OUCL Information Systems Group

# Static Analysis of Declarative Update Languages

Joint work with James Cheney
(includes DBPL2009, VLDB2009,
and current work)

# Outline

- What is this about? Why bother?
- Who has worked on it? Did they get anywhere?
- Discussion of independence/Interaction analysis
    - Unique to update setting
    - Connections to provenance
- Our approach to independence analysis
    - System description and results
- Future directions

# What Do You Mean by Analysis?

Explanation of what an XML update expression produces.
E.g. type-inference:
Update U when applied to DTD $D_0$ produces documents satisfying $D_1$ .

Answering questions about what one or more update expressions can do.
Can U invalidate the schema? Are $U_1$ and $U_2$ the same?

This kind of problem has been studied to death for XML queries. What's new with updates?

Old questions bec  for the new language.

Question about a program

Can this program update this part of the document?

"classical static question"

Intersection of regular languages. Constraint satisfaction problem.

Answer

# What Do You Mean by Analysis?

This has been studied to death for XML queries. What's new with updates?

New questions:

Updates are created in order to do things incrementally.
Update should change only part of the document.
Analysis could tell what parts of the document can change.

- Result understanding.
  - Really want to know the effect of an update – not captured by a type

obvious application to view maintenance: want to know if nothing relevant to a view changes)

# Prior Work

| Language | Authors | Problems |
|---|---|---|
| Simple Updates: {Insert/Delete/Replace} XPath | Schmueli Raghavachari Byun/Yun/Park | "Conflict" |
| Functional Updates | Cheney | Type Checking |
| Snapshot Updates | B. & Cheney | Type Inference Conflict |
| Iterative Updates | B., Bonifati, Flesca, Vyas | Conversion to Snapshot |
| Flexible Updates (Snapshot+Iterative) | Ghelli, Rose, Simeon | Commutativity |

**Snapshot:** all queries performed on input document.

**Iterative:** For $x in E  P'($x) → Evaluate E, iterate over it, performing P'  each time.

# Independence Analysis

- Data migration scenario
  - At periodic intervals a large dataset is to be upgraded/evolved. The dataset must satisfy several hundred integrity constraints.
    - Developers write integrity constraints $R_1$ ... $R_m$ update expressions $U_1$...$U_n$
    - Written independently
    - Integrity constraints must remain valid
  - At evolution time, the goal is to apply the rules in sequence.
- If updates don't interact with each other, can be applied in parallel.
- If updates and constraints don't interact, constraints do not need to be revalidated.

Focus on update/query interaction: is Q independent of U?
The same techniques apply to update/update commutativity.

# Query/Update Independence

The natural approach to such problems:
see whether "U updates anything that Q can read"

What "U updates" is usually unproblematic:
but what are the items that Q reads?

Several notions of "the nodes accessed by an XML query Q".
Mostly given syntactically.

First used by Marian and Simeon in VLDB 2003: given a document D, find a subdocument proj(Q,D) of D such that the Q returns same result on proj(Q,D) as on D: **"projection of Q on D"**

- Intuitively, proj(Q,D) contains the nodes "read by Q".
- used by B. , Bonifati, Flesca, Vyas for update analysis
- Variations (with significant differences) by Koch, Scherzinger et. al., Benzaken/Castagna/Colazzo/Nguyen, Ghelli, Rose, Simeon…

# Independence

Many notions of "accessed nodes".
Algorithmically defined or axiomatically-defined

Could be in terms of paths (Marian/Simeon, Ghelli/Rose/Simeon)
or in terms of schemas types (Benzaken, Castagna, Colazzo, Nguyen)

Several prior works on independence analysis consist of two components:

- A notion of "updated stuff/ accessed stuff" (paths, types) and an algorithm for
producing the accessed stuff"from a query and the updated stuff from an update expression.

- An intersection test.

In many case the intersection test has been fairly trivial.
E.g. for downward paths.

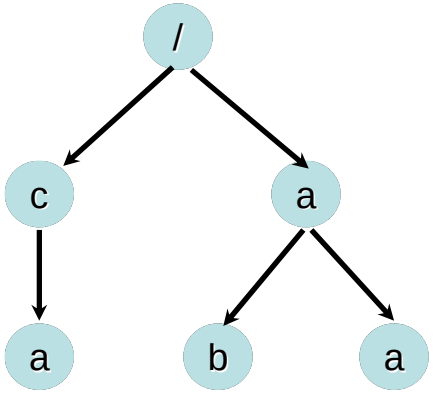
## w3c update language proposal: XQuery Update Facility

- Snapshot language: simpler “one-pass” implementation
- Analysis simpler than for “iterative” languages
- Certain kinds of errors cannot occur
- Counterintuitive (?) semantics

```
delete $x//a,
insert <foo>bar</foo>
  before $x//a
```

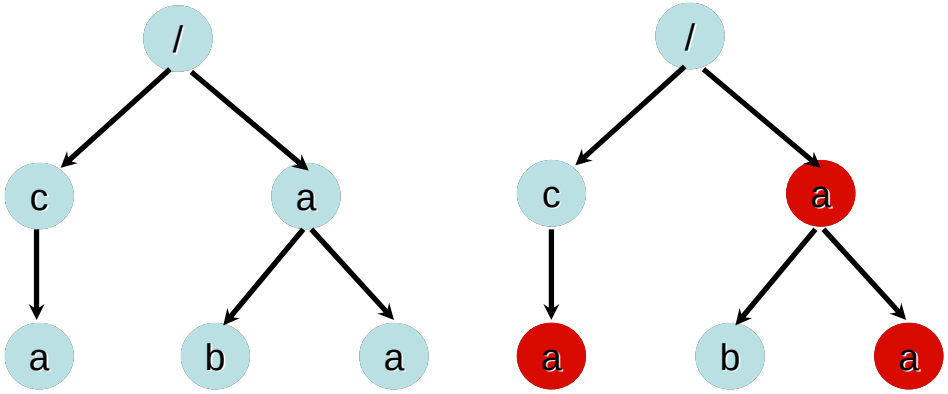
- does **not** do what you (probably) expect

# Example



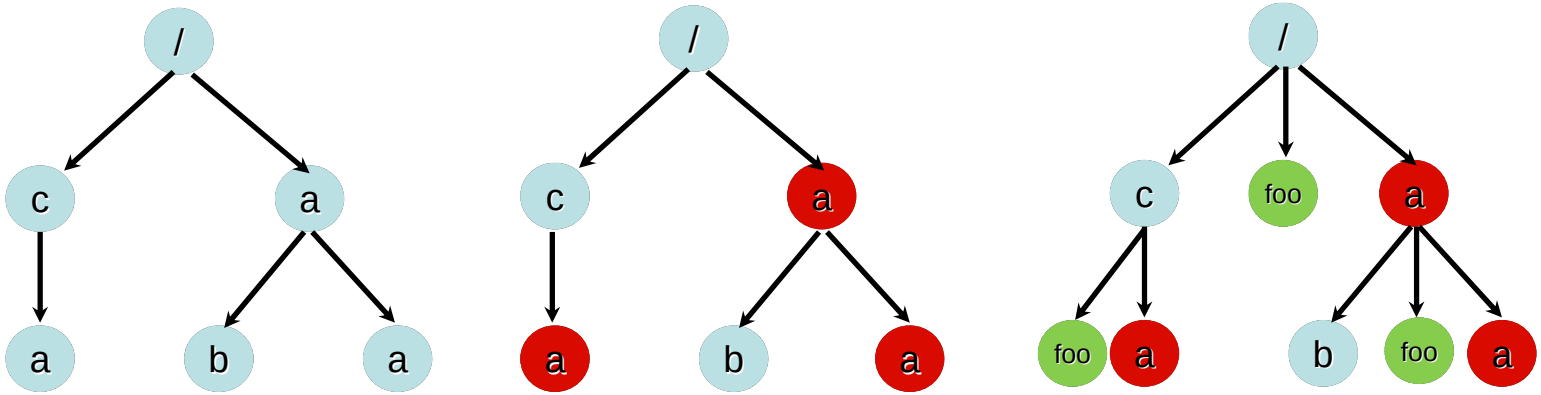

```
delete $x//a,
insert <foo>bar</foo>
  before $x//a
```

# First **collect** updates



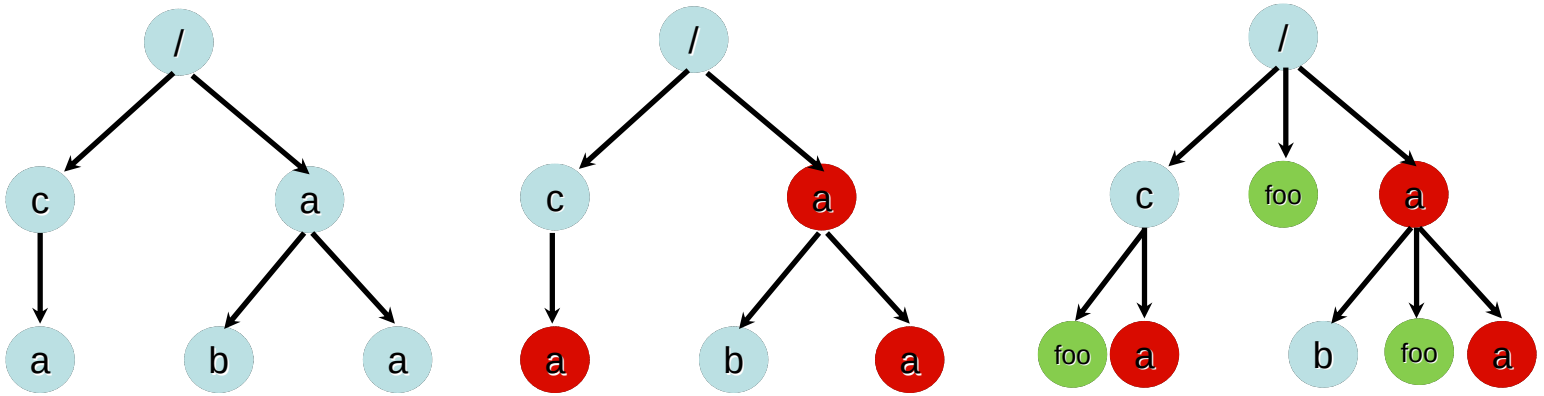

```
delete $x//a,
insert <foo>bar</foo>
  before $x//a
```

# First **collect** updates




```
delete $x//a,
insert <foo>bar</foo>
  before $x//a
```

# Then **reorder** & **apply**




```
delete $x//a,
insert <foo>bar</foo>
  before $x//a
```

# Snapshot Languages – theoretical perspective

Core Atomic XQuery – XQuery without aggregate features or "value equality"
Nav. XQuery: as above, but fix the tag alphabet.
Core XUpdate = w3c proposal where queries come from Core Atomic XQuery.
Navigational XUpdate – as above but where only fixed tags are allowed.

Easy to show that Core/Nav XUpdate is more expressive than Core/Nav XQuery.
But not for boolean queries.

Theorem:
For boolean queries, Core XUpdate has same expressiveness as Core XQuery = first-order logic over ordered data trees.

These equivalences are effective (but never implemented).
Can be used to show that questions about Nav. XUpdate are decidable in principle.

# Snapshot Languages – theoretical perspective

Independence analysis for XQuery, XUpdate
is closely related to the equivalence problem for these languages.

Several notions of equivalence of queries.

$\cong_{bool}$ both results return nonempty or both empty (boolean semantics of queries)
$\cong_{node}$ nodeids in result sequences are isomorphic over the input model
$\cong_{subtree}$ subtrees in result sequences are isomorphic over the input model

Independence of U and Q can be defined as
$\forall$ D Q(U(D)) $\cong$ Q(D) for any of the above equivalences.

Solvable if we could solve the equivalence problem for compositions of queries and updates

# Independence Analysis

Closely related to the problem of Core XQuery Equivalence

Theorem For $\cong_{bool}$ , static equivalence of Navigational XQuery and Navigational XUpdate is decidable but non-elementary.

From this, can show that independence is decidable.

Uses translation of XUpdate into FO, extending that for Core XQuery in B./Koch TODS 2009.

# Independence Analysis

Closely related to the problem of Core XQuery Equivalence

Theorem For $\cong_{node}$ and $\cong_{subtree}$ decidability of query equivalence and update equivalence is an open question.
Decidability of independence for these equivalence relations is also open.

Some special cases of equivalence is ok;
• “Linear Navigational XQuery” – only increase the size of the DB linearly (to syntactically enforce, allow no nesting of For loops)
• “Linear Navigational Updates” --- no Insert/Replace lies within the scope of two For Loops
From results of Maneth and Engelfriet, it follows that equivalence of Linear Updates is decidable. Using this we can show that independence for Linear XQuery queries and Linear Updates is decidable.

# Approaches to Independence for XQuery Update Facility

For Navigational XQuery/XUpdate, boolean equivalence have an "exact approach": translate updates and queries to logic, and test equivalence.

Only works for these restricted languages, ridiculous worst-case complexity.

Approximate approach based on Projection + Intersection.
Utilized in B. and Cheney, VLDB 2009, assuming a schema.
- Get a set of schema types representing the "nodes read by the query"
- Get another set of schema types representing the "nodes updated by the update expression."
- See if the two sets can overlap.

Very similar to the approach taken for Iterative Updates by Ghelli, Rose, Simeon

Here we will outline a more general approach, which links the independence problem with provenance.

# Destabilizers: A Framework For Relating Updates and Query Changes

Fix
- "any data model", consisting of query inputs and query outputs
- "any query language" QL
- "any update language" UL
- Let $\cong$ be an equivalence relation on query outputs

Fix a query Q, data object D, and output object Q(D)=D'

The **destabilizer** of Q(D)=D' is the set of updates $u \in UL$ such that : $Q(u(D)) \ncong D'$

Of course, for any reasonable update language, there will be infinitely many such updates.
We want a **finite representation** of this set.

# Destabilizer Framework

An destabilizer representation system RS is a collection of objects and an effective mapping associating each rep $\in$ RS to a collection of updates.

An **exact destabilizer** of Q(D)=D' is an object in RS representing the set of updates u such that Q(u(D)) $\not\cong$ D'

A **sound destabilizer** of Q(D)=D' is an object in RS representing a superset of the updates u such that Q(u(D)) $\not\cong$ D'

Would like an exact destabilizer, will settle for a sound one

# Destabilizers for XML queries

Data Model =XML documents along with variable environments
QL= XML Query Language. E.g. Core XQuery
Output objects= Query results (new document + node list)

UL= sequence of “concrete updates”:
delete nodeid, rename nodeid as a, insert T into nodeid as first,…

Possible values of $\cong$ are : $\cong_{bool}$ , $\cong_{node}$ , $\cong_{subtree}$

Talk about $Destab_{bool}(Q,D)$, $Destab_{node}(Q,D)$, $Destab_{subtree}(Q,D)$

# Representation Systems

How to represent infinite collections of update sequences?

"Node Destabilizer": an update sequence is abstracted by the set of nodes that are targets of some update in the sequence.

Unwinding the general definition to this setting:
A set of nodes S is a sound node destabilizer if every sequence of updates that modifies the result of Q on D contains a target in S

# Representation Systems

How to represent infinite collections of update sequences?

Node/Update Type Destabilizer: an update sequence is abstracted by the set of pairs (target node,update type) that are contained within it

Update type = **Insert into as first**, **Rename**, **Replace**, **Delete**, **Insert after …**
E.g. *(Insert into as first, $n_4$), (Replace, $n_6$)*

A set of node/type pairs S is a sound node-type destabilizer if every sequence of updates that modifies the result of Q on D contains an update with node type T and target n with (T,n) in S

# Representation Systems

How to represent infinite collections of update sequences?

Many others:

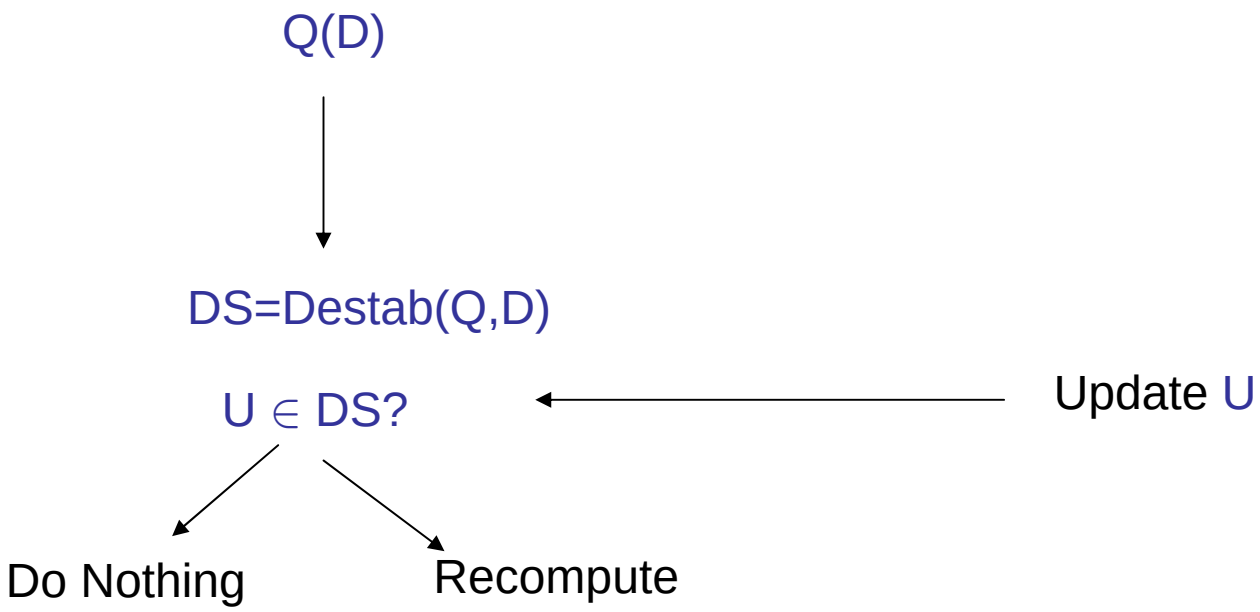
Sets of update nodes:
$(i_1,\ldots i_k)$ represents update sequences that contain updates that modify each of $i_1$ …. $i_k$

Insert A in nodeId, where A is a tree automaton describing the content of the insert

# What good is a destabilizer?

Could be useful for understanding query results?

Can be useful for maintaining query at runtime.



# Negative Results

Theorem:

- Even for simple tree pattern queries, one cannot calculate an exact destabilizer in either the node or node/type representations.
- Calculating a minimal sound destabilizer in either of them is NP-hard even for very simple query languages. It is non-elementary for Core XQuery.

Best one can do is calculating a sound destabilizer that is (heuristically) pretty good.
Of course, one can always calculate a trivial sound destabilizer:
e.g. all nodes.

# Destabilizer Algorithms

Calculate Inductively – all inductive cases are indifferent to the type of destabilizer used!

XQuery control constructs:
$Destab_*((Q_1 , Q_2), D) = Destab_*(Q_1,D) , Destab_*(Q_2,D)$

Inductive cases are sensitive to the equivalence relation used:

$Q = \text{if } Q_1 \text{ then } Q_2 \text{ else } Q_3$

$Destab_{node}(Q,D) = Destab_{bool}(Q_1,D)$
$= \text{if } Q_1 \text{ then } Destab_{node}(Q_2,D) \text{ else } Destab_{node}(Q_3,D)$

# Destabilizer Algorithms

Q=<A> Q' </A>

$Destab_{bool}(Q,D)$ = ??

∅

$Destab_{node}(Q,D)$ =
$Destab_{subtree}(Q,D) = Destab_{subtree}(Q',D)$

# Destabilizer Algorithms: base cases

Q=$x/axis::A
Focus on
Q=$x/child::A

Here depends heavily on the representation system.

For node-based destabilizer:
$Destab_{node}(Q,D)= \$x(D) \cup \$x/child::*(D)$

For node/type-based destabilizer

$Destab_{node}(Q,D)=$ (delete, $x(D)), (rename, $x/child::*(D))…
…

# Destabilizers and Query Rewriting

Destabilizer based on the previous two representation systems can be implemented by query rewriting.

In the case of node destabilizer, get a single query.

For node based destabilizer: Q=$x/child::A
$Destab_{node}(Q)$= $x ∪ $x/child::*

In the case of node/type destabilizer, need a query for each operation type op, returning the nodes n such that (n,op) is in the destabilizer

# Query-rewriting

There are several motivations for destabilizers
- result understanding
- runtime view maintenance:

determine when an update sequence comes in whether there is some need to recompute the query. By pre-computing the destabilizer, get a linear time test
- static analysis

Given an update program, determine whether it can possibly generate an update that modifies the query result; if answer is no, can eliminate runtime checks.

Being able to calculate destabilizer via query-rewriting has particular advantages for the third application

# Destabilizers and Independence Analysis

Compile Time

Q ⟶ Query $Destab_Q$

U ⟶ Query $Targ_U$

Runtime:

D ⟶ ($Destab_Q$) ⟶ $Destab(Q,D)$ representing destabilizing updates

$Targ_u$ ⟶ Representation of generated updates

Analysis time: Check if $Targ_U$ can overlap with $Detab_Q$

# Independence Analysis: a pragmatic approach

Q=$doc/child::C

UP=Delete $doc/child::A/child::E

Given update program UP, get a query that represents the targets:

E.g. Delete $doc/child::A/child::E
→ $doc/child::A/child::E

Get destabilizer of Q:

For node destabilizer:
$doc/child::C → $doc ∪ $doc/child::*

If these two queries cannot overlap, then U and Q are independent

For Node/Type destabilizers, get something more accuracy than above.

# Independence Analysis and Query Rewriting

Can think of an independence analysis as consisting of consisting of two components.

Just described →

- A destabilizer representation and an algorithm for producing a representation via query-rewriting.

Obvious approach for w3c language →

- An algorithm for producing a representation of the update sequences produced by from an update expression

Many Options →

- An intersection test.

# Intersection tests

Need to see if $\forall$ D $Q_1(D) \cap Q_2(D) = \emptyset$

Some prior approaches to this problem:

Translate queries into logical expressions, use a satisfiability test for the logic.

MONA, mu-calculus of ordered trees (Geneves &Layaida)

Automata-theoretic approaches: translate directly into tree or string automata. (B., Bonifati, Flesca, Vyas)

Abstract queries by downward paths, and then use a simple path overlap test

# Intersection tests

Need to see if $\forall$ D $Q_1(D) \cap Q_2(D) = \emptyset$

Translate queries into logical expressions, use a satisfiability test for the logic.

MONA, mu-calculus of ordered trees (Geneves &Layaida)

Automata-theoretic approaches: translate directly into tree or string automata. (B., Bonifati, Flesca, Vyas)

Abstract queries by downward paths, and then use a simple path overlap test

Abstract by existential first-order formulas, and then translate into a logic over the integers with orders. Use Satisfiability Modulo Order constraint solvers.

# Summary on Independence Analysis via Destabilizers

Unified approach to several update analyses
- Represent updates that can destabilize a query
- Useful for comprehending results, view maintenance, static analysis
- Many representations possible
    - trade off precision of representation for complexity of algorithm, ability to perform intersection tests
    - Allows one to track the places where precision is being lost in an algorithm: coarse representation, computation of representation, intersection test.
- Can work with a schema or without

# Future Challenges

- Extending from the core: True scalability to XQuery/Update Facility features.
- Dealing with data values, aggregation.
    - Need to understand how to deal with these in core uery analysis (e.g. intersection tests).
- Integration with optimization frameworks of update processors

# Challenges II

- Integration with other approaches to view-maintenance
- Analysis of more powerful languages.
    - Iterative features
- Commutativity analysis of updates *in the presence of a schema*
- Benchmarks