and P.-Å. Larson. Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates. *ACM TODS*, 14(3):369–400, 1989.

[9] D. Chamberlin, M. Dyck, D. Florescu, J. Melton, J. Robie, and J. Siméon. XQuery update facility 1.0. W3C Candidate Recommendation, August 2008.

[10] L. Mendonça de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.

[11] D. DeHaan, D. Toman, M. Consens, and M. T. Özsu. A comprehensive XQuery to SQL translation using dynamic interval encoding. In *SIGMOD*, 2003.

[12] Katica Dimitrova, Maged El-Sayed, and Elke A. Rundensteiner. Order-Sensitive View Maintenance of Materialized XQuery Views. In *ER*, 2003.

[13] D. Draper, P. Fankhauser, M. Fernández, A. Malhotra, K. Rose, M. Rys, J. Siméon, and P. Wadler. XQuery 1.0 and XPath 2.0 formal semantics. W3C Recommendation, January 2007.

[14] B. Dutertre and L. de Moura. The Yices SMT solver. Tool paper at http://yices.csl.sri.com/tool-paper.pdf, August 2006.

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

[16] P. Genevès and N. Layaïda. Deciding XPath containment with MSO. *Data Knowl. Eng.*, 63(1):108–136, 2007.

[17] P. Genevès, N. Layaïda, and A. Schmitt. Efficient static analysis of XML paths and types. In *PLDI*, 2007.

[18] G. Ghelli, K. Rose, and J. Siméon. Commutativity analysis for XML updates. *ACM TODS*, 33(4):1–47, 2008.

[19] T. Griffin, L. Libkin, and H. Trickey. An improved algorithm for the incremental recomputation of active relational expressions. *IEEE TKDE*, 9(3):508–511, 1997.

[20] B. Hammerschmidt, M. Kempa, and V. Linnemann. On the Intersection of XPath Expressions. In *IDEAS*, 2005.

[21] J. Hidders. Satisfiability of XPath expressions. In *DBPL*, 2003.

[22] N. Klarlund and A. Møller. Mona v. 1.4 user manual. Technical Report BRICS NS-01-1, U. Aarhus, 2001.

[23] A. Levy and Y. Sagiv. Queries Independent of Updates. In *VLDB*, 1993.

[24] A. Marian and J. Siméon. Projecting XML documents. In *VLDB*, 2003.

[25] X. Qian and G. Wiederhold. Incremental Recomputation of Active Relational Expressions. *IEEE TKDE*, 3(3):337–341, 1991.

[26] M. Raghavachari and O. Shmueli. Conflicting XML updates. In *EDBT*, 2006.

[27] A. Sawires, J. Tatemura, O. Po, D. Agrawal, A. El Abbadi, and K. Candan. Maintaining XPath Views In Loosely Coupled Systems. In *VLDB*, 2006.

[28] A. Schmidt, F. Waas, M. Kersten, M. Carey, I. Manolescu, and R. Busse. XMark: A Benchmark for XML Data Management. In *VLDB*, 2002.

[29] Sergei G. Vorobyov. An improved lower bound for the elementary theories of trees. In *CADE*, 1996.

$$\frac{\sigma(\mathtt{l}) = \mathsf{text}[s] \quad \mathtt{l}' \notin \mathrm{dom}(\sigma)}{\sigma, \mathtt{l} \overset{\mathsf{copy}}{\mapsto} \sigma[\mathtt{l}' := \mathsf{text}[s]], \mathtt{l}'}$$

$$\frac{\sigma(\mathtt{l}) = a[\mathtt{L}] \quad \sigma, \mathtt{L} \overset{\mathsf{copy}}{\mapsto} \sigma', \mathtt{L}' \quad \mathtt{l}' \notin \mathrm{dom}(\sigma')}{\sigma, \mathtt{l} \overset{\mathsf{copy}}{\mapsto} \sigma'[\mathtt{l}' := a[\mathtt{L}']], \mathtt{l}'}$$

$$\frac{}{\sigma, () \overset{\mathsf{copy}}{\mapsto} \sigma, ()}$$

$$\frac{\sigma, \mathtt{L}_1 \overset{\mathsf{copy}}{\mapsto} \sigma', \mathtt{L}_1' \quad \sigma', \mathtt{L}_2 \overset{\mathsf{copy}}{\mapsto} \sigma'', \mathtt{L}_2'}{\sigma, \mathtt{L}_1 \cdot \mathtt{L}_2 \overset{\mathsf{copy}}{\mapsto} \sigma'', \mathtt{L}_1' \cdot \mathtt{L}_2'}$$

Figure 8: Copying rules

APPENDIX

A. SEMANTICS OF QUERIES AND UPDATES

A.1 Query semantics

A store is modeled as a mapping $\sigma : \mathit{Loc} \to \Sigma \times \mathit{Loc}^* \uplus \mathsf{String}$, where Σ is the set of element tags; we often write $\sigma(\mathtt{l}) = a[\mathtt{L}]$ or $\mathsf{text}[s]$, instead of $\sigma(\mathtt{l}) = (a, \mathtt{L})$ or s, respectively.

An environment is modeled as a mapping $\gamma : \mathit{Var} \to \mathit{Loc}^*$ from variables to lists of nodes.

We define the semantics of queries using the following judgments:

- $\sigma, \mathtt{L} \overset{\mathsf{copy}}{\mapsto} \sigma', \mathtt{L}'$ — takes a store and a list of nodes in the store and copies the subtrees below the nodes, yielding a new store and a new list of nodes pointing to the roots of the new trees.
- $\sigma, \gamma \models \mathtt{q} \Rightarrow \sigma', \mathtt{L}$ — takes a store and environment and evaluates a query to a new store and a list of nodes.
- $\sigma, \gamma \models \mathtt{q} \overset{\mathsf{copy}}{\Rightarrow} \sigma', \mathtt{L}$ — an abbreviation for evaluating q and then making a copy of the result.
- $\sigma, \gamma, x \in \mathtt{L} \models^\star \mathtt{q} \Rightarrow \sigma', \mathtt{L}$ — takes a store, environment, variable x, and list of nodes $\mathtt{L}$, and evaluates the query with x bound to each node in $\mathtt{L}$, concatenating the results in order.
- $\sigma \models \mathtt{L}/ax{::}\phi \overset{\mathsf{step}}{\Rightarrow} \mathtt{L}'$ —given a store and list of nodes, evaluates XPath axis step $ax :: \phi$ to produce a new list of nodes. This judgment is standard and we omit its definition.

A.2 Update semantics

Atomic updates. We consider atomic updates of the form:

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

Here, the direction $\mathtt{d}$ indicates whether to insert before ($\leftarrow$), after ($\rightarrow$), or into the child list in first ($\swarrow$), last ($\searrow$) or arbitrary position ($\downarrow$). We define the *target* of an atomic update ι to be the distinguished node $\mathtt{L}$; for example $\mathtt{Targ}(\mathtt{ins}(\mathtt{L}, \mathtt{d}, \mathtt{l})) = \mathtt{l}$.

Our basic update model will be *sequences of atomic updates*. We will let ω range over sequences with the empty sequence written ϵ and concatenation written $\omega; \omega'$. Formally, we can define the semantics of atomic updates as a relation $\sigma \models \iota \rightsquigarrow \sigma'$; for the precise definition, see Figure 10. We will refer to this relation as `EvalUpd`. Our semantics is a formalization of that given by the W3C's candidate recommendation for updates, see [9].

Updating expressions. The W3C XQuery Update proposal defines high-level *updating expressions* that can be used to generate atomic updates that are to be performed [9]. We use the following

$$\frac{}{\sigma, \gamma \models \$x \Rightarrow \sigma, \gamma(x)}$$

$$\frac{}{\sigma, \gamma \models () \Rightarrow \sigma, ()}$$

$$\frac{\sigma, \gamma \models \mathtt{q}_1 \Rightarrow \sigma_2, \mathtt{L}_1 \quad \sigma_2, \gamma \models \mathtt{q}_2 \Rightarrow \sigma_3, \mathtt{L}_2}{\sigma, \gamma \models \mathtt{q}_1, \mathtt{q}_2 \Rightarrow \sigma_3, \mathtt{L}_1 \cdot \mathtt{L}_2}$$

$$\frac{\sigma, \gamma \models \mathtt{q} \overset{\mathsf{copy}}{\Rightarrow} \sigma_2, \mathtt{L} \quad \mathtt{l} \notin \mathrm{dom}(\sigma_2)}{\sigma, \gamma \models \langle a \rangle \mathtt{q} \langle /a \rangle \Rightarrow \sigma_2[\mathtt{l} := a[\mathtt{L}]], \mathtt{l}}$$

$$\frac{\sigma, \gamma \models \mathtt{q} \Rightarrow \sigma_2, \mathtt{l} \cdot \mathtt{L} \quad \sigma_2, \gamma \models \mathtt{q}_1 \Rightarrow \sigma_3, \mathtt{L}_1}{\sigma, \gamma \models \mathtt{if}\ \mathtt{q}\ \mathtt{then}\ \mathtt{q}_1\ \mathtt{else}\ \mathtt{q}_2 \Rightarrow \sigma_3, \mathtt{L}_1}$$

$$\frac{\sigma, \gamma \models \mathtt{q} \Rightarrow \sigma_2, () \quad \sigma_2, \gamma \models \mathtt{q}_2 \Rightarrow \sigma_3, \mathtt{L}_2}{\sigma, \gamma \models \mathtt{if}\ \mathtt{q}\ \mathtt{then}\ \mathtt{q}_1\ \mathtt{else}\ \mathtt{q}_2 \Rightarrow \sigma_3, \mathtt{L}_2}$$

$$\frac{\sigma, \gamma \models \mathtt{q}_1 \Rightarrow \sigma_2, \mathtt{L} \quad \sigma_2, \gamma, x \in \mathtt{L} \models^\star \mathtt{q}_2 \Rightarrow \sigma_3, \mathtt{L}'}{\sigma, \gamma \models \mathtt{for}\ \$x \in \mathtt{q}_1\ \mathtt{return}\ \mathtt{q}_2 \Rightarrow \sigma_3, \mathtt{L}'}$$

$$\frac{\sigma \models \gamma(x)/ax{::}\phi \overset{\mathsf{step}}{\Rightarrow} \mathtt{L}}{\sigma, \gamma \models x/ax :: \phi \Rightarrow \sigma, \mathtt{L}}$$

$$\frac{\sigma, \gamma \models \mathtt{q} \Rightarrow \sigma_0, \mathtt{L}_0 \quad \sigma_0, \mathtt{L}_0 \overset{\mathsf{copy}}{\mapsto} \sigma', \mathtt{L}}{\sigma, \gamma \models \mathtt{q} \overset{\mathsf{copy}}{\Rightarrow} \sigma', \mathtt{L}}$$

$$\frac{}{\sigma, \gamma, x \in () \models^\star \mathtt{q} \Rightarrow \sigma, ()}$$

$$\frac{\sigma, \gamma[x := \mathtt{l}] \models \mathtt{q} \Rightarrow \sigma_2, \mathtt{L}_1 \quad \sigma, \gamma, x \in \mathtt{L} \models^\star \mathtt{q} \Rightarrow \sigma_3, \mathtt{L}_2}{\sigma, \gamma, x \in \mathtt{l} \cdot \mathtt{L} \models^\star \mathtt{q} \Rightarrow \sigma_3, \mathtt{L}_1 \cdot \mathtt{L}_2}$$

Figure 9: Query evaluation rules

core language for update expressions:

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

The XQuery Update proposal re-uses existing query syntax for updates. The $()$ expression does nothing, while $\mathtt{u}, \mathtt{u}'$ is sequential composition, and XQuery conditionals and for-loops can also be used. There are four atomic update expressions: insertion $\mathtt{insert}\ \mathtt{q}\ \mathtt{d}\ \mathtt{q}_0$, which says to insert a copy of the value of $\mathtt{q}$ in position $\mathtt{d}$ relative to node $\mathtt{q}_0$; deletion $\mathtt{delete}\ \mathtt{q}_0$, which says to delete the node $\mathtt{q}_0$, disconnecting it from its parent; renaming $\mathtt{rename}\ \mathtt{q}_0\ \mathtt{as}\ a$, which says to rename the node $\mathtt{q}_0$ to a and replacement $\mathtt{replace}\ \mathtt{q}_0\ \mathtt{with}\ \mathtt{q}$, which says to replace the value of node $\mathtt{q}_0$ with a copy of $\mathtt{q}$. In each case, the target expression $\mathtt{q}_0$ is required to be a selection query and expected to evaluate to a single node; if not, evaluation fails. Again, we omit `let`-binding without loss of generality.

Semantics of updates. 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 $\sigma, \gamma \models \mathtt{u} \Rightarrow \sigma', \omega$, defined in Figure 11 (and based on a semantics developed elsewhere [3]). Note that, as for queries, the store may grow as a result of allocation, but trees already in σ do *not* change while the update expression is being evaluated. Next, ω is *validated* to avoid pathological problems such as renaming a node in two different ways. We do not model this validation phase explicitly here. Finally, the pending updates are *applied* to the store in some order. The W3C semantics mandates reordering the update list so that insertions and renamings are applied first, then replacements, and finally deletions. We will conservatively assume that atomic updates in the pending update list might be performed in *any* order. A static analysis that is sound with respect to this se-

$$\frac{\sigma(\mathtt{l}') = a[\mathtt{L}_1 \cdot \mathtt{l} \cdot \mathtt{L}_2]}{\sigma \models \mathtt{ins}(\mathtt{L}, \leftarrow, \mathtt{l}) \rightsquigarrow \sigma[\mathtt{l}' := a[\mathtt{L}_1 \cdot \mathtt{L} \cdot \mathtt{l} \cdot \mathtt{L}_2]]}$$

$$\frac{\sigma(\mathtt{l}) = a[\mathtt{L}']}{\sigma \models \mathtt{ins}(\mathtt{L}, \swarrow, \mathtt{l}) \rightsquigarrow \sigma[\mathtt{l} := a[\mathtt{L} \cdot \mathtt{L}']]}$$

$$\frac{\sigma(\mathtt{l}') = a[\mathtt{L}_1 \cdot \mathtt{l} \cdot \mathtt{L}_2]}{\sigma \models \mathtt{ins}(\mathtt{L}, \rightarrow, \mathtt{l}) \rightsquigarrow \sigma[\mathtt{l}' := a[\mathtt{L}_1 \cdot \mathtt{l} \cdot \mathtt{L} \cdot \mathtt{L}_2]]}$$

$$\frac{\sigma(\mathtt{l}) = a[\mathtt{L}']}{\sigma \models \mathtt{ins}(\mathtt{L}, \searrow, \mathtt{l}) \rightsquigarrow \sigma[\mathtt{l} := a[\mathtt{L}' \cdot \mathtt{L}]]}$$

$$\frac{\sigma(\mathtt{l}) = a[\mathtt{L}_1 \cdot \mathtt{L}_2]}{\sigma \models \mathtt{ins}(\mathtt{L}, \downarrow, \mathtt{l}) \rightsquigarrow \sigma[\mathtt{l} := a[\mathtt{L}_1 \cdot \mathtt{L} \cdot \mathtt{L}_2]]}$$

$$\frac{\sigma(\mathtt{l}) = a[\mathtt{L}]}{\sigma \models \mathtt{ren}(\mathtt{l}, b) \rightsquigarrow \sigma[\mathtt{l} := b[\mathtt{L}]]}$$

$$\frac{\sigma(\mathtt{l}') = a[\mathtt{L}_1 \cdot \mathtt{l} \cdot \mathtt{L}_2]}{\sigma \models \mathtt{repl}(\mathtt{l}, \mathtt{L}) \rightsquigarrow \sigma[\mathtt{l}' := a[\mathtt{L}_1 \cdot \mathtt{L} \cdot \mathtt{L}_2]]}$$

$$\frac{\sigma(\mathtt{l}') = a[\mathtt{L}_1 \cdot \mathtt{l} \cdot \mathtt{L}_2]}{\sigma \models \mathtt{del}(\mathtt{l}) \rightsquigarrow \sigma[\mathtt{l}' := a[\mathtt{L}_1 \cdot \mathtt{L}_2]]}$$

$$\frac{}{\sigma \models \epsilon \rightsquigarrow \sigma}$$

$$\frac{\sigma \models \omega_1 \rightsquigarrow \sigma' \quad \sigma' \models \omega_2 \rightsquigarrow \sigma''}{\sigma \models \omega_1, \omega_2 \rightsquigarrow \sigma''}$$

$$\frac{\sigma, \gamma \models \mathtt{u} \Rightarrow \sigma', \omega \quad \omega' \text{ any permutation of } \omega \quad \sigma' \models \omega' \rightsquigarrow \sigma''}{\sigma, \gamma \models \mathtt{u} \rightsquigarrow \sigma''}$$

Figure 10: Update application

$$\frac{}{\sigma, \gamma \models () \Rightarrow \sigma, \epsilon}$$

$$\frac{\sigma_1, \gamma \models \mathtt{u}_1 \Rightarrow \sigma_2, \omega_1 \quad \sigma_2, \gamma \models \mathtt{u}_2 \Rightarrow \sigma_3, \omega_2}{\sigma_1, \gamma \models \mathtt{u}_1, \mathtt{u}_2 \Rightarrow \sigma_3, \omega_1; \omega_2}$$

$$\frac{\sigma_1, \gamma \models \mathtt{q} \Rightarrow \sigma_2, \mathtt{l} \cdot \mathtt{L} \quad \sigma_2, \gamma \models \mathtt{u}_1 \Rightarrow \sigma_3, \omega_1}{\sigma_1, \gamma \models \mathtt{if\ q\ then\ u_1\ else\ u_2} \Rightarrow \sigma_3, \omega_1}$$

$$\frac{\sigma_1, \gamma \models \mathtt{q} \Rightarrow \sigma_2, () \quad \sigma_2, \gamma \models \mathtt{u}_2 \Rightarrow \sigma_3, \omega_2}{\sigma_1, \gamma \models \mathtt{if\ q\ then\ u_1\ else\ u_2} \Rightarrow \sigma_3, \omega_2}$$

$$\frac{\sigma_1, \gamma \models \mathtt{q} \Rightarrow \mathtt{L}, \sigma_2 \quad \sigma, \gamma, x \in \mathtt{L} \models^\star \mathtt{u} \Rightarrow \sigma_3, \omega}{\sigma_1, \gamma \models \mathtt{for}\ \$x \in q\ \mathtt{return\ u} \Rightarrow \sigma_3, \omega}$$

$$\frac{\sigma_1, \gamma \models \mathtt{q}_1 \overset{\text{copy}}{\Rightarrow} \sigma_2, \mathtt{L}_1 \quad \sigma_2, \gamma \models \mathtt{q}_2 \Rightarrow \sigma_3, \mathtt{l}_2}{\sigma_1, \gamma \models \mathtt{insert\ q_1\ d\ q_2} \Rightarrow \sigma_3, \mathtt{ins}(\mathtt{L}_1, \mathtt{d}, \mathtt{l}_2)}$$

$$\frac{\sigma_1, \gamma \models \mathtt{q} \Rightarrow \sigma_2, \mathtt{l}}{\sigma_1, \gamma \models \mathtt{delete\ q} \Rightarrow \sigma_2, \mathtt{del}(\mathtt{l})}$$

$$\frac{\sigma_1, \gamma \models \mathtt{q}_1 \Rightarrow \sigma_2, \mathtt{l}_1 \quad \sigma_2, \gamma \models \mathtt{q}_2 \overset{\text{copy}}{\Rightarrow} \sigma_3, \mathtt{L}_2}{\sigma_1, \gamma \models \mathtt{replace\ q_1\ with\ q_2} \Rightarrow \sigma_3, \mathtt{repl}(\mathtt{l}_1, \mathtt{L}_2)}$$

$$\frac{\sigma_1, \gamma \models \mathtt{q} \Rightarrow \sigma_2, \mathtt{l}}{\sigma_1, \gamma \models \mathtt{rename\ q\ as}\ a \Rightarrow \sigma_2, \mathtt{ren}(\mathtt{l}, a)}$$

$$\frac{}{\sigma, \gamma, x \in () \models^\star \mathtt{u} \Rightarrow \sigma, \epsilon}$$

$$\frac{\sigma_1, \gamma[x := \mathtt{l}] \models \mathtt{u} \Rightarrow \sigma_2, \omega_1 \quad \sigma_2, \gamma, x \in \mathtt{L} \models^\star \mathtt{u} \Rightarrow \sigma_3, \omega_2}{\sigma_1, \gamma, x \in \mathtt{l} \cdot \mathtt{L} \models^\star \mathtt{u} \Rightarrow \sigma_3, \omega_1; \omega_2}$$

Figure 11: Update expression evaluation

mantics will also be sound with respect to any implementation of the W3C semantics.

To summarize, the semantics of updates is defined using the following judgments in Figure 10 and Figure 11.

- $\sigma \models \iota \rightsquigarrow \sigma'$ — given a store σ and an atomic update, produce a store resulting from applying ι to σ. (This, and all later judgments, can be nondeterministic because of node copying and because insert-into operations are nondeterministic.)
- $\sigma \models \omega \rightsquigarrow \sigma'$ — given a store σ and an atomic update sequence, perform the atomic updates in order.
- $\sigma, \gamma \models \mathtt{u} \Rightarrow \sigma', \omega$ — given a store σ and variable environment γ, evaluate $\mathtt{u}$ to a new store and atomic update list ω.
- $\sigma, \gamma, x \in \mathtt{L} \models^\star \mathtt{u} \Rightarrow \sigma', \omega$ — given a store σ, variable environment γ, variable x and node list $\mathtt{L}$, evaluate update $\mathtt{u}$ with x bound to each element of $\mathtt{L}$ in turn, concatenating the resulting update sequences in order.
- $\sigma, \gamma \models \mathtt{u} \rightsquigarrow \sigma'$ — given input store and variable environment γ, evaluate u and apply the resulting atomic updates in any order, yielding σ'.

B. PROOFS

B.1 Proof of Theorem 1

Recall the statement of the result: There is no elementary algorithm for constructing a pointwise minimal static destabilizer.

For the first part, recall that the satisfiability problem for first-order logic on labeled trees can not be resolved in elementary time [29]. From the results of [5] it follows that the satisfiability problem for $SelXQ$ is non-elementary. Now consider the document Σ_0 consisting only of a single node, and a boolean query Q. If Q is unsatisfiable, then the empty set is a minimal set of nodes of Σ_0 having the property that every update sequence making Q turn from false to true on Σ_0 contains a node in the set. If Q is satisfiable but does not hold on Σ_0, then the empty set is not such a set of nodes, since we could replace the root of Σ_0 by some document satisfying Q. Hence we can determine whether Q is satisfiable by first running Q on d_0 and then running an exact destabilizer for Q on d_0 and checking if the result is empty. Since evaluation of a $SelXQ$ query is polynomial time (e.g. [5]) this would provide a contradiction. □

C. DETAILS OF INDEPENDENCE ANALYSIS

C.1 Additional destabilizer rules

In Figure 12 we show additional rules for rename, insert before, and insert after destabilizers.

For renames, note that a rename operation can only affect the label of an element node, not the tree structure. Hence, for positive boolean and node destabilizers, rename operations that do not involve element label tests have empty destabilizers, no matter what axis is used. Furthermore, for negative boolean destabilizers, the only way to destabilize a step involving a node test $ax :: a$ is to rename a node already labeled a to something else, so in this case we can improve precision slightly.

The insert-before and insert-after axes are symmetric; we discuss only the former case. For positive boolean destabilizers, we can never make a `child` $:: *$ step nonempty by inserting before a node, since since if a node has no children we cannot add a child by inserting before a node, while if a node has children already then inserting before one of them does not change the Boolean value of

$$
\begin{array}{rcl}
\Delta^{\mathtt{b+}}_{\mathtt{rename}}(\$x/ax :: a) & = & \$x/ax :: * \\
\Delta^{\mathtt{b+}}_{\mathtt{rename}}(\$x/ax :: \phi) & = & () \quad (\phi \in \{*, \mathtt{text}()\}) \\
\Delta^{\mathtt{b+}}_{\mathtt{insert}(\leftarrow)}(\$x/\mathtt{child} :: *) & = & () \\
\Delta^{\mathtt{b+}}_{\mathtt{insert}(\leftarrow)}(\$x/\mathtt{child} :: \phi) & = & \$x/\mathtt{child} :: * \quad (\phi \neq *) \\
\Delta^{\mathtt{b+}}_{\mathtt{insert}(\leftarrow)}(\$x/\mathtt{foll\_sib} :: \phi) & = & (\$x/\mathtt{foll\_sib} :: *) \\
\Delta^{\mathtt{b+}}_{\mathtt{insert}(\leftarrow)}(\$x/\mathtt{prec\_sib} :: \phi) & = & (\$x, \$x/\mathtt{prec\_sib} :: *) \\
\Delta^{\mathtt{b+}}_{\mathtt{insert}(\rightarrow)}(\$x/\mathtt{child} :: *) & = & () \\
\Delta^{\mathtt{b+}}_{\mathtt{insert}(\rightarrow)}(\$x/\mathtt{child} :: \phi) & = & \$x/\mathtt{child} :: * \quad (\phi \neq *) \\
\Delta^{\mathtt{b+}}_{\mathtt{insert}(\rightarrow)}(\$x/\mathtt{foll\_sib} :: \phi) & = & (\$x, \$x/\mathtt{foll\_sib} :: *) \\
\Delta^{\mathtt{b+}}_{\mathtt{insert}(\rightarrow)}(\$x/\mathtt{prec\_sib} :: \phi) & = & (\$x/\mathtt{prec\_sib} :: *) \\
\\
\Delta^{\mathtt{b-}}_{\mathtt{rename}}(\$x/ax :: a) & = & \$x/ax :: a \\
\Delta^{\mathtt{b-}}_{\mathtt{rename}}(\$x/ax :: \phi) & = & () \quad (\phi \in \{*, \mathtt{text}()\}) \\
\Delta^{\mathtt{b-}}_{\mathtt{insert}(\leftarrow)}(\$x/ax :: \phi) & = & () \\
\Delta^{\mathtt{b-}}_{\mathtt{insert}(\rightarrow)}(\$x/ax :: \phi) & = & () \\
\\
\Delta^{\mathtt{n}}_{\mathtt{rename}}(\$x/ax :: a) & = & \$x/ax :: * \\
\Delta^{\mathtt{n}}_{\mathtt{rename}}(\$x/ax :: \phi) & = & () \quad (\phi \in \{*, \mathtt{text}()\}) \\
\Delta^{\mathtt{n}}_{\mathtt{insert}(\leftarrow)}(\$x/\mathtt{child} :: \phi) & = & \$x/\mathtt{child} :: * \\
\Delta^{\mathtt{n}}_{\mathtt{insert}(\leftarrow)}(\$x/\mathtt{foll\_sib} :: \phi) & = & (\$x/\mathtt{foll\_sib} :: *) \\
\Delta^{\mathtt{n}}_{\mathtt{insert}(\leftarrow)}(\$x/\mathtt{prec\_sib} :: \phi) & = & (\$x, \$x/\mathtt{prec\_sib} :: *) \\
\Delta^{\mathtt{n}}_{\mathtt{insert}(\rightarrow)}(\$x/\mathtt{child} :: \phi) & = & \$x/\mathtt{child} :: * \\
\Delta^{\mathtt{n}}_{\mathtt{insert}(\rightarrow)}(\$x/\mathtt{foll\_sib} :: \phi) & = & (\$x, \$x/\mathtt{foll\_sib} :: *) \\
\Delta^{\mathtt{n}}_{\mathtt{insert}(\rightarrow)}(\$x/\mathtt{prec\_sib} :: \phi) & = & (\$x/\mathtt{prec\_sib} :: *)
\end{array}
$$

Figure 12: Additional op-sensitive destabilizer rules

this step. Of course, for a node-test destabilizer $\mathtt{child} :: \phi$ where ϕ is not $*$, we can make the result go from empty to nonempty by inserting a node matching ϕ before an already-present node. For negative destabilizers, there is no way an insert-before operation can make the result of a child step empty. Finally, for node destabilizers, we conservatively assume that any insert-before to a child list of a node can destabilize the node list returned by the child step. Other axes such as descendant can be handled similarly to the child axis. For the preceding and following axes, however, we can leverage the order behavior of the insert-before operation. Specifically, for the following sibling axis, only inserts before some following sibling can affect the result of this step, while inserts before both the node and its preceding siblings can affect the preceding-sibling step.

C.2 Update target queries

We define update target queries $\mathtt{Targ}_{op}(\mathtt{u})$ as shown in Figure 13, where $op \in \mathtt{Ops}$. Note that $\mathtt{Targ}_{op}(\mathtt{u})$ is a selection query. Its key correctness property is:

LEMMA 1. *Suppose σ is a store and $(\sigma', \omega) \in \mathtt{Upd}[\![\mathtt{u}]\!]\sigma$. Then for any $\iota \in \omega$, if ι : op and $\mathtt{Targ}(\iota) \in \sigma$ then*

$$\mathtt{Targ}(\iota) \in \mathtt{Sel}\left[\!\left[\mathtt{Targ}_{op}(\mathtt{u})\right]\!\right]\sigma \ .$$

C.3 Query translations

We employ three translations from selection queries to other representations:

- The *first-order abstraction* translates a selection query to a first-order formula over the basic axis predicates.
- The *existential first-order abstraction* translates a selection query to a *positive* existential first-order formula (that is, one with no negation or universal quantifiers).

$$
\begin{array}{rcl}
\mathtt{Targ}_{op}(()) & = & () \\
\mathtt{Targ}_{op}(\mathtt{u}_1, \mathtt{u}_2) & = & \mathtt{Targ}_{op}(\mathtt{u}_1), \mathtt{Targ}_{op}(\mathtt{u}_2) \\
\mathtt{Targ}_{op}(\mathtt{if\ q\ then\ u}_1\ \mathtt{else\ u}_2) & = & \mathtt{if\ q\ then\ Targ}_{op}(\mathtt{u}_1) \\
 & & \mathtt{else\ Targ}_{op}(\mathtt{u}_2) \\
\mathtt{Targ}_{op}(\mathtt{for}\ \$x \in \mathtt{q\ return\ u}) & = & \mathtt{for}\ \$x \in \mathtt{q\ return\ Targ}_{op}(\mathtt{u}) \\
\mathtt{Targ}_{op}(\mathtt{insert\ q\ d\ q}') & = & \mathtt{q}' \quad (\mathtt{insert(d)} : op) \\
\mathtt{Targ}_{op}(\mathtt{delete\ q}') & = & \mathtt{q}' \quad (\mathtt{delete} : op) \\
\mathtt{Targ}_{op}(\mathtt{rename\ q}'\ \mathtt{as}\ a) & = & \mathtt{q}' \quad (\mathtt{rename} : op) \\
\mathtt{Targ}_{op}(\mathtt{replace\ q\ with\ q}') & = & \mathtt{q}' \quad (\mathtt{replace} : op) \\
\mathtt{Targ}_{op}(\mathtt{u}) & = & () \quad \text{(otherwise)}
\end{array}
$$

Figure 13: Target query of an update (with respect to update operator op)

$$
\begin{array}{rcl}
\mathsf{FO}_y(()) & = & \bot \\
\mathsf{FO}_y(\$x) & = & x = y \\
\mathsf{FO}_y(\$x/axis :: *) & = & Axis(x, y) \\
\mathsf{FO}_y(\$x/axis :: \mathtt{text}()) & = & Axis(x, y) \wedge \mathtt{text}(y) \\
\mathsf{FO}_y(\$x/axis :: A) & = & Axis(x, y) \wedge A(y) \\
\mathsf{FO}_y((\mathtt{q}_1, \mathtt{q}_2)) & = & \mathsf{FO}_y(\mathtt{q}_1) \vee \mathsf{FO}_y(\mathtt{q}_2) \\
\mathsf{FO}_y(\mathtt{if\ q\ then\ q}_1\ \mathtt{else\ q}_2) & = & (\exists z.\mathsf{FO}_z(\mathtt{q}) \wedge \mathsf{FO}_y(\mathtt{q}_1)) \\
 & \vee & (\neg(\exists z.\mathsf{FO}_z(\mathtt{q})) \wedge \mathsf{FO}_y(\mathtt{q}_2) \\
\mathsf{FO}_y(\mathtt{for}\ \$x \in \mathtt{q}_1\ \mathtt{return\ q}_2(x)) & = & \exists x.\mathsf{FO}_x(\mathtt{q}_1) \wedge \mathsf{FO}_y(\mathtt{q}_2(x))
\end{array}
$$

Figure 14: Translating selection queries to first-order formulas. In each case we assume y is not free in the query.

- The *path abstraction* translates a selection query to a set of paths (each of which can be viewed individually as an EFO formula)

These translations from queries to first-order formulas or paths are well-understood and straightforward (see e.g. [5]); we provide them in this appendix for completeness.

First-order translation. Figure 14 defines a function $\mathsf{FO}_x(\mathtt{q})$ that maps a selection query $\mathtt{q}(y_1, \ldots, y_n)$ to an equivalent first-order formula $\phi(x, y_1, \ldots, y_n)$, in the following sense:

PROPOSITION 1. *For any document selection query $\mathtt{q}(\$doc)$, we have $\mathtt{Sel}[\![\mathtt{q}]\!]\sigma = \{\mathtt{l} \mid \sigma \models \phi(\mathtt{l}, root(\sigma))\}$.*

Existential first-order abstraction. For some analysis techniques, it is necessary to over-approximate a selection query by a positive existential formula, that is, avoiding the use of negation or universal quantification. In Figure 14, we can see that only one case can involve negation, namely the case for the else-branch of a conditional. We can safely over-approximate a selection query by

$$
\begin{array}{rcl}
\mathtt{Pt}(\Gamma, ()) & = & \emptyset \\
\mathtt{Pt}(\Gamma, \$x) & = & \Gamma(x) \\
\mathtt{Pt}(\Gamma, \$x/step) & = & \{p/step \mid p \in \Gamma(x)\} \\
\mathtt{Pt}(\Gamma, (\mathtt{q}_1, \mathtt{q}_2)) & = & \mathtt{Pt}(\Gamma, \mathtt{q}_1) \cup \mathtt{Pt}(\Gamma, \mathtt{q}_2) \\
\mathtt{Pt}(\Gamma, \mathtt{if\ q\ then\ q}_1\ \mathtt{else\ q}_2) & = & \mathtt{Pt}(\Gamma, \mathtt{q}_1) \cup \mathtt{Pt}(\Gamma, \mathtt{q}_2) \\
\mathtt{Pt}(\Gamma, \mathtt{for}\ \$x \in \mathtt{q}_1\ \mathtt{return\ q}_2) & = & \mathtt{Pt}(\Gamma[x := \mathtt{Pt}(\Gamma, \mathtt{q}_1)], \mathtt{q}_2) \\
\mathtt{Pt}(\mathtt{q}) & = & \mathtt{Pt}([\$doc = /], \mathtt{q})
\end{array}
$$

Figure 15: Approximating selection queries by paths. Here, Γ is an environment mapping each variable to a set of paths.

rewriting each conditional in it as follows:

$$\texttt{if q then q}_1\ \texttt{else q}_2 \rightarrow (\texttt{if q then q}_1), q_2$$

and then translating the query to a first-order formula as shown in Figure 14.

Path abstraction. Figure 15 defines a function $\texttt{Pt(q)}$ that constructs a set of XPath paths that covers all of the nodes that might be returned by q.

PROPOSITION 2. *For any selection query* q *and store* σ*, we have* $\texttt{Sel}[\![\texttt{q}]\!]\sigma \subseteq \bigcup_{p \in \texttt{Pt(q)}} \texttt{Sel}[\![p]\!]\sigma$*. Hence, if every pair of paths* $p_1 \in \texttt{Pt}(\texttt{q}_1)$ *and* $p_2 \in \texttt{Pt}(\texttt{q}_2)$ *are disjoint then so are* $\texttt{q}_1$ *and* $\texttt{q}_2$*.*

C.4 Heuristics for path disjointness

The SMT-based approach can be used with path abstraction, since the paths translate into simple positive queries as described in Section 4. In addition we employ several heuristics for either simplifying or solving certain classes of path disjointness problems.

1. *Suffix incompatibility.* If two paths end with suffixes of child steps that are obviously incompatible (e.g. $//*/a/b$ and $//c//d/e$ then the paths cannot overlap.
2. *Displacement tests.* Define the *minimum displacement* of a path $\delta_{min}(p)$ to be the smallest difference in height between the root of the tree and the node selected by the path, if this exists; otherwise $\delta_{min}(p) = -\infty$. Similarly, define the *maximum displacement* $\delta_{max}(p)$ to be the maximum difference in height between the root of the tree and the node selected by the path, if this exists; otherwise $\delta_{max}(p) = +\infty$. For example, the minimum and maximum displacement of $/\texttt{child} :: */\texttt{desc} :: a$ are 2 and ∞, respectively. Define the *displacement interval* $\delta(p)$ of p to be the set $\{n \in \mathbb{Z} \mid \delta_{min}(p) \leq n \leq \delta_{max}(p)\}$.
   Given paths p_1, p_2, if $\delta_{max}(p_i) < 0$ for either p_1 or p_2 then p_i is unsatisfiable (starting from the root of a tree) and so p_1 and p_2 are trivially disjoint. Otherwise, if $\delta(p_1) \cap \delta(p_2) = \emptyset$ then p_1 and p_2 are disjoint.
3. *Prefix-incompatibility for downward paths.* For two downward-only paths, if the prefixes of child steps are incompatible then the paths are disjoint. For example, $/a/b/c//*//d$ and $/a/b/e//*$ are prefix-incompatible.
   Although this is a special case that is also solvable by Hammerschmidt et al.'s algorithm, this check is faster than their automaton construction and detects many common cases where the paths are disjoint.

If the paths are downward-only and their disjointness is not resolved by the heuristics, we employ the automaton construction in Hammerschmidt et al. [20]. Otherwise, we attempt the approaches via SMT-solving described in the body of the paper.

D. ADDITIONAL RELATED AND FUTURE WORK

Although view maintenance for XML queries has been explored in much prior work (see for example [7, 12]) the focus there has been the use of data structures that aid efficient recomputation when the view must be refreshed. Fast static techniques can effectively complement incremental techniques, especially in distributed settings, as argued here and in [27, 2].

Our query-rewriting algorithms for computing destabilizers are reminiscent of algebraic approaches to view-maintenance, e.g. those based on finite differencing [25, 19]. However, there is no formal connection between these problems.

Disjointness testing for XPath/XQuery is not well understood either in theory or practice. Hidders [21] established that XPath satisfiability is in PTIME, and intersection testing is NP-hard, but the complexity of binary intersection of simple paths containing arbitrary axes remains open. Hammerschmidt et. al. [20] showed that it is in PTIME for downward-only paths, while Benedikt et al. [4] show that even satisfiability (and hence binary intersection) of downward paths is NP-hard in the presence of a schema. Genevès and Layaïda [16] present a translation from XPath containment problems to $MSO(Tree)$. Our translation to `mona` is similar. Subsequently, Genevès, Layaïda and Schmitt [17] have developed a system for analyzing XPath expressions with respect to schemas. Our approach to intersection testing using `yices` appears more efficient than their prototype but only solves a special case of this problem.

One of our main conceptual contributions is the idea of finitely representing collections of updates that can change a query result. This is motivated by prior work on XML projection [24, 6], which concentrates on sets of nodes as a representation system. We explore a more precise representation system here via the use of op-based destabilizers, but we intend to explore much richer systems in the future. For example, one can gain precision by utilizing descriptions that alternate conjunction and disjunction, stating a sequence to *simultaneously* overlap with several sets of atomic updates. In Example 2, a finer representation system would allow us to specify that for an update sequence to change Q_2, it must either replace node 1, or it must both modify nodes 2 or 3 and also modify one of nodes 4 and 5. While we use these representation systems in conjunction with static analysis, they could also be used at runtime – either in the context of incremental view maintenance, or simply as a way of understanding how an anomalous query result could be changed.

E. BENCHMARK QUERIES AND UPDATES

The XMark queries are written in plain XQuery using complex XPath expressions and other high-level constructs, and as such need to be translated to the core language before static analysis. We translated them by hand following the rules in the XQuery standard. We also inlined let-binding expressions (this is not always safe in full XQuery but appears safe for the XMark queries). Finally, XMark queries also use several features that we do not directly handle, such as built-in functions (`count` or equality). We translated these queries to queries that do not use these features, but have similar dependencies. Thus, for a query such as Q10:

```
for $b in $auction//site/regions
return count($b//item)
```

we used the simplified query:

```
for $b in $auction//site/regions
return $b//item
```

The XPathMark queries were used as the basis for updates derived in a systematic way. For example, XPathMark query A_2 is:

```
$auction//closed_auction//keyword
```

and from this we generated an insert:

```
insert nodes <foo/>
into $auction//closed_auction//keyword
```