Incremental Maintenance of Materialized XML Views

Leonidas Fegaras

University of Texas at Arlington

The Incremental View Maintenance Problem

- Given:
  1) A relational database (RDB)
  2) A materialized view stored in a native XML database (XDB)
  3) An SQL update over RDB
- Goal:
  - Derive an update to XDB so that:

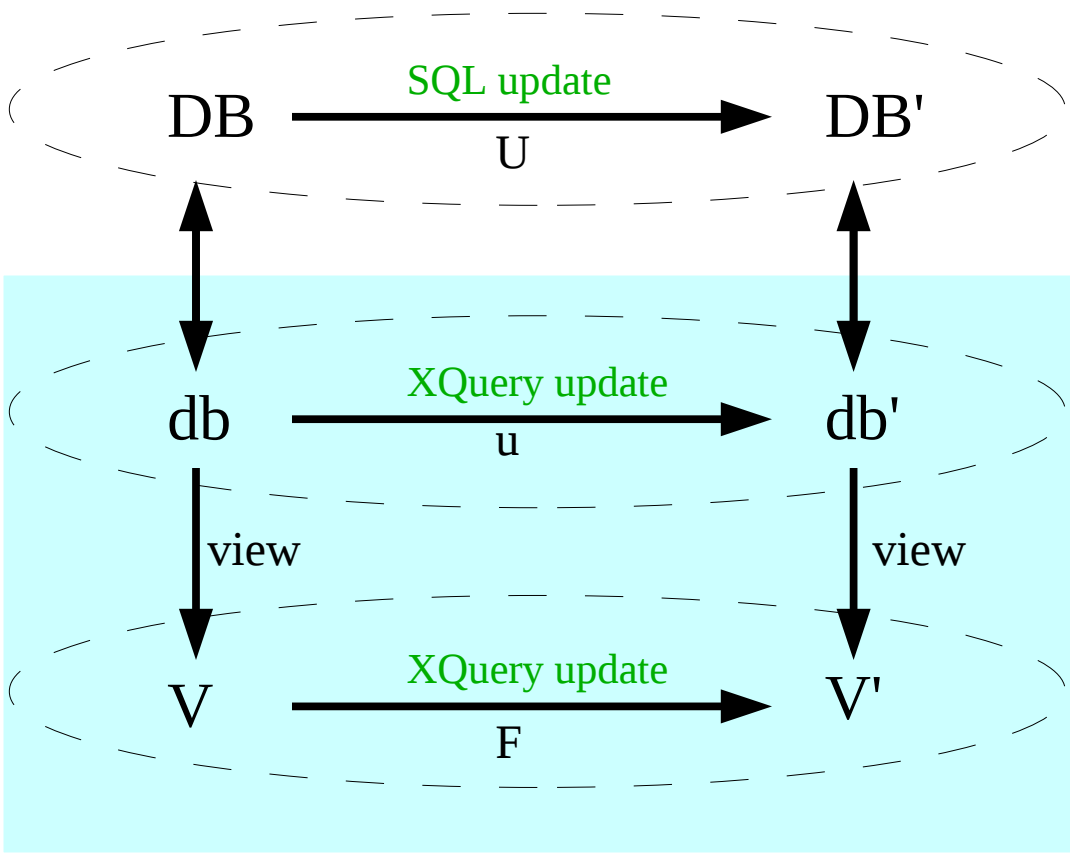
    updated XDB = materialized view over the updated RDB

- Naive approach:
  - Reconstruct the entire XDB from the updated RDB
- Incremental view maintenance:
  - Synthesize an XQuery update that incrementally updates XDB
- Requirement: XDB must support the XQuery Update Facility (XUF)

Why bother?

- Materialized XML views are very important for data integration
  - they speed up query processing
  - they provide a light-weight interface to data services, without exposing the main database
- Incremental maintenance of materialized views is important for applications that require freshness of view data, but it is too expensive to recompute the view data every time the source data change

Our Approach



relational DB

canonical XML view

materialized XML view

For a given view, we synthesize *a* right-inverse view' such that:

for all V: view(view'(V)) = V

Then: F(V) = view(u(view'(V)))

in general:
view'(view(db)) ≠ db

Steps

1) Map the RDB schema to a canonical XML schema (an isomorphic virtual view)
   - so that we can work on relational data using XQuery
2) express the SQL update in pure XQuery code that reconstructs the canonical XML view, reflecting the SQL update
3) synthesize *a* right-inverse of view, view'
4) optimize F(V) = view(u(view'(V))) to avoid the reconstruction of most parts of the intermediate data
   - result: a pure function from V to V' that recalculates the entire view to reflect the update
5) rewrite F(V) into efficient an XQuery update that destructively modifies the original view V to become equal to V'

Steps 3 & 5 are the most challenging

Is it a good idea?

- Basic idea:

  SQL update → pure function that reconstructs the entire view → XQuery update

- It uses source-to-source, compositional transformations only
  - can be built on top of existing XQuery engines
- It breaks view maintenance into a number of more manageable tasks that are easier to verify & implement
- It has the potential of becoming a general methodology for incremental view maintenance

Limitations:

1) it is schema-based
2) it uses heuristics

Synthesizing the Right-Inverse

- *Problem:* given f(x)=y, for two variables x and y and an XQuery expression f(x), find an XQuery expression g(y) such that:

  y = f(x) => x = g(y)

- Notation:

  I_x(e,y) is the left inverse of e with respect of x as a function of y

- We use a schema-guided algorithm
- May return an error $\perp$

  y = if (x>4) then x-1else x+2

  => x = if (y+1>4) then (if (y-2>4) then y+1 else $\perp$) else y-2

- Requires unification:

  let x be of type: element A {element B int, element C int}

  y = x/B => x = <A>{y/self::B,*}</A>

  y = x/C => x = <A>{*,y/self::C}</A>

  y = (x/C,x/B) => x = <A>{y[1]/self::B,y[2]/self::C}</A>

Synthesizing the Right-Inverse

- The inversion of XPath steps is schema-based:

  if the type of e is element B { ..., element A t, ... }

  I_x(e/A, y)

  = I_x(e, <B>{*,...,y/self::A,...,*}</B>)

- Inverting FLWOR loops

  I_x(for $v in e return u, y)

  = I_x(e, for $w in y return I_v(u,$w))

Example:

We want to find I_x(for $v in $x/A/data() return $v+1, y)

- I_v($v+1, $w) = $w-1
- I_x($x/A/data(), $z) = <B>{ *, ..., <A>{$z}</A>, ..., * }</B>

thus, I_x(for $v in $x/A/data() return $v+1, y)

= <B>{ *, ..., <A>{for $w in y return $w-1}</A>, ..., * }</B>

Synthesizing the Right-Inverse

- What if the FLWOR body refers to a non-local variable?
  - it happens in joins expressed as FLWOR nested loops
  - all solutions to a non-local variable $v are added to a global binding list as contributions
  - if there is already a contribution in the binding list, they are unified
- Equality predicates add contributions to be unified with the solution

Example:

I_x($x/person[pid=$a/id], y) introduces two new variables $z and $w

- unify y with I_z($z/pid, $w) and I_z($a/id, $w)
- return I_x($x/person, y)

Synthesizing the Right-Inverse

- Hard to invert sequences

$I_x((e_1,e_2), y)$

$= \text{unify}(I_x(e_1,y_1), I_x(e_2,y_2))$

where y is split to (y_1,y_2) in such a way that y_i has the same type as e_i

Example:

$I_x((x-1,x*2), y) = \text{unify}(I_x(y[1], x-1), I_x(y[2], x*2))$

$= \text{unify}(y[1]+1, y[2]/2)$

$= \text{if } (y[1]+1=y[2]/2) \text{ then } y[1]+1 \text{ else } \perp$

Synthesizing the XQuery Updates

- The XQuery F(V) reconstructs the view V, reflecting the base updates
  - in other words, F(V) is equal to V in all but the updated places
- *Goal:* we want to synthesize an efficient program (an XQuery update) that destructively modifies V to become F(V)
- Conservative approach:
  1) guided by the type of V, we synthesize the copy function that constructs a *deep* copy of V
  2) we recursively compare the code that constructs the copy of V with the F(V) code
     - if the corresponding components are equal, we don't generate any update
     - if they differ and are composite, we brake them into parts and recursively compare the associated parts
     - otherwise, we generate an XQuery update that updates this particular V part

Conclusion

- An effective method for incremental view maintenance
  - using pure source-to-source transformations that are easy to verify
    - instead of using a special algebra
  - does not require any change to the XQuery engine
- Implemented in Haskell
- Evaluated over a number of XML views over the DBLP data set
  - Compared view recreation vs. view incremental update time
  - Underlying RDB: MySQL
  - Materialize XML view: HXQ
    - an XQuery engine that supports XUF
    - it shreds and stores XML data into MySQL