XML-Relational Mapping

Introduction to Databases
CompSci 316 Fall 2019
Announcements (Mon., Oct. 21)

• Homework 3 due in two weeks
• Project milestone 2 due in 2½ weeks
  • Milestone #1 feedback available through Gradescope
• Weekly project progress updates starting this Wed.
  • Due on Piazza every Wednesday
Approaches to XML processing

• Text files/messages
• Specialized XML DBMS
  • Tamino (Software AG), BaseX, eXist, Sedna, ...
  • Not as mature as relational DBMS
• Relational (and object-relational) DBMS
  • Middleware and/or extensions
  • IBM DB2’s pureXML, Oracle XML DB, and XML type/functions from Microsoft SQL Server, PostgreSQL, MySQL...
Mapping XML to relational

• Store XML in a column
  • Simple, compact
  • CLOB (Character Large OBject) type + full-text indexing, or better, special XML type + functions
  • Poor integration with relational query processing
  • Updates are expensive

• Alternatives?
  • **Schema-oblivious mapping:** well-formed XML $\rightarrow$ generic relational schema
    • Node/edge-based mapping for graphs
    • Interval-based mapping for trees
    • Path-based mapping for trees
  • **Schema-aware mapping:** valid XML $\rightarrow$ special relational schema based on DTD

← Focus of this lecture
Node/edge-based: schema

- **Element**(eid, tag)
- **Attribute**(eid, attrName, attrValue)  Key: (eid, attrName)
  - Attribute order does not matter
- **ElementChild**(eid, pos, child)  Keys: (eid, pos), (child)
  - pos specifies the ordering of children
  - child references either **Element**(eid) or **Text**(tid)
- **Text**(tid, value)
  - tid cannot be the same as any eid

⚠️ Need to “invent” lots of id’s
⚠️ Need indexes for efficiency, e.g., **Element**(tag), **Text**(value)
Node/edge-based: example

```xml
<bibliography>
  <book ISBN="ISBN-10" price="80.00">
    <title>Foundations of Databases</title>
    <author>Abiteboul</author>
    <author>Hull</author>
    <author>Vianu</author>
    <publisher>Addison Wesley</publisher>
    <year>1995</year>
  </book>
</bibliography>
```

**Element**

<table>
<thead>
<tr>
<th>eid</th>
<th>tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>e0</td>
<td>bibliography</td>
</tr>
<tr>
<td>e1</td>
<td>book</td>
</tr>
<tr>
<td>e2</td>
<td>title</td>
</tr>
<tr>
<td>e3</td>
<td>author</td>
</tr>
<tr>
<td>e4</td>
<td>author</td>
</tr>
<tr>
<td>e5</td>
<td>author</td>
</tr>
<tr>
<td>e6</td>
<td>publisher</td>
</tr>
<tr>
<td>e7</td>
<td>year</td>
</tr>
</tbody>
</table>

**Attribute**

<table>
<thead>
<tr>
<th>eid</th>
<th>attrName</th>
<th>attrValue</th>
</tr>
</thead>
<tbody>
<tr>
<td>e1</td>
<td>price</td>
<td>80</td>
</tr>
</tbody>
</table>

**Text**

<table>
<thead>
<tr>
<th>tid</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>t0</td>
<td>Foundations of Databases</td>
</tr>
<tr>
<td>t1</td>
<td>Abiteboul</td>
</tr>
<tr>
<td>t2</td>
<td>Hull</td>
</tr>
<tr>
<td>t3</td>
<td>Vianu</td>
</tr>
<tr>
<td>t4</td>
<td>Addison Wesley</td>
</tr>
<tr>
<td>t5</td>
<td>1995</td>
</tr>
</tbody>
</table>
Node/edge-based: simple paths

• //title
  • SELECT eid FROM Element WHERE tag = 'title';

• //section/title
  • SELECT e2.eid
    FROM Element el, ElementChild c, Element e2
    WHERE el.tag = 'section'
    AND e2.tag = 'title'
    AND el.eid = c.eid
    AND c.child = e2.eid;

⚠️ Path expression becomes joins!
  • Number of joins is proportional to the length of the path expression
Node/edge-based: complex paths

- //bibliography/book[author="Abiteboul"]/@price
  - SELECT a.attrValue
    FROM Element e1, ElementChild c1,
        Element e2, Attribute a
    WHERE e1.tag = 'bibliography'
    AND e1.eid = c1.eid AND c1.child = e2.eid
    AND e2.tag = 'book'
  AND EXISTS (SELECT * FROM ElementChild c2,
              Element e3, ElementChild c3, Text t
    WHERE e2.eid = c2.eid AND c2.child = e3.eid
    AND e3.tag = 'author'
    AND e3.eid = c3.eid AND c3.child = t.tid
    AND t.value = 'Abiteboul'
    )
  AND e2.eid = a.eid
  AND a.attrName = 'price';
Node/edge-based: descendent-or-self

• //book//title
  • Requires SQL3 recursion
  • WITH RECURSIVE ReachableFromBook(id) AS
    ((SELECT eid FROM Element WHERE tag = 'book')
     UNION
     (SELECT c.child
      FROM ReachableFromBook r, ElementChild c
      WHERE r.eid = c.eid))
    SELECT eid
    FROM Element
    WHERE eid IN (SELECT * FROM ReachableFromBook)
    AND tag = 'title';
Interval-based: schema

• **Element**(left, right, level, tag)
  • *left* is the start position of the element
  • *right* is the end position of the element
  • *level* is the nesting depth of the element (strictly speaking, unnecessary)
  • Key is *left*

• **Text**(left, right, level, value)
  • Key is *left*

• **Attribute**(left, attrName, attrValue)
  • Key is (left, attrName)
Interval-based: example

Where did `ElementChild` go?

• $e_1$ is the parent of $e_2$ iff:
  
  \[
  [e_1.left, e_1.right] \supset [e_2.left, e_2.right], \text{ and } e_1.level = e_2.level - 1
  \]
Interval-based: queries

• //section/title
  • SELECT e2.left
    FROM Element el, Element e2
    WHERE el.tag = 'section' AND e2.tag = 'title'
    AND el.left < e2.left AND e2.right < el.right
    AND el.level = e2.level-1;
  ✐ Path expression becomes “containment” joins!
  • Number of joins is proportional to path expression length

• //book/title
  • SELECT e2.left
    FROM Element el, Element e2
    WHERE el.tag = 'book' AND e2.tag = 'title'
    AND el.left < e2.left AND e2.right < el.right;
  ✐ No recursion!
Summary so far

Node/edge-based vs. interval-based mapping

• Path expression steps
  • Equality vs. containment join

• Descendent-or-self
  • Recursion required vs. not required
Path-based mapping: approach 1

Label-path encoding: paths as strings of labels

- **Element**(pathid, left, right, ...), **Path**(pathid, path), ...

  - path is a string containing the sequence of labels on a path starting from the root
  - Why are left and right still needed?

<table>
<thead>
<tr>
<th>pathid</th>
<th>left</th>
<th>right</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>999</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>9</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>12</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>pathid</th>
<th>path</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>/bibliography</td>
</tr>
<tr>
<td>2</td>
<td>/bibliography/book</td>
</tr>
<tr>
<td>3</td>
<td>/bibliography/book/title</td>
</tr>
<tr>
<td>4</td>
<td>/bibliography/book/author</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Label-path encoding: queries

- Simple path expressions with no conditions
  //book/title
  - Perform string matching on Path
  - Join qualified pathid’s with Element
- //book[publisher='Prentice Hall']/title
  - Evaluate //book/title
  - Evaluate //book/publisher[text()='Prentice Hall']
  - Must then ensure title and publisher belong to the same book (how?)

Path expression with attached conditions needs to be broken down, processed separately, and joined back
Path-based mapping: approach 2

Dewey-order encoding

• Each component of the id represents the order of the child within its parent
  • Unlike label-path, this encoding is “lossless”

```
Element(dewey_pid, tag)
Text(dewey_pid, value)
Attribute(dewey_pid, attrName, attrValue)
```
Dewey-order encoding: queries

• Examples:
  //title
  //section/title
  //book//title
  //book[publisher='Prentice Hall']/title

  • Works similarly as interval-based mapping
    • Except parent/child and ancestor/descendant relationship are checked by prefix matching
  • Serves a different purpose from label-path encoding
  • Any advantage over interval-based mapping?
Summary

• XML data can be “shredded” into rows in a relational database

• XQueries can be translated into SQL queries
  • Queries can then benefit from smart relational indexing, optimization, and execution

• With schema-oblivious approaches, comprehensive XQuery-SQL translation can be easily automated
  • Different data mapping techniques lead to different styles of queries

• Schema-aware translation is also possible and potentially more efficient, but automation is more complex