**XML-Relational Mapping**

CPS 216
Advanced Database Systems

---

**Announcements (March 18)**

- Midterm sample solution available outside my office
- Course project milestone 2 due March 30
- Homework #3 due April 6
- Talk by Amol Deshpande
  - Adaptive Query Processing to Handle Estimation Errors
  - Monday, 11:30am-12:30pm, D106
- Reading assignment due next Monday
  - Two *VLDB* papers on native XML databases

---

**Approaches to XML processing**

- Text files (!)
- Specialized XML DBMS
  - Lore (Stanford), Strudel (AT&T), Tamino/Quip (Software AG), X-Hive, Timber (Michigan), etc.
  - Still a long way to go
- Object-oriented DBMS
  - eXcelon (ObjectStore), ozone, etc.
  - Not as mature as relational DBMS
- Relational (and object-relational) DBMS
  - Middleware and/or object-relational extensions
Mapping XML to relational

- Store XML in a CLOB (Character Large OBject) column
  - Simple, compact
  - Full-text indexing can help (often provided by DBMS vendors as object-relational “extensions”)

- Alternatives?
  - Schema-oblivious mapping:
    - 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)**
  - Attribute order does not matter
- **ElementChild(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>
```

<table>
<thead>
<tr>
<th>Element</th>
<th>ElementChild</th>
<th>Attribute</th>
<th>Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>eid</td>
<td>pos</td>
<td>attrName</td>
<td>value</td>
</tr>
<tr>
<td>e0</td>
<td>1</td>
<td>bibliography</td>
<td>t0</td>
</tr>
<tr>
<td>e1</td>
<td>1</td>
<td>title</td>
<td>Foundations of Databases</td>
</tr>
<tr>
<td>e2</td>
<td>2</td>
<td>author</td>
<td>Abiteboul</td>
</tr>
<tr>
<td>e3</td>
<td>3</td>
<td>author</td>
<td>Hull</td>
</tr>
<tr>
<td>e4</td>
<td>4</td>
<td>author</td>
<td>Vianu</td>
</tr>
<tr>
<td>e5</td>
<td>5</td>
<td>publisher</td>
<td>Addison Wesley</td>
</tr>
<tr>
<td>e6</td>
<td>6</td>
<td>year</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 e1, ElementChild c, Element e2
    WHERE e1.tag = 'section'
    AND e2.tag = 'title'
    AND e1.eid = c.eid
    AND c.child = e2.eid;

Node/edge-based: more 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.eid = e2.eid
    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 e2.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
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)
  - *tag* is *left*

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

- **Text(left, level, value)**

  *Where did ElementChild go?*

Interval-based: example

```
<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>
```

```
//book//title
SELECT e2.left
FROM Element e1, Element e2
WHERE e1.tag = 'book' AND e2.tag = 'section'
AND e1.left < e2.left AND e2.right < e1.right;

//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
```
How about XQuery?

DeHaan et al. SIGMOD 2003

- Evaluating an XQuery expression results in a sequence of environments
  - An environment $E$ maps each query variable $v$ to its value: a forest of XML trees (a node-set) $f_v$
- Encode using tables with “dynamic intervals”
  - Table $f$: increasing sequence of integers, one per environment
  - For each query variable $v$, create a table $T_v(\text{string}, \text{left}, \text{right})$ representing the value of $v$ in all environments
    - Sorted on $l$ to support efficient processing
    - Different environments form non-overlapping regions

Example $T_v$

<table>
<thead>
<tr>
<th>$f_v$</th>
<th>$v$</th>
<th>$s$</th>
<th>$l$</th>
<th>$r$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$\text{f@id}$</td>
<td>122</td>
<td>105</td>
<td>170</td>
</tr>
<tr>
<td>2</td>
<td>$\text{f@did}$</td>
<td>125</td>
<td>126</td>
<td>170</td>
</tr>
<tr>
<td>3</td>
<td>$\text{f@person}$</td>
<td>126</td>
<td>127</td>
<td>162</td>
</tr>
<tr>
<td>4</td>
<td>$\text{f@name}$</td>
<td>160</td>
<td>161</td>
<td>163</td>
</tr>
<tr>
<td>5</td>
<td>$\text{mailto:\text{f@email}}$</td>
<td>163</td>
<td>165</td>
<td>165</td>
</tr>
<tr>
<td>6</td>
<td>$\text{c@phone}$</td>
<td>165</td>
<td>169</td>
<td>169</td>
</tr>
<tr>
<td>7</td>
<td>$\text{<a href="http://www.lab.com/%5Ctext%7Bf@home%7D%7D$">http://www.lab.com/\text{f@home}}$</a></td>
<td>169</td>
<td>169</td>
<td>169</td>
</tr>
</tbody>
</table>

Translating /

- Given $T_v$ for values of $v$, compute $v$/name
Translating //

- Given $T_i$ for values of $v$, compute $v$//

Translating for

- Path expression steps become containment joins
- No recursion needed for descendent-or-self
- Comprehensive XQuery-SQL translation is possible with dynamic interval encoding
  - Looks hairy, but with some special tweaks to the relational engine, it actually performs better than many of the currently available native XQuery products!
  - Set-oriented processing helps!
A path-based mapping
Label-path encoding
- **Element**\(\text{pathid}, \text{left}, \text{right}, \text{value}\), **Path**\(\text{pathid}, \text{path}\)
  - \(\text{path}\) is a label path starting from the root
  - Why are \(\text{left}\) and \(\text{right}\) still needed?

<table>
<thead>
<tr>
<th>Element</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>pathid</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</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\ila
    - Join qualified \text{pathid}'s with \Element
- Path expression with attached conditions need to be broken down, processed separately, and joined back
  - \//book[publisher='Prentice Hall']/title
    - Evaluate \//book
    - Evaluate \//book/title
    - Evaluate \//book/publisher[text()='Prentice Hall']
    - Join to ensure \text{title} and \text{publisher} belong to the same \text{book}

Another path-based mapping
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

Schema-aware mapping

- Idea: use DTD to design a better schema
- Basic approach: elements of the same type go into one table
  - Tag name → table name
  - Attributes → columns
    - If one exists, ID attribute → key column; otherwise, need to "invent" a key
    - IDREF attribute → foreign key column
  - Children of the element → foreign key columns
    - Ordering of columns encodes ordering of children

```xml
<!DOCTYPE bibliography [...
<!ELEMENT book (title, …)
<!ATTLIST book ISBN ID #REQUIRED
<!ATTLIST book price CDATA #IMPLIED
<!ELEMENT title (#PCDATA)
```

Handling * and + in DTD

- What if an element can have any number of children?
- Example: Book can have multiple authors
  - BCNF?

- Idea: create another table to track such relationships
  - BCNF decomposition in action!
  - A further optimization: merge book_author into author

- Need to add position information if ordering is important
Inlining

- An author element just has a PCDATA child
- Instead of using foreign keys
  - author(id, PCDATA_id)
  - PCDATA(id, value)
- Why not just “inline” the string value inside book?
  - book_author(ISBN, author_PCDATA_value)
  - PCDATA table no longer stores author values

More general inlining

- As long as we know the structure of an element and its number of children (and recursively for all children), we can inline this element where it appears
  - \(<\text{book ISBN}="\ldots">\ldots\</\text{book}>\)
    - \(<\text{publisher}>\ldots</\text{publisher}>\)
    - \(<\text{name}>\ldots</\text{name}><\text{address}>\ldots</\text{address}>\)</n
- With no inlining at all
- With inlining
  - publisher(id, name_id, address_id)
  - name(id, PCDATA_id)
  - address(id, PCDATA_id)
  - book(ISBN, publisher_name_PCDATA_value, publisher_address_PCDATA_value)

Queries

- //title
- //section/title
- //bibliography/book[author="Abiteboul"]/@price
- //book/title

These queries only work for the given DTD
Pros and cons of inlining

- Not always applicable

Result restructuring

- Simple results are fine
  - Each tuple returned by SQL gets converted to an element

- Simple grouping is fine (e.g., books with multiple authors)
  - Tuples can be returned by SQL in sorted order; adjacent tuples are grouped into an element

- Complex results are problematic: one SQL query only returns a single table; columns cannot contain sets or structures
  - E.g., books with multiple authors and multiple references
    - Option 1: one table with all combos of authors/references → bad
    - Option 2: two tables, one with authors and the other with references → join is done as post processing
    - Option 3: sorted “union” of NULL-padded authors and references

Comparison of approaches

- Schema-oblivious
  - Flexible and adaptable; no DTD needed
  - Queries are easy to formulate
    - Translation from XPath/XQuery can be easily automated
  - Queries involve lots of join and are expensive

- Schema-aware
  - Less flexible and adaptable
  - Need to know DTD to design the relational schema
  - Query formulation requires knowing DTD and schema
  - Queries are more efficient
  - XQuery is tougher to formulate because of result restructuring