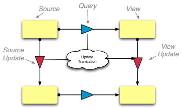
An Algebraic Approach to XQuery View Maintenance

J. Nathan Foster (Penn)
Ravi Konuru (IBM)
Jérôme Siméon (IBM)
Lionel Villard (IBM)



PLAN-X '08

Quick!

$$1 + 2 + \cdots + 99 + 100 = ???$$

Introduction

$$
\begin{aligned}
& 1 + 2 + \cdots + 99 + 100 \\
= \; & (1 + 100) + (2 + 99) + \ldots (50 + 51) \\
= \; & 101 \times 50 \\
= \; & 5050
\end{aligned}
$$

Introduction

$$
\begin{aligned}
& 1 + 2 + \cdots + 99 + 100 \\
= \; & (1 + 100) + (2 + 99) + \ldots (50 + 51) \\
= \; & 101 \times 50 \\
= \; & 5050
\end{aligned}
$$

Rewritings like this are often used to optimize the *initial* evaluation of a query.

But sometimes we want to *maintain* a view over a source that changes over time.

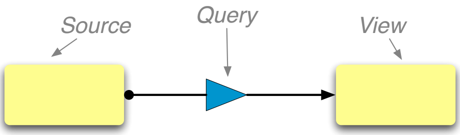
View Maintenance

$$(1+2+\cdots+99+100) \qquad = 5050$$

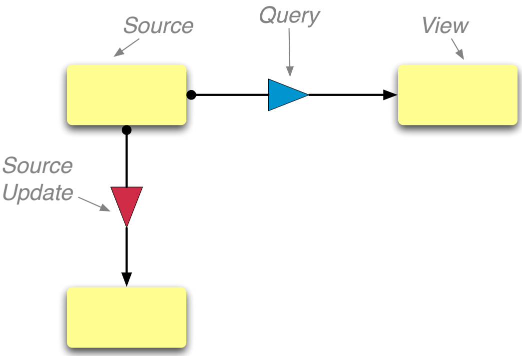
View Maintenance

$$(1+2+\cdots+99+100)-50 = 5050-50$$

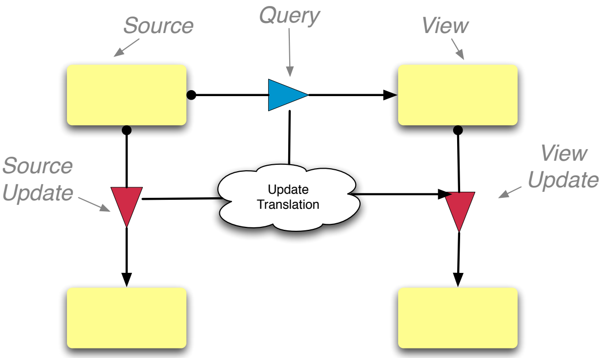
View Maintenance



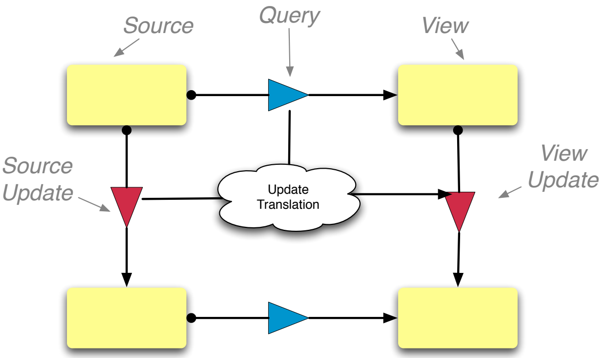
View Maintenance



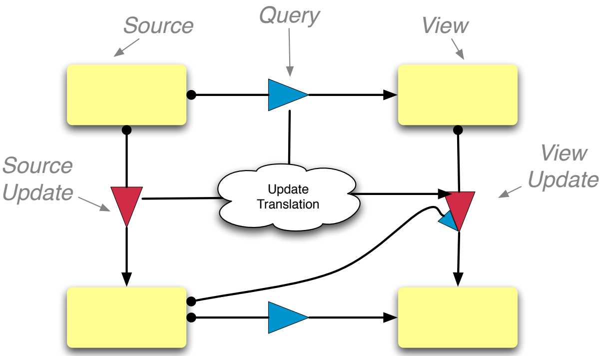
View Maintenance



View Maintenance



View Maintenance

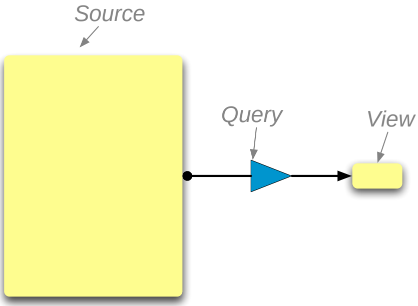


This talk: maintenance of views defined in XQuery.

Why Maintain?

Sometimes source is very large compared to the view:

- e.g., web page for a single item on eBay.

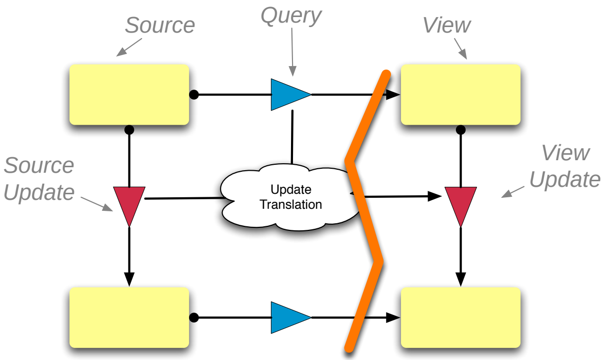


Many source updates are *irrelevant* to the view.

Why Maintain?

Sometimes view and source reside on different hosts:

- e.g., in an AJAX-style web application.



Cheaper to send an update than the whole view.

XQuery: Surface Syntax

XQuery: W3C-recommended query language

- XPath for navigation.
- FLWOR-blocks for iterating, pruning, grouping.

XQuery: Surface Syntax

XQuery: W3C-recommended query language

- XPath for navigation.
- FLWOR-blocks for iterating, pruning, grouping.

Example: simple join

```
for $x in $d/self::a/text(),
    $y in $d/self::b/text()
where $x = $y
return <c>{ $x }</c>
```

```
<a>1</><a>2</><a>3</>
<b>2</><b>3</><b>4</>
```

⇝

```
<c>2</><c>3</>
```

XQuery: Surface Syntax

XQuery: W3C-recommended query language

- XPath for navigation.
- FLWOR-blocks for iterating, pruning, grouping.

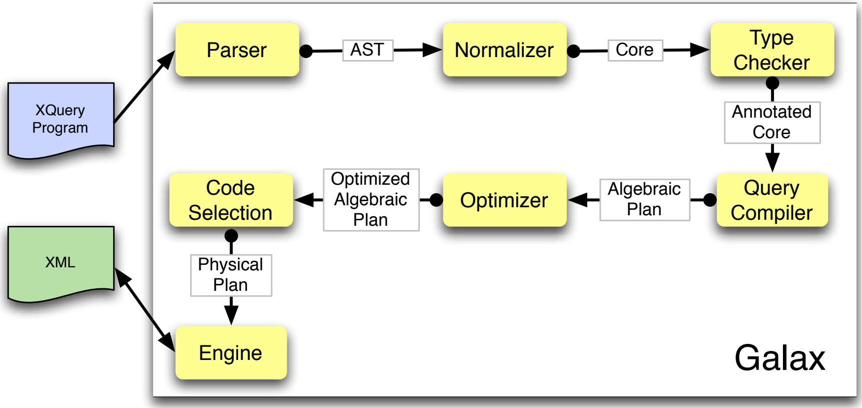
Example: simple join

```
for $x in $d/self::a/text(),
    $y in $d/self::b/text()
where $x = $y
return <c>{ $x }</c>
```

```
<a>1</><a>2</><a>3</>
<b>2</><b>3</><b>4</>
        ⇝
<c>2</><c>3</>
```

XQuery surface syntax is quite complex...

XQuery: Engine Architecture



XQuery: Compilation

```
for $x in $d/self::a/text(),
    $y in $d/self::b/text()
where $x = $y
return <c>{ $x }</c>
```

$\rightsquigarrow$

```
Map{Elem[c](#x)}
  (Select {eq(#x,#y) }
    (Product
      (Map{[x : ID]} (TreeJoin[self::a/text()](#d)),
      (Map{[y : ID]} (TreeJoin[self::b/text()](#d)))))
```

XQuery Algebra: Advantages

Simpler than surface syntax:

- FLWOR blocks broken down into simple operators.
- Variables translated into tuple operations;

Compositional semantics:

- Facilitates straightforward, inductive proof of correctness;
- Easily extended to new operators and built-in functions.

Exposes fundamental issues:

- Constants, tree constructors, and maps simple;
- Navigation, grouping, and selecting challenging.

Connects to previous work on view maintenance:

- Relations and bags.
- Complex values.

XQuery Algebra Syntax

$p ::= \mathrm{ID}$	(identity)
$\mid \mathrm{Empty}()$	(empty sequence)
$\mid \mathrm{Elem}[qn](p_1)$	(element)
$\mid \mathrm{Seq}(p_1, p_2)$	(sequence)
$\mid \mathrm{If}(p_1)\{p_2, p_3\}$	(conditional)
$\mid \mathrm{TreeJoin}[s](p_1)$	(navigation)
$\mid \#\mathrm{x}$	(tuple access)
$\mid [x : p_1]$	(tuple construction)
$\mid \mathrm{Map}\{p_1\}(p_2)$	(dependent map)
$\mid \mathrm{MapConcat}\{p_1\}(p_2)$	(concatenating map)
$\mid \mathrm{Select}\{p_1\}(p_2)$	(selection)
$\mid \mathrm{Product}(p_1, p_2)$	(product)
$s ::= ax :: nt$	(navigation step)

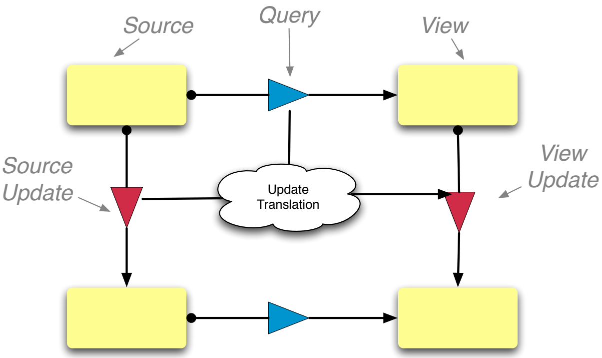
Update Language Syntax

Atomic updates + forms for nodes, tuples, sequences, tables.

$$
\begin{array}{rll}
u ::= & \mathrm{UNop} & \text{(no op)} \\
| & \mathrm{UDel} & \text{(deletion)} \\
| & \mathrm{UIns}(p) & \text{(insertion)} \\
| & \mathrm{URepl}(p) & \text{(replacement)} \\
| & \mathrm{UNode}(qno, u) & \text{(node update)} \\
| & \mathrm{USeq}(ul) & \text{(sequence update)} \\
| & \mathrm{UTup}(um) & \text{(tuple update)} \\
| & \mathrm{UTab}(ul) & \text{(table update)} \\
qno ::= & None \mid Some\ qn & \text{(optional name)} \\
ul ::= & [\,] \mid (i, u) :: ul & \text{(update list)} \\
um ::= & \{\} \mid \{x \mapsto u\} {+}{+} um & \text{(update map)}
\end{array}
$$

Can express *effect* of any update to an XML value.

Update Translation



Strategy: propagate an update u from bottom to top through the operators in an algebraic query p: $u \overset{p}{\rightsquigarrow} u'$.

Update Translation: Easy Operators

The first few cases are easy:

- If $p = \mathsf{ID}$
  then $u \overset{p}{\leadsto} u$.
- If $p = \mathsf{Empty}()$
  then $u \overset{p}{\leadsto} \mathsf{UNop}$.
- If $p = \mathsf{Elem}[qn](p_1)$ and $u \overset{p_1}{\leadsto} u_1$
  then $u \overset{p}{\leadsto} \mathsf{UNode}(None, u_1)$.

Update Translation: Conditional

But other algebraic operators compute, and then discard, intermediate views.

$$\frac{p_1 : t \longrightarrow \{Item\} \qquad p_2, p_3 : t \longrightarrow t'}{\mathsf{If}(p_1)\{p_2, p_3\} : t \longrightarrow t'}$$

Intermediate view: sequence computed by $p1$.

If $u \overset{p_1}{\leadsto} u_1$ then...

To finish the job, need to know:

- which of the branches (p_2 or p_3) was selected
- and whether the u_1 affects that choice!

Update Translation: Annotations

We could cache every intermediate view, but this would require *a lot* of redundant storage...

...so instead, we use a sparse annotation scheme that records:

- n the *length* of the sequence computed by p_1,
- x_1 the annotation for p_1,
- x_b the annotation for the selected branch.

Update Translation: Annotations

We could cache every intermediate view, but this would require *a lot* of redundant storage...

...so instead, we use a sparse annotation scheme that records:

- n the *length* of the sequence computed by p_1,
- x_1 the annotation for p_1,
- x_b the annotation for the selected branch.

To finish the job, let $u \overset{p_1}{\rightsquigarrow} u_1$. Then use a conservative analysis to test if u_1 changes branch selected.

- If "no", then $u \overset{p}{\rightsquigarrow} u'$, where $u \overset{p_b}{\rightsquigarrow} u'$.
- If "yes", then $u \overset{p}{\rightsquigarrow} \text{URepl}(p_{\overline{b}})$.
- If "maybe", then $u \overset{p}{\rightsquigarrow} \text{URepl}(p)$.

Update Translation: Sequences

A similar issue comes up with operators that *merge* sequences of values.

$$\frac{p_1, p_2 : t \longrightarrow \{t'\}}{\mathsf{Seq}(p_1, p_2) : t \longrightarrow \{t'\}}$$

If $u \overset{p_1}{\rightsquigarrow} u_1$ and $u \overset{p_2}{\rightsquigarrow} u_2$ then...

To finish the job, need to know how to merge u_1 and u_2 into an update that applies to the concatenated sequence.

We use an annotation that records the *lengths* of the sequences computed by p_1 and p_2.

Update Translation: Other Operators

Annotations record:

- XPath Navigation: paths to nodes in the view.
- Maps: lengths of sequences produced for each iteration.
- Tuple Operators: lengths of sequences
- Relational Operators: “fingerprint” and lengths of sequences of tuples.

See paper for many fiddly details...

Prototype

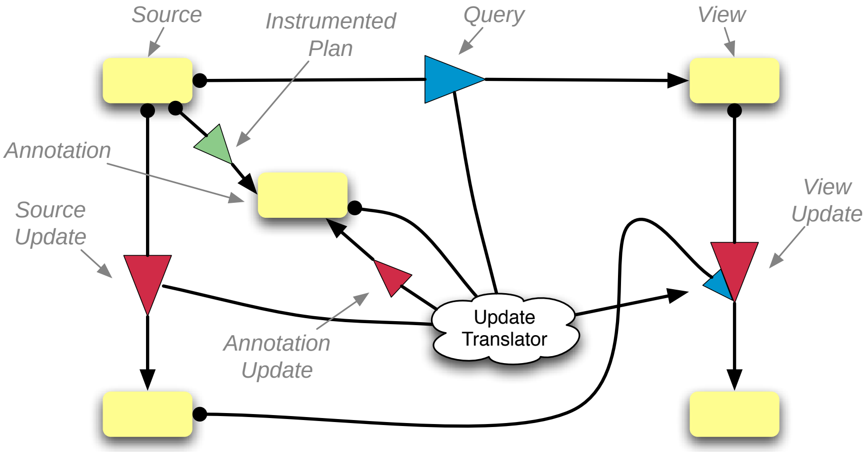
Built on top of the Galax XQuery engine.

2,500 lines of OCaml code

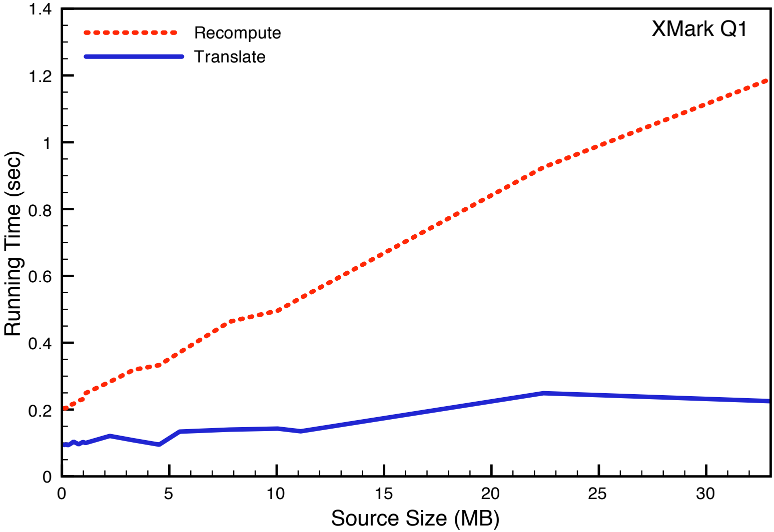
- Update Compiler: translates update language into XQuery! algebraic plans.
- Query Instrumentor: translates queries into instrumented plans that compute annotation files.
- Update Translator: takes as inputs a source update, a query, and an annotation, and calculates a view update.

Currently handles a core set of operators and built-in functions expressive enough to handle some simple XMark benchmarks; falls back to recomputation as needed.

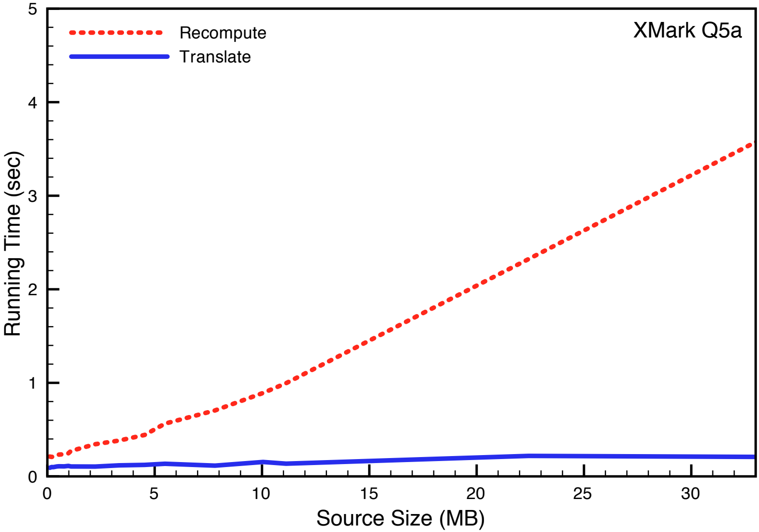
Final Architecture



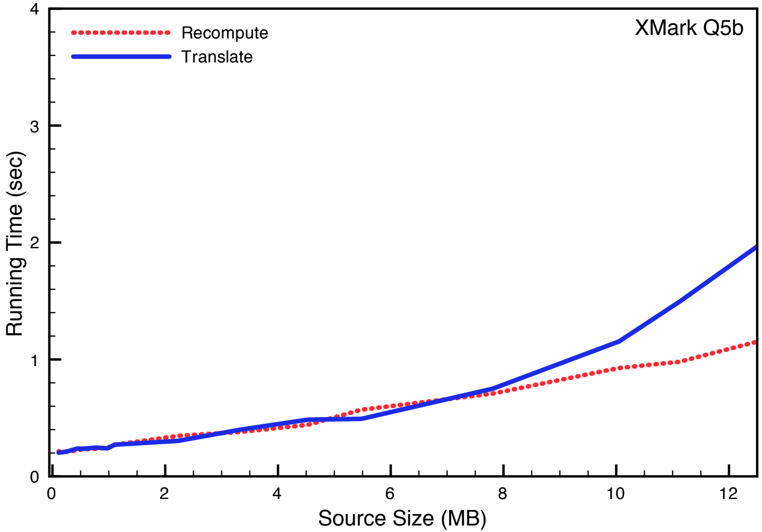
Experiments: Running Time (XMark Q1)



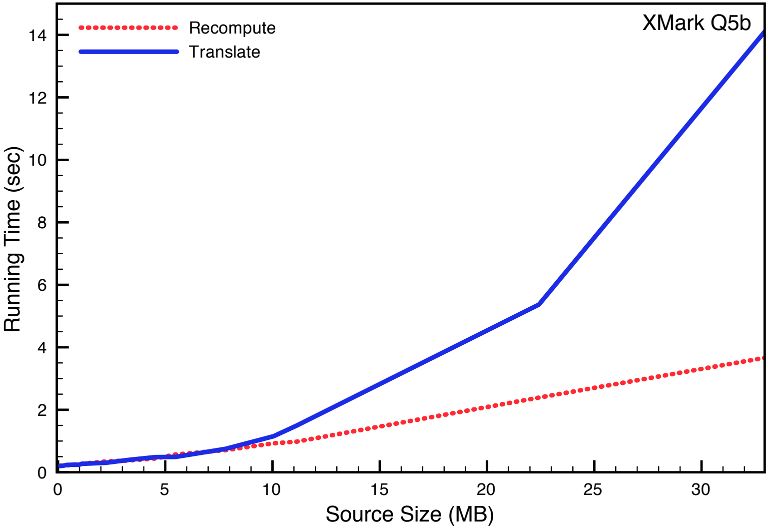
Experiments: Running Time (XMark Q5a)



Experiments: Running Time (XMark Q5b)



Experiments: Running Time (XMark Q5b)



Related Work

[Libkin + Griffin '96]: Relations and bags. Championed algebraic approach, notion of "minimal" updates.

[Zhuge + Garcia-Molina '97]: Graph-structured views. Early use of annotations.

[Liefke + Davidson '00]: Maintenance for simple queries over semi-structured data satisfying nice "distributive" properties.

[Sawires et. al. '05]: Maintenance for XPath views. Size of annotations only depends on the view–not the source.

[Rudensteiner et.al.'02-05]: Closest work to ours.

- Operates on XAT tree algebra; uses auxiliary data.
- Uses node identities to handle ordering.

Summary

Developed a *maintenance system* for XQuery views over XML.

Based on a compositional translation of simple updates through *algebraic* operators.

Uses *annotations* to guide update translation.

Prototype *implemented* on top of Galax.

Experimental results *validate* approach.

Future Work

Add support for *complete set* of algebraic operators, built-in functions. (Simple, since operators are fully compositional.)

Investigate maintenance of *recursive queries*.

Explore *query rewritings* motivated by maintainability.

Harness *type information* to reduce annotations, guide translation.

Measure effect of varying *annotations* on practical examples.

Hybrid approach using *provenance* metadata.

Thank you!