Yet Another Query Algebra for XML Data

Carlo Sartiani
Antonio Albano

Dipartimento di Informatica
Università di Pisa
Issues in XML Query Processing

- Path expression evaluation
Issues in XML Query Processing

- Path expression evaluation
- Nested query resolution
Issues in XML Query Processing

- Path expression evaluation
- Nested query resolution
- Order preservation
Issues in XML Query Processing

- Path expression evaluation
- Nested query resolution
- Order preservation
  - document order
Issues in XML Query Processing

- Path expression evaluation
- Nested query resolution
- Order preservation
  - document order
  - user-defined order
Issues in XML Query Processing

- Path expression evaluation
- Nested query resolution
- Order preservation
  - document order
  - user-defined order
  - join order
Logical Query Algebra

- An evolution of past OO and semistructured query algebras: GOM, YAT
- Three main objectives
  - preserving relational and OO optimization techniques
  - supporting efficient evaluation of path expressions
  - addressing specific problems: nested queries and ordering preservation
Main Features

- Multi-sorted: ordered and unordered sets
- Covering the FLWR fragment of XQuery
- XML nodes have oids
  - supporting both copy and reference semantics
Algebraic Operators

- Traditional operators manipulate sets (ordered and unordered) of flat tuples
  - $\sigma$, $\pi$, $\bowtie_P$, $< \cdot >$, $\chi$, $Sort$, $TupSort$, and $\Gamma$
- Border operators manage conversions XML $\rightarrow$ tuples, tuples $\rightarrow$ XML
  - $path$ evaluates paths and binds variables
  - $return$ builds up new XML elements
- Preservation of relational and OO optimization properties
• *path* extracts information from data sources, and builds variable bindings

• *path* behavior is described by a path filter

```
FOR $b$ in $\text{root/book}$,
    $a$ in $b//\text{author}$,

(path(/,$b, in)book[(//,$a, in)\text{author}[]]) (db1)
```

```
<table>
<thead>
<tr>
<th>$b$ : book1</th>
<th>$a$ : author1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b$ : book1</td>
<td>$a$ : author2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
```
FOR $b$ in $\text{root/book}$,
  $p$ in $\text{op:union($b//author,}$
  $\text{$b//publisher$)}$
FOR $b$ in $root/book,$
    $p$ in op:union($b//author,$
    $b//publisher)$

\[
\text{path}\ (\text{db1})
\]

\[
(/,b, in) book[(//,p, in)author[0] \lor (//,p, in)publisher[0])
\]

\[
(2)
\]
FOR $b$ in $\text{root/book}$,

$p$ in op:union($b//\text{author}$, $b//\text{publisher}$)

\[\text{path} \quad (db1)\]

\[\begin{array}{c}
\text{path} \\
(//, b, in) \text{book} [([^/], p, in) \text{author}[\emptyset] \lor ([/], p, in) \text{publisher}[\emptyset]]
\end{array}\]
return

- *return* uses binding tuples to produce a new XML document
return

- *return* uses binding tuples to produce a new XML document

```xml
FOR $b in $root/book,
    $t in $b/title,
    $a in $b/author
RETURN
    <entry> $t, $a </entry>
```
return

- *return* uses binding tuples to produce a new XML document

FOR $b$ in $\text{root/book}$,
  $t$ in $b/title$,
  $a$ in $b/author$
RETURN

\[
\text{<entry> } t, a \text{ </entry>}
\]

\[
\text{return}_{\text{entry}}[\nu t, \nu a](\n\text{path}(/, b, in)\text{book}[(/, a, in)\text{author}[\emptyset], (/, t, in)\text{title}[\emptyset]](db1))
\]
Ordering Issues

• Preservation of document order (order among elements in original documents)
• Preservation of join order
  • XQuery does not distinguish between joins and d-joins
• Imposing user-defined order
  • XQuery `SORTBY` clause
Sort

- *Sort* returns a set of tuples ordered according to a given predicate
- *Sort* may be used to preserve document order, join order as well as user-defined order
  - a specialized version $TupSort$ is used for document order
Sort Example

FOR $b$ in $root/book$
RETURN $b$
SORTBY (title)
Sort Example

FOR $b$ in $\text{root/book}$
RETURN $b$
SORTBY (title)

\[
\text{return}_v b \\
\text{Sort}_{u,v_t<v_t} (b) \\
\text{TupSort} (b) ( \\
\text{path}/,b,\text{in}\text{book}[/(,t,\text{in}\text{title}[]) (db1)])
\]
Optimization Properties

- Most operators are linear: $\sigma, \pi, \chi, \bowtie_P, \text{return}$
  - reordinability laws can be safely applied
- Most common rewriting rules can be applied
- There exist laws for decomposing complex path operations into simpler ones
- There exist laws for query unnesting
Path Decompositions

- These rules allow the query optimizer to choose the best evaluation strategy for each path
- Vertical decompositions
- Horizontal decompositions
Vertical Decompositions

- Useful for exploiting path indexes

FOR $b$ in $root/lib/book,$
   $a$ in $b/author,$
   $y$ in $b/year,$
Vertical Decompositions

- Useful for exploiting path indexes

\[
\text{FOR } b \text{ in } \text{root/lib/book,}
\]
\[
\text{a in } b/\text{author,}
\]
\[
y \text{ in } b/\text{year,}
\]

\[\text{path}(/,_,in)\text{lib}[(/,b, in)\text{book}[(/,a, in)\text{author}[]],(/,y, in)\text{year}[])](db_2)\]
Vertical Decompositions

- Useful for exploiting path indexes

\[
\text{FOR } b \text{ in } $root/lib/book, \\
\quad a \text{ in } b/author, \\
\quad y \text{ in } b/year, \\
\]

\[
\text{path}(\text{lib[}[/,b,\text{in})book[\text{}/,a,\text{in})author[\emptyset],(/,y,\text{in})year[\emptyset]])(db_2) \\
\downarrow \\
\text{path}_{F_1}(\text{path}(\text{lib[}[/,b,\text{in})book[\emptyset])(db_2))
\]
Horizontal Decompositions

- Useful for exploiting value indexes

FOR $b$ in $\text{root/lib/book}$,
$\quad$ $a$ in $\text{b/author}$,
$\quad$ $y$ in $\text{b/year}$,
WHERE $y = "1975"$
Horizontal Decompositions

- Useful for exploiting value indexes

\[
\sigma_{y = "1975"} \left( 
\begin{array}{c}
\text{path}(/,\_\_\_\_\_\_\_\_\_in)\text{lib}[(/,$b,in)\text{book}][\emptyset]](db2) < \\
\text{path}(/,$a,in)\text{author}[$b][\emptyset]($b) \bowtie true \text{ path}(/,$y,in)\text{year}[$b]($b) > 
\end{array} \right)
\]
Nested Queries

- Free nesting philosophy
- Widely used for
  - reshaping elements
  - regrouping elements
Brief Taxonomy

• Only type-N and type-J queries
  • predicate dependency
  • range dependency
  • projection dependency
FOR $a$ in library//author
RETURN $a$, <publist>
FOR $p$ in library/*,
   $aa$ in $p$/author
   WHERE $aa = $a
RETURN $p$
</publist>
Predicate Dependency

FOR $a$ in library//author
RETURN $a$, <publist> FOR $p$ in library/*,
    $aa$ in $p$/author
    WHERE $aa = a$
    RETURN $p$
</publist>

- Separating local variables from global ones
Predicate Dependency

FOR $a$ in library//author
RETURN $a$, <publist>
FOR $p$ in library/*,
$aa$ in $p/author$
WHERE $aa$ = $a$
RETURN $p$
</publist>

- Separating local variables from global ones
- Transforming the inner `return` filter
Range and Projection Dependency

- Range dependencies cannot be efficiently solved
  - no type extents
- Projection dependencies cannot be efficiently solved
  - cross products
Conclusions

• A query algebra for XML data
  • path evaluation
  • order preservation
  • nested query resolution
• Improving nested query resolution
• Merging a type system
XQuery

- A Turing-complete query language for XML data
  - maybe a database programming language
- Developed by W3C
  - enriched with some nasty stuff for industrial purposes
- Based on the Quilt core
- Query results are statically typed for inspection purposes
A FLWR query is composed by

- **FOR** and **LET** clauses (variable bindings)
- **WHERE** clause (variable filtering)
- **IF THEN ELSE**
- **RETURN** clause (result production)
- **SORTBY** clause (sort order enforcement)
XQuery(3)

FOR $b in $root/lib/book,
   $a in $b/author,
   $y in $b/year,
WHERE $y = "1975"
RETURN <entry> $a, $y </entry>
SORTBY (title)