XML Query-Update Independence Analysis Revisited

Muhammad Junedi
Inria/LIG
655 avenue de l'Europe,
38334 Saint Ismier, France
muhammad.junedi@inria.fr

Pierre Genevès
CNRS and Inria/LIG
655 avenue de l'Europe,
38334 Saint Ismier, France
pierre.geneves@inria.fr

Nabil Layaïda
Inria/LIG
655 avenue de l'Europe,
38334 Saint Ismier, France
nabil.layaida@inria.fr

ABSTRACT

XML transformations can be resource-costly in particular when applied to very large XML documents and document sets. Those transformations usually involve lots of XPath queries and may not need to be entirely re-executed following an update of the input document. In this context, a given query is said to be independent of a given update if, for any XML document, the results of the query are not affected by the update. We revisit Benedikt and Cheney's framework for query-update independence analysis and show that performance can be drastically enhanced, contradicting their initial claims. The essence of our approach and results resides in the use of an appropriate logic, to which queries and updates are both succinctly translated. Compared to previous approaches, ours is more expressive from a theoretical point of view, equally accurate, and more efficient in practice. We illustrate this through practical experiments and comparative figures.

Categories and Subject Descriptors

H.2.3 [**Database Management**]: Languages—*Query languages*

Keywords

XML, Query, Update, Independence

1. INTRODUCTION

XQuery is becoming increasingly popular as a transformation and query language for XML documents. XML Update Language extends the syntax and semantics of XQuery to provide update features, i.e. side effects on documents (for example insertion and deletion of nodes).

XML query-update independence analysis consists in statically checking whether the results of a given query are affected by an update. Determining independence has many important applications which makes it an important line of research: avoiding view re-materialization, concurrency control and transaction management optimization, access policy enforcement, query engine optimization, web page consistency and others. The fact that independence detection is undecidable in general for XQuery and XML Update Language raises the challenge of finding a compromise between the accuracy and efficiency of static analysis and trying to support the largest fragments of query and update languages.

The static detection of query-update independence usually follows a two-step approach. As a first step, a common representation of queries and updates should be established in order to analyze them jointly. This operation usually relies on the extraction of path expressions that model both queries and updates. As a second step, once the common representation is obtained, independence can be checked using some intersection testing techniques on these paths. Two aspects are key for the accuracy and efficiency of the whole process:

- the adopted common representation, as its expressiveness (e.g. being able to capture schemas) has a direct effect on the accuracy, and its succinctness has a direct impact on efficiency;
- the chosen intersection testing technique, as its theoretical complexity and algorithmic effectiveness directly affect the performance of the analyzer.

In [2], Benedikt and Cheney provide a generic framework for independence analysis based on destabilizers and compare different approaches for intersection analysis. The clear separation of concerns in [2] between the representation problem and the problem of intersection analysis provides a good environment for comparison between different independence analysis approaches in addition to the genericity of the framework that can handle any update and query language.

Contribution

In this paper, we revisit query-update independence analysis as proposed by Benedikt and Cheney [2] under the light of the latest developments in intersection testing through a satisfiability checker for an expressive logic, namely the μ-calculus.

We provide evidence that our approach is more expressive in theory, equally accurate, and provides better performance compared to the various alternatives suggested in [2]. We provide experimental results that back our claims and invalidates Benedikt and Cheney's claims that using such an approach is not competitive.

2. XML QUERY-UPDATE ANALYSIS

In this section we recall the XML query-update framework introduced by Benedikt and Cheney [2] that we have reimplemented as a prerequisite for developing our approach.

The framework of [2] relies on a common representation called selection queries for both updates and queries. Specifically, the query is represented as the set of updates that can change the result of that query, which is called the query *destabilizer* (i.e. the set of nodes that can destabilize the query if they are a target of some update).

Updates are represented by a description of the target nodes that are modified. Then, different intersection analysis techniques can be applied in order to check that none of the target nodes are in the query's destabilizer.

The XML query language supported by [2] is an XQuery subset called XQ and the supported update language is a part of snapshot semantics-based XQuery Update Facility [5]. XML documents are considered as tree structures. The reader is referred to [2] for full details on the syntax and semantics of XQ and XML Updates.

To illustrate destabilizers, we consider as an example the XPath query $Q1 = \$doc/C$, \$doc refers to the document root. The result of Q1 can be affected in different manners:

- the subtree value of the result of Q1 is destabilized if update operations (i.e. insert, delete, replace and rename operations) have a target node in $\{\$doc, \$doc/C, \$doc/C/descendant\text{-}or\text{-}self::*\}$. This set is called the value destabilizer of \$doc/C query, which is abbreviated as $\Delta^{v}_{*}(\$doc/C)$;

- the boolean result of Q1 can be changed from nonempty to empty if update operations have a target node in {\$doc, \$doc/C}. This set is called the negative boolean destabilizer and is abbreviated as $\Delta^{b-}_{*}(\$doc/C)$;

- the boolean result of Q1 can be changed from empty to nonempty if update operations have a target node in {\$doc, \$doc/child::*}. This set is called the positive boolean destabilizer and is abbreviated as $\Delta^{b+}_{*}(\$doc/C)$;

- the node structure of the result of Q1 is destabilized if update operations have a target node in {\$doc, \$doc/child::*}. This set is called a node destabilizer and is abbreviated as $\Delta^{n}_{*}(\$doc/C)$.

The previous example explains the four types of generic destabilizers. To improve accuracy, an update sensitive version of destabilizers is proposed which takes only the update in question into consideration. For example no insert operation can change the results of a query from nonempty to empty. Similarly, no delete operation can change the results of a query from empty to nonempty. Moreover, only a renaming operation on \$doc/child::* can destabilize the node structure of Q1, which is an example of a rename sensitive node destabilizer: $\Delta^{n}_{rename}(\$doc/C) = \{\$doc/child::*\}$ which is more accurate than including \$doc also in node destabilizer set.

These four types of generic and operation sensitive destabilizers are used in the query rewriting algorithm (from an XQ query to a destabilizer) as four mutually recursive functions on the structure of the query. The resulting query belongs to a subset of XQ called *SelXQ* (Selection Query) that excludes from XQ the rule for output generation. The rewriting algorithm for generic and operation sensitive destabilizers can be found in [2].

While queries are transformed into destabilizers, updates are transformed into the selection query which represents the targets of update operation. The transformation function *Targ* is defined in [2].

After representing the queries as destabilizers and updates as target queries, a function can translate this unified intermediate representation into a target logical language depending on the chosen intersection testing technique.

3. INTERSECTION TESTING

Intersection testing uses different static analysis techniques to find the intersection between two structured representations that correspond to a query and an update.

The simplest form of intersection testing for XPath sets is by using heuristics: suffix incompatibility, displacement tests and prefix incompatibility for downward paths [2]. These techniques have a polynomial time complexity but they work only for a very restricted fragment of downward XPaths.

For downward XPath fragments with child, descendant and wildcard fragments, intersection can be solved in polynomial time by building an automaton for XPath expressions and finding the intersection using product automata and emptiness check [14]. This solution is accurate but obviously of very limited expressiveness.

To handle more expressive fragments, [2] proposes to use the Satisfiability Modulo Theory (SMT) approach for intersection analysis using the theory of linear order (N,<). This is done by first transforming selection queries to positive existential first order logic (over trees) formulas. Then these formulas are translated using interval encoding [6] and passed to an SMT solver such as Yices [7] to check for independence.

To handle all the fragments of SelXQ queries:

- Selection queries can be transformed into first-order formulas over trees for child, descendant and sibling relations [3] and intersection analysis reduces to satisfiability which has a non-elementary lower bound for time complexity [17]. These formulas can be checked for disjointness by monadic second order (MSO) satisfiability solvers, like MONA [15, 10];

- Selection queries can be transformed into a set of XPath expressions which in turn are translated into μ-calculus formulas. The conjunction of these formulas can be checked for satisfiability using a μ-calculus satisfiability solver [12, 11].

4. COMPARATIVE STUDY

We now compare the three mentioned approaches (SMT Solver, MSO Solver and μ-Solver) first from a theoretical perspective, and then from a practical point of view.

Theoretical aspects

Figure 1 illustrates the whole approach and each alternative for performing the intersection test. Vertical arrows correspond to translations which turn a given expression from a source language into an equivalent expression in another target language. The figure also summarizes the theoretical characteristics of each alternative regarding expressivity, accuracy, and complexity.

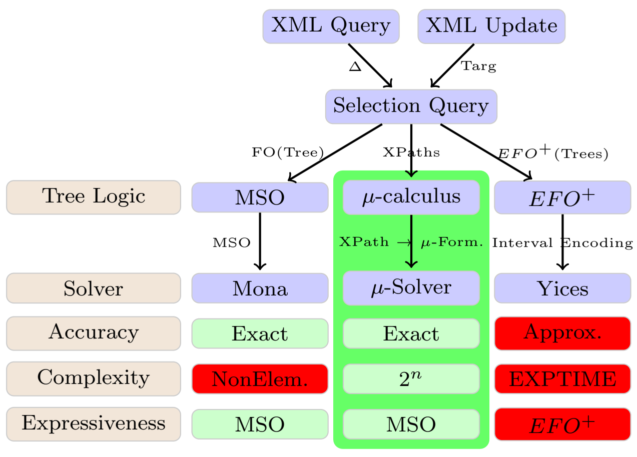

Figure 1: Independence Analysis Framework

The theoretical complexity of satisfiability on monadic second order formulas is known to be non-elementary whereas it is exponential for simplex based linear arithmetic satisfiability. The complexity of satisfiability on μ-formulas is simple exponential 2^n.

Reduction to Satisfiability modulo order theory requires positive existential first order formulas to be translated into constraints; which is not sufficient to represent negation in qualifiers for example. Mondaic second order logic is equal in expressiveness to μ-logic [9].

Practical aspects

We have implemented the destabilizer approach, in Java, to evaluate intersection analysis using the μ-solver. We compared our results with those obtained in [2]. The input to the analysis method consists in an XQuery expression and an XQuery Update Facility update.

The analyzer first transforms the update expression into its corresponding target selection query using *Targ* method. Then, it transforms queries into query destabilizers using the function Δ^v_* which is called on the query expression in the case of a generic destabilizer (or $\cup_{op}(\Delta^v_{op})$ in the case of an operation sensitive destabilizer where op is the set of update operations detected in the input update expression).

Then all XPath expressions are extracted from the query destabilizer and the update target query. Finally, they are passed to the μ-solver that checks the satisfiability of the formula $QueryXPaths \wedge UpdateXPaths$, where QueryXPaths is the disjunction of the members of the set of XPaths extracted from query destabilizer and UpdateXPaths is the disjunction of the members of the set of XPaths extracted from the update target query.

For example, the following query Q and update U:

Q = ***for*** *\$x* ***in*** *\$doc/C* ***return*** *<A>\$x/D</A>*
U = ***delete*** *\$doc/A/B*

are independent because none of the extracted path expressions of value destabilizer of Q intersects with the target XPath expressions of U. The following steps denote the whole execution process:

1. The target query is executed on the update U: $Targ_*$ (**delete** \$doc/A/B) = (\$doc/A/B);
2. The Destabilizers rules are applied on the query Q: Δ^v_*(**for** \$x **in** \$doc/C **return** <A>\$x/D</A>)= (\$doc, \$doc/child::*), **for** \$x **in** \$doc/C **return** (\$x/desc-or-self::*, \$x/child::*/desc-or-self::*)
3. XPath expressions are extracted from the target query and the result is the set {/child::A/child::B}
4. XPath expressions are extracted from destabilizers and the result is { /self::*, /child::*, /child::C, /child::C/desc-or-self::*, /child::C/child::*/desc-or-self::* }
5. The two sets are passed to the μ-solver that indicates the unsatisfiability.

On the opposite, Q and U'= **delete** \$doc/C/D/E are not independent because the XPath expression /C/D/E will intersect with /C/D/descendant-or-self::*.

We used the exact same set of 20 Query examples of XMark benchmark [16] used by Benedikt and Cheney in their experiments, with the same necessary modifications to fit in the supported query fragments. We automatically generated 4 kinds of updates (rename, insert into, insert before, delete)[1] on the 16 XPath queries (A1-A8, B1-B8) of XPathMark [8].

We applied this analysis on all the combinations of queries and updates and we compared the mean execution time of the μ-solver with the results found in [2] for the Yices SMT solver.

We used the version of the μ-solver implemented in Java (using JavaBDDs) and shared with Benedikt and Cheney. Experiments are carried out on a laptop with an Intel Processor (3.0 GHz), with 2GB of memory.

Figure 2 compares the running times of the μ-Solver and the SMT-solver. The running times of the SMT-solver (Yices) are taken from [2]. The running times of the μ-solver carefully followed the method for calculating running times found in [2]. Specifically, each update is tested for independence using the generic analysis with the twenty queries of the XMark benchmark and then the mean time is taken and the results are grouped by the update.

Efficiency. Figures 2 and 3 show comparisons between the execution times obtained with the μ-solver with those obtained with the MONA solver and the Yices solver. [2] finds that Yices is the most efficient in practice, MONA follows and they do not give measurements for the μ-solver whose performance was judged not attractive. Our measurements prove that the μ-solver approach is far more efficient in practice compared to Yices (and obviously to MONA).

An interesting finding was revealed when looking carefully at the time decomposition of the resolution process. This process involves various aspects such as constructing destabilizers, parsing formulas, initializing the BDD library, and the actual resolution time within the solver. It turned out that if we concentrate on the μ-solver time (see Figure 4), leaving aside the other costs, we observe that time spent in the resolution procedure is negligible compared to the overall figures presented in [2]. This gives a clear evidence (somehow counter-intuitive compared to the conclusions of [2]) that the dominant cost do not necessarily reside in the resolution procedure. We believe that times spent in the resolution procedure better reflect the intrinsic difficulty of each problem instance.

[1]these four updates are sufficient to cover different destabilizer rules

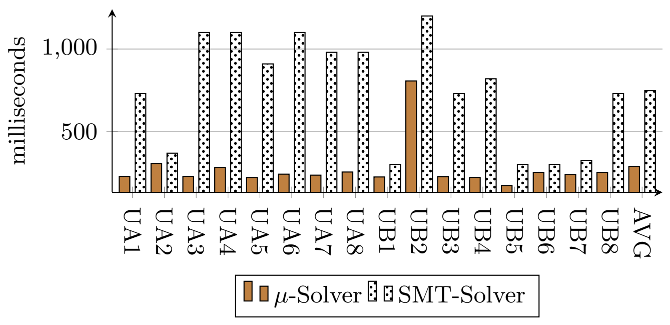

Figure 2: μ-solver vs. SMT-solver running times.

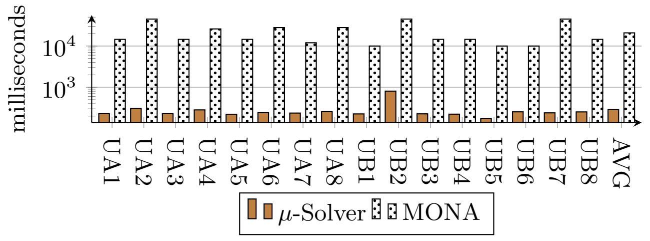

Figure 3: μ-solver vs. MONA running times [logarithmic scale].

Accuracy. Both MONA and the μ-solver approaches can determine exactly whether two selection queries overlap (because they both have the expressivity of MSO, notably encompassing First-Order logic, FO) whereas Positive Existential First-Order logic (EFO^+) can return false negatives (since EFO^+ is strictly less expressive than FO and thus obviously strictly less expressive than MSO).

Providing MSO expressivity is an unquestionable advantage as this allows capturing regular tree languages (XML schema languages) for which FO expressivity is insufficient.

5. RELATED WORK

The notion of destabilizers introduced by Benedikt and Cheney [2] was partly inspired by previous works on XML projection [4], where the goal is to identify nodes that can be deleted without modifying the result of a query (this corresponds to independence problems involving deletion only). As noticed in [2], prior techniques either required a schema [1, 4] or apply only to downward fragments of XPath. [13] proposes a conservative analysis for an XML update language and a theorem that can be used to identify commuting expressions. However a different update language proposal is considered and independence is not addressed.

6. CONCLUSION

We have revisited the XML query-update independence analysis problem by exploring in depth the approach based on the μ-solver. For that purpose, we have reimplemented a destabilizer's framework for evaluating the relevance of this approach which was neglected in previous studies. We show that performance can be drastically enhanced using the μ-solver based approach. Surprisingly, our results show that the best performing approach in practice does not require any loss of precision. This questions systematic search for trade-offs between expressivity and precision.

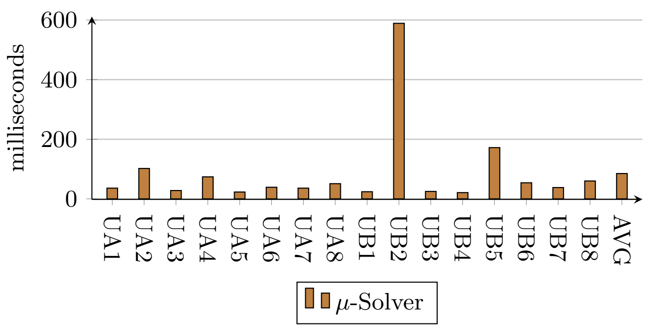

Figure 4: Actual resolution times for the μ-Solver.

7. REFERENCES

[1] Michael Benedikt and James Cheney. Schema-based independence analysis for XML updates. *Proc. VLDB Endow.*, 2(1):61–72, August 2009.

[2] Michael Benedikt and James Cheney. Destabilizers and independence of XML updates. *Proc. VLDB Endow.*, 3(1-2):906–917, September 2010.

[3] Michael Benedikt and Christoph Koch. From XQuery to relational logics. *TODS*, 34(4):25:1–25:48, 2009.

[4] Véronique Benzaken, Giuseppe Castagna, Dario Colazzo, and Kim Nguyên. Type-based XML projection. In *VLDB'06*, pages 271–282, 2006.

[5] Don Chamberlin, Daniela Florescu, and Jonathan Robie. XQuery Update Facility. W3C WD, 2006.

[6] David Dehaan, David Toman, Mariano P. Consens, and M. Tamer Özsu. A comprehensive XQuery to SQL translation using dynamic interval encoding, 2003.

[7] Bruno Dutertre and Leonardo De Moura. The Yices SMT solver. Technical report, 2006.

[8] Massimo Franceschet. XPathMark: an XPath benchmark for the XMark generated data. In *XSym'05*.

[9] Pierre Genevès. *Logics for XML*. PhD thesis, Institut National Polytechnique de Grenoble, December 2006.

[10] Pierre Genevès and Nabil Layaïda. Deciding XPath containment with MSO. *Data Knowl. Eng.*, 63(1):108–136, October 2007.

[11] Pierre Genevès and Nabil Layaïda. XML reasoning solver user manual. R. Report 6726, INRIA, 2008.

[12] Pierre Genevès, Nabil Layaïda, and Alan Schmitt. Efficient static analysis of XML paths and types. In *PLDI '07*, pages 342–351, 2007.

[13] Giorgio Ghelli, Kristoffer Rose, and Jérôme Siméon. Commutativity analysis for XML updates. *ACM TODS*, 33(4):29:1–29:47, December 2008.

[14] Beda Christoph Hammerschmidt, Martin Kempa, and Volker Linnemann. On the intersection of XPath expressions. In *IDEAS'05*, pages 49–57, 2005.

[15] Nils Klarlund and Anders Møller. *MONA Version 1.4 User Manual*. BRICS, January 2001.

[16] Albrecht Schmidt, Florian Waas, Martin Kersten, Michael J. Carey, Ioana Manolescu, and Ralph Busse. XMark: a benchmark for XML data management. In *VLDB'02*, pages 974–985, 2002.

[17] Sergei G. Vorobyov. An improved lower bound for the elementary theories of trees. In *CADE*, pages 275–287, 1996.