XML-Relational Mapping
Introduction to Databases
CompSci 316 Fall 2016

Announcements (Thu., Nov. 3)
• Homework #3 due next Tuesday
• Project milestone #2 due next Thursday

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, PostgreSQL’s XML type/functions...
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: Focus of this lecture
  - well-formed XML → generic relational schema
  - Node/edge-based mapping for graphs
  - Interval-based mapping for trees
  - Path-based mapping for trees
- Schema-aware mapping:
  - valid XML → special relational schema based on DTD

Node/edge-based: schema

- **Element** (eid, tag)
- **Attribute** (eid, attrName, attrValue) Key:
  - Attribute order does not matter
- **ElementChild** (eid, pos, child) Keys:
  - 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

<table>
<thead>
<tr>
<th>Element</th>
<th>ElementChild</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
<td>1</td>
</tr>
<tr>
<td>a2</td>
<td>2</td>
</tr>
<tr>
<td>a3</td>
<td>3</td>
</tr>
<tr>
<td>a4</td>
<td>4</td>
</tr>
<tr>
<td>a5</td>
<td>5</td>
</tr>
<tr>
<td>a6</td>
<td>6</td>
</tr>
<tr>
<td>a7</td>
<td>7</td>
</tr>
<tr>
<td>a8</td>
<td>8</td>
</tr>
<tr>
<td>a9</td>
<td>9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1</td>
<td>Foundations of Database, Abiteboul, Hull, Vianu, Addison Wesley, 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 e1, ElementChild c, Element e2
    WHERE e1.tag = 'section'
    AND e2.tag = 'title'
    AND e1.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 a.attrName = 'price'
    AND EXISTS (SELECT * FROM ElementChild c2,
                 Element e3, ElementChild c3, Text t
                 WHERE e2.eid = c2.eid AND c2.child = e3.eid
                 AND e1.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

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

- **Attribute** (left, attrName, attrValue)
  - Key is

Interval-based: queries

- //section/title
  - SELECT e2.left
    FROM Element e1, Element e2
    WHERE e1.tag = 'section' AND e2.tag = 'title'
    AND e1.left < e2.left AND e2.right < e1.right
    AND e1.level = e2.level - 1;
  - Path expression becomes “containment” joins!
  - Number of joins is proportional to path expression length

- //book/title

---

Where did ElementChild go?

- e1 is the parent of e2 iff:
  - \([e1.left, e1.right] \supset [e2.left, e2.right]\), and
  - e1.level = e2.level - 1
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>Element</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>pathid</td>
<td>path</td>
</tr>
<tr>
<td>1</td>
<td>/bibliography/books/book/title</td>
</tr>
<tr>
<td>2</td>
<td>/bibliography/books/book/author</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”

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