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

\[
1 + 2 + \cdots + 99 + 100
= (1 + 100) + (2 + 99) + \cdots (50 + 51)
= 101 \times 50
= 5050
\]
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

\[(1+2+\cdots+99+100) = 5050\]
View Maintenance

\[(1 + 2 + \cdots + 99 + 100) - 50 = 5050 - 50\]
View Maintenance
View Maintenance

Source ➔ Query ➔ View

Source Update
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: 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

<table>
<thead>
<tr>
<th>XQuery Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>for $x in $d/self::a/text(), $y in $d/self::b/text() where $x = $y return &lt;c&gt;{ $x }</code></td>
<td><code>&lt;a&gt;1&lt;/a&gt;&lt;a&gt;2&lt;/a&gt;&lt;a&gt;3&lt;/a&gt;&lt;b&gt;2&lt;/b&gt;&lt;b&gt;3&lt;/b&gt;&lt;b&gt;4&lt;/b&gt;</code></td>
</tr>
<tr>
<td></td>
<td>↓</td>
</tr>
<tr>
<td></td>
<td><code>&lt;c&gt;2&lt;/c&gt;&lt;c&gt;3&lt;/c&gt;</code></td>
</tr>
</tbody>
</table>
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>

Example output:

```
<a>1</a><a>2</a><a>3</a>
<b>2</b><b>3</b><b>4</b>
\[\]
<c>2</c><c>3</c>
```

XQuery surface syntax is quite complex...
XQuery: Engine Architecture

- Parser
- Normalizer
- Type Checker
- Core
- Annotated Core
- Code Selection
- Optimized Algebraic Plan
- Optimizer
- Algebraic Plan
- Query Compiler
- Engine
- Physical Plan
- XQuery Program
- XML

Galax
XQuery: Compilation

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

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

\[ s ::= ax :: nt \quad (\text{navigation step}) \]
Update Language Syntax

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

\[
\begin{align*}
  u ::= & \text{UNop} \quad \text{(no op)} \\
        & \text{UDel} \quad \text{(deletion)} \\
        & \text{UIns}(p) \quad \text{(insertion)} \\
        & \text{URepl}(p) \quad \text{(replacement)} \\
        & \text{UNode}(qno, u) \quad \text{(node update)} \\
        & \text{USeq}(ul) \quad \text{(sequence update)} \\
        & \text{UTup}(um) \quad \text{(tuple update)} \\
        & \text{UTab}(ul) \quad \text{(table update)}
\end{align*}
\]

\[
\begin{align*}
  qno ::= & \text{None} \mid \text{Some } qn \quad \text{(optional name)} \\
  ul ::= & [] \mid (i, u) : : ul \quad \text{(update list)} \\
  um ::= & \{} \mid \{ x \mapsto u \} ++ um \quad \text{(update map)}
\end{align*}
\]

Can express \textit{effect} of any update to an XML value.
Strategy: propagate an update $u$ from bottom to top through the operators in an algebraic query $p: u \xrightarrow{p} u'$. 
The first few cases are easy:

- If $p = \text{ID}$
  
  then $u \xrightarrow{\text{p}} u$.

- If $p = \text{Empty}()$
  
  then $u \xrightarrow{\text{p}} \text{UNop}$.

- If $p = \text{Elem}[qn](p_1)$ and $u \xrightarrow{\text{P}^1} u_1$
  
  then $u \xrightarrow{\text{p}} \text{UNode}(\text{None}, u_1)$. 
But other algebraic operators compute, and then discard, intermediate views.

\[
p_1 : t \rightarrow \{ \text{Item} \} \quad p_2, p_3 : t \rightarrow t' \\
\text{If}(p_1)\{p_2, p_3\} : t \rightarrow t'
\]

**Intermediate view**: sequence computed by \( p_1 \).

If \( u \xrightarrow{p_1} u_1 \) then...

To finish the job, need to know:
- which of the branches \((p_2\text{ or } p_3)\) 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 \textit{a lot} of redundant storage...

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

\begin{itemize}
  \item \texttt{n} the \textit{length} of the sequence computed by \texttt{p}_1,
  \item \texttt{x}_1 the annotation for \texttt{p}_1,
  \item \texttt{x}_b the annotation for the selected branch.
\end{itemize}

To finish the job, let \texttt{u} \overset{p_1}{\sim} \texttt{u}_1. Then use a conservative analysis to test if \texttt{u}_1 changes branch selected.

\begin{itemize}
  \item If “no”, then \texttt{u} \overset{p}{\sim} \texttt{u}', where \texttt{u} \overset{p_b}{\sim} \texttt{u}'.
  \item If “yes”, then \texttt{u} \overset{p}{\sim} \texttt{URepl}(p_b).
  \item If “maybe”, then \texttt{u} \overset{p}{\sim} \texttt{URepl}(p).
\end{itemize}
A similar issue comes up with operators that merge sequences of values.

\[
p_1, p_2 : t \rightarrow \{ t' \}
\]

\[
\text{Seq}(p_1, p_2) : t \rightarrow \{ t' \}
\]

If \( u \overset{p_1}{\sim} u_1 \) and \( u \overset{p_2}{\sim} u_2 \) 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 \( p_2 \).
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)

[Graph showing running time vs. source size for Recompute and Translate cases.]

- Recompute curve shows a steady increase with source size.
- Translate curve remains relatively flat and lower compared to Recompute.
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!