Relational Algebra: Mother Tongue
XQuery: Fluent
How to Compile XQuery Expressions into a Relational Algebra

Torsten Grust    Jens Teubner

University of Konstanz
Dept. of Computer and Information Science
Konstanz, Germany

June 21, 2004
Relational databases can efficiently back XPath evaluation.

- Encode XML structure using a numbering scheme.
  - “XPath accelerator” with pre/post tuples.
- Re-use existing database technology.
  - Storage management, index structures, query optimization.
- Support XPath evaluation through tailor-made operators.
  - “Staircase join” exploits properties of the pre/post encoding.
  - “Holistic” join algorithms to process entire XPath expressions.

Compile XQuery expressions into Relational Algebra.
Important XQuery features are not yet covered.

- Iteration in XQuery FLWOR expressions.
  - Contrary to set-oriented processing in relational databases?
- Construction of transient XML tree nodes.
  - Document table must be “extended” during query processing.

This talk addresses the first issue.
(The paper additionally covers the second.)

The ideas are orthogonal to efficient XPath evaluation.
XQuery FLWOR Expressions

XQuery is built around a looping primitive, the \texttt{for} construct.

\texttt{for $v$ in ($x_1$, $x_2$, ..., $x_n$) return $e$}

\begin{equation}
\equiv
( e[x_1/$v$], e[x_2/$v$], \ldots , e[x_n/$v$] )
\end{equation}

- $v$ is successively bound to the values of $x_i$.
- The return expression $e$ is evaluated for each binding.
- XQuery is a functional-style language.
  - It is sound to evaluate $e$ for all bindings in parallel.
XQuery’s basic data type is the sequence.

- Any expression evaluates to an **ordered sequence of items**.
- Sequences are always **flat**.

We may represent a sequence using a two-column table.

<table>
<thead>
<tr>
<th>pos</th>
<th>item</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>&quot;a&quot;</td>
</tr>
<tr>
<td>2</td>
<td>&quot;b&quot;</td>
</tr>
<tr>
<td>3</td>
<td>&quot;c&quot;</td>
</tr>
</tbody>
</table>

- Encode order in *pos*, the atom value in *item*.

(For now, we assume a polymorphic item type.)
We extend our sequence encoding by the column $\text{iter}$ that accounts for the independent iterations.

<table>
<thead>
<tr>
<th>iter</th>
<th>pos</th>
<th>item</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>20</td>
</tr>
</tbody>
</table>

Example:

for $\nu_0$ in (1, 2, 3) return 10

→ 10 appears in 3 iterations.

for $\nu_0$ in (1, 2, 3) return (10, 20)

→ (10, 20) appears in 3 iterations.

We refer to this as the loop lifted representation of a sequence.
We derive a compilation procedure that solely operates on loop lifted sequences.

**Example:** Body of `for $v$ in (10, 20, 30) return $v$

<table>
<thead>
<tr>
<th>iter</th>
<th>pos</th>
<th>item</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>30</td>
</tr>
</tbody>
</table>

- Start with representation of `(10, 20, 30)`.  
- Generate a new iteration for each value.  
- Each value forms a singleton sequence.  
  
(\(pos = 1\) for each tuple)

The **row number** operator \(\varrho\) in our algebra generates unique *iters*.
Nested XQuery Expressions

XQuery allows arbitrary expression nesting.

- **Example:**

\[
\begin{align*}
\text{s} & \left\{ \begin{array}{l}
\text{for } \nu_0 \text{ in } (1,2) \text{ return } \text{s}_0 \\
\quad \text{s}_0 \left\{ \begin{array}{l}
\text{for } \nu_{0.0} \text{ in } (10,20) \text{ return } \text{s}_{0.0} \\
\quad \text{s}_{0.0} \{ (\nu_0, \nu_{0.0}) \}
\end{array} \right.
\end{array} \right.
\end{align*}
\]

- We need to **map** value representations between scopes.
  - Variables defined in surrounding scopes.
  - The expression result of the for expression.
We capture expression nesting with help of a relation \textit{map}.

**Example:** (previous slide)

\[
\begin{align*}
\text{s} & \left\{ \\
\text{for } & \$v_0 \text{ in } (1,2) \text{ return} \\
\text{for } & \$v_{0.0} \text{ in } (10,20) \text{ return} \\
\text{s}_{0.0} & \left\{ \left( \$v_0, \$v_{0.0} \right) \right\} \\
\end{align*}
\]

<table>
<thead>
<tr>
<th>\text{outer}</th>
<th>\text{inner}</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

The relation lists corresponding \textit{iter} values of a \texttt{for} body and its surrounding scope.

Value mapping then renders into a join with this relation.

We handle \textbf{arbitrary} nesting this way.
We support an XQuery subset that suffices to handle the XMark benchmark set.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>literals</td>
<td>42, &quot;foo&quot;, (), ...</td>
</tr>
<tr>
<td>arithmetics</td>
<td>$e_1 + e_2$, $e_1 - e_2$, ...</td>
</tr>
<tr>
<td>builtin functions</td>
<td>fn:sum($e$), fn:count($e$), fn:doc($uri$)</td>
</tr>
<tr>
<td>variable bindings</td>
<td>let $v := e_1$ return $e_2$</td>
</tr>
<tr>
<td>iteration</td>
<td>for $v$ at $p$ in $e_1$ return $e_2$</td>
</tr>
<tr>
<td>conditionals</td>
<td>if $p$ then $e_1$ else $e_2$</td>
</tr>
<tr>
<td>sequence construction</td>
<td>$e_1$, $e_2$</td>
</tr>
<tr>
<td>function calls</td>
<td>$f(e_1, e_2, \ldots, e_n)$</td>
</tr>
<tr>
<td>element construction</td>
<td>element $e_1$ { $e_2$ }</td>
</tr>
<tr>
<td>XPath steps</td>
<td>$e/\alpha::\nu$</td>
</tr>
</tbody>
</table>
We generate a (almost) standard Relational Algebra.

\[
\pi_{a_1:b_1,\ldots,a_n:b_n} \quad \text{projection/renaming}
\]
\[
\sigma_a \quad \text{selection}
\]
\[
\bigcup \quad \text{disjoint union}
\]
\[
\times \quad \text{Cartesian product}
\]
\[
- \quad \text{difference}
\]
\[
\bowtie_{a=b} \quad \text{equi-join}
\]
\[
\varrho_{b: \langle a_1,\ldots,a_n \rangle}/p \quad \text{row numbering}
\]
\[
\boxtimes_{\alpha,\nu} \quad \text{XPath axis join}
\]
\[
\varepsilon \quad \text{element construction}
\]
\[
\otimes_{b: \langle a_1,\ldots,a_n \rangle} \quad \text{n-ary arithmetic/comparison operator *}
\]
\[
\begin{array}{c}
a \mid b \\
\end{array} \quad \text{literal table}
\]
\[
\text{count, } \ldots \quad \text{count and other aggregation functions}
\]
We implemented our translation for XMark queries in SQL.
We propose a fully relational evaluation for XQuery.

- A compilation procedure translates XQuery expressions into Relational Algebra.
- We can handle FLWOR expression, element construction and others.
- We can deal with arbitrary expression nesting.
- Experiments with an SQL based DBMS are promising.
- This work is part of our ongoing project “Pathfinder.”
- Resulting algebra expressions, however, can get large...