XML-Relational Mapping

CPS 116
Introduction to Database Systems

Announcements (October 20)
- Homework #3 due next Tuesday
- Project milestone #2 due in 2½ weeks
  - Check your email for feedback on milestone #1

Approaches to XML processing
- Text files (!)
- Specialized XML DBMS
  - Lore (Stanford), Strudel (AT&T), Timber (Michigan), MonetDB/XQuery (CWI, Netherlands), Tamino (Software AG), eXist, Sedna, Apache Xindice, XML:DB API initiative…
  - Still some way to go
- Object-oriented DBMS
  - ObjectStore, ozone, …
  - Not as mature as relational DBMS
- Relational (and object-relational) DBMS
  - Middleware and/or extensions
  - IBM DB2’s pureXML, …

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”)
  - Poor integration with relational query processing
  - Updates are expensive
- 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)
  - 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 ids
  - Need indexes for efficiency, e.g., Element(tag), Text(value)

Node/edge-based: example

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>isbn</td>
<td>1388-10</td>
</tr>
<tr>
<td>price</td>
<td>80.00</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Text</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>publisher</td>
<td>Addison Wesley</td>
</tr>
<tr>
<td>year</td>
<td>1995</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>title</td>
<td>Foundations of Databases</td>
</tr>
<tr>
<td>author</td>
<td>Abiteboul, Hull, Vianu</td>
</tr>
<tr>
<td>year</td>
<td>1995</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ElementChild</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>isbn</td>
<td>1388-10</td>
</tr>
<tr>
<td>price</td>
<td>80.00</td>
</tr>
<tr>
<td>author</td>
<td>Abiteboul, Hull, Vianu</td>
</tr>
<tr>
<td>publisher</td>
<td>Addison Wesley</td>
</tr>
<tr>
<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;
- Path expression becomes joins!
  - Number of joins is proportional to the length of the path expression

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 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 ALL
    (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

```
//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>
```

- Where did ElementChild go?
  - E1 is the parent of E2 iff:
    [E1.left, E1.right] ⊃ [E2.left, E2.right], and
    E1.level = E2.level – 1

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
  - SELECT e2.left
    FROM Element e1, Element e2
    WHERE e1.tag = 'book' AND e2.tag = 'title'
    AND e1.left < e2.left AND e2.right < e1.right;
  - No recursion!
Summary of interval-based mapping

- Path expression steps become containment joins
- No recursion needed for descendent-or-self
- Comprehensive XQuery-SQL translation is possible

A path-based mapping

Label-path encoding

- `Element(pathid, left, right, ...), Path(pathid, path, ...)
- path is a label path starting from the root
- Why are left and right still needed? To preserve structure

<table>
<thead>
<tr>
<th>pathid</th>
<th>left</th>
<th>right</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>9</td>
<td>11</td>
</tr>
<tr>
<td>5</td>
<td>12</td>
<td>14</td>
</tr>
</tbody>
</table>

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`

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’]`
  - How to ensure `title` and `publisher` belong to the same `book`?
  - Path expression with attached conditions needs to be broken down, processed separately, and joined back

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?

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
Complex results are problematic (e.g. books with 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

Pros and cons of inlining
- Not always applicable
  - * and +, recursive schema (e.g., section)
- Fewer joins
- More “scattering” (e.g., there is no longer any table containing all titles; author information is scattered across book, section, etc.)
  - Heuristic: do not inline elements that can be shared

Simple grouping is fine (e.g., books with multiple authors)

- With no inlining at all
- With inlining

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

Handling * and + in DTD
- What if an element can have any number of children?
- Example: Book can have multiple authors
  - BCNF?

Queries
- book(ISBN, price, title, publisher, year),
  section(id, title, text), section_section(id, section_pos, section_id)
- //title
  - [SELECT title FROM book] UNION ALL
  - [SELECT title FROM section];
  - //section/title
    - SELECT title FROM section;
  - //bibliography/book[author="Abiteboul"]/@price
    - SELECT price FROM book, book_author

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 (e.g., books with multiple authors and multiple references)
  - One SQL query returns one table, whose columns cannot store sets
    - Option 1: return one table with all combinations of authors and references → bad
    - Option 2: return two tables, one with authors and the other with references → join is done as post processing
    - Option 3: return one table with all author and reference columns; pad with NULLs; order determines grouping → messy

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
Comparison of approaches

Schema-oblivious
- Flexible and adaptable; no DTD needed
- Queries are easy to formulate
  - Translation 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