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
  - σ, π, $\bowtie_P$, $< \cdot >$, χ, $Sort$, $TupSort$, and Γ
- 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

- *path* extracts information from data sources, and builds variable bindings
- *path* behavior is described by a path filter

```
FOR $b in $root/book,
    $a in $b//author,
```

$$path_{(/,\$b,in)book[(//,\$a,in)author[\emptyset]]}(db1)$$

$\$b : book_1$	$\$a : author_1$
$\$b : book_1$	$\$a : author_2$
$\ldots$	$\ldots$

path (2)

```
FOR $b in $root/book,
    $p in op:union($b//author,
                   $b//publisher)
```

$path$ (2)

```
FOR $b in $root/book,
    $p in op:union($b//author,
                   $b//publisher)
```

$$path_{(/,\$b,in)book[(//,\$p,in)author[\emptyset] \ \vee \ (//,\$p,in)publisher[\emptyset]]}(db1)$$

path (2)

```
FOR $b in $root/book,
    $p in op:union($b//author,
                   $b//publisher)
```

$$path_{(/,\$b,in)book[(//,\$p,in)author[\emptyset] \vee (//,\$p,in)publisher[\emptyset]]}(db1)$$

return

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

return

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

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

$$return_{entry[\nu\$t,\nu\$a]}($$
$$path_{(/,\$b,in)book[(/,\$a,in)author[\emptyset],(/,\$t,in)title[\emptyset]]}(db1))$$

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 $root/book
RETURN $b
SORTBY (title)
```

$$return_{\nu \$b}($$
$$Sort_{u.\$t<v.\$t}($$
$$TupSort_{(\$b)}($$
$$path_{(/,\$b,in)book[(/,\$t,in)title[\emptyset]]}(db1))))$$

Optimization Properties

- Most operators are linear: σ, π, χ, $\bowtie_P$, $return$
  - 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 allows 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

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

$$path_{(/,\_,in)lib[(/,\$b,in)book[(/,\$a,in)author[\emptyset],(/,\$y,in)year[\emptyset]]]}(db_2)$$

Vertical Decompositions

- Useful for exploiting path indexes

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

$$path_{(/,\_,in)lib[(/,\$b,in)book[(/,\$a,in)author[\emptyset],(/,\$y,in)year[\emptyset]]]}(db_2)$$

$$\Downarrow$$

$$path_{F_1}(path_{(/,\_,in)lib[(/,\$b,in)book[\emptyset]]}(db2))$$

Horizontal Decompositions

- Useful for exploiting value indexes

```
FOR $b in $root/lib/book,
    $a in $b/author,
    $y in $b/year,
WHERE $y = "1975"
```

Horizontal Decompositions

- Useful for exploiting value indexes

```
FOR $b in $root/lib/book,
    $a in $b/author,
    $y in $b/year,
WHERE $y = "1975"
```

$$\sigma_{\$y=\text{"}1975\text{"}}($$
$$path_{(/,\_,in)lib[(/,\$b,in)book[\emptyset]]}(db2) <$$
$$path_{(/,\$a,in)author[\emptyset]}(\$b) \bowtie_{true} path_{(/,\$y,in)year[\emptyset]}(\$b) >)$$

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

Predicate 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

XQuery(2)

- 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)
```