XML-Relational Mapping

CompSci 316
Introduction to Database Systems

Announcements (Tue. Nov. 5)

- Homework #3
  - Gradience part due today
  - Non-Gradience part due Thursday
- Project milestone #2 due in 9 days

Approaches to XML processing

- Text files (!)
- Specialized XML DBMS
  - Lore (Stanford), Strudel (AT&T), Timber (Michigan), MonetDB/XQuery (CWI, Netherlands), Tamino (Software AG), BaseX, eXist, Sedna, …
  - 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, PostgreSQL’s XML type/functions…

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(tid, 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 tid’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>id</td>
<td>author</td>
</tr>
<tr>
<td>ISBN-10</td>
<td>title</td>
</tr>
<tr>
<td>ISBN</td>
<td>year</td>
</tr>
<tr>
<td>price</td>
<td>publisher</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attribute</th>
<th>EAT/LAT/LAN</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAT</td>
<td>ISBN-10</td>
<td>12</td>
</tr>
<tr>
<td>LAT</td>
<td>ISBN-10</td>
<td>2</td>
</tr>
<tr>
<td>LAN</td>
<td>ISBN-10</td>
<td>6</td>
</tr>
<tr>
<td>ISBN</td>
<td>title</td>
<td>5</td>
</tr>
<tr>
<td>title</td>
<td>ISBN-10</td>
<td>5</td>
</tr>
<tr>
<td>ISBN</td>
<td>year</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Text</th>
<th>node</th>
</tr>
</thead>
<tbody>
<tr>
<td>node</td>
<td>node</td>
</tr>
<tr>
<td>node</td>
<td>node</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 e1.eid = c.eid
    AND e2.eid = e1.eid;
  - Path expression becomes joins!
    - Number of joins is proportional to the length of the path expression

Node/edge-based: descendent-or-self

- //book/title
  - Requires SQL5 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 e2.left
    FROM Element e1, ElementChild c1,
    Element e2
    WHERE e1.tag = 'bibliography'
    AND e1.eid = c1.eid AND c1.child = e2.eid
    AND e2.tag = 'title';

Interval-based: example

- 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

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!

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)
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>Element</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>pathid</td>
<td>pathid</td>
</tr>
<tr>
<td>left</td>
<td>right</td>
</tr>
<tr>
<td>isbn</td>
<td>isbn</td>
</tr>
<tr>
<td>title</td>
<td>title</td>
</tr>
<tr>
<td>author</td>
<td>author</td>
</tr>
<tr>
<td>publisher</td>
<td>publisher</td>
</tr>
<tr>
<td>year</td>
<td>year</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”

<table>
<thead>
<tr>
<th>Dewey pid</th>
<th>Tag</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>biblio</td>
<td>graph</td>
</tr>
<tr>
<td>2</td>
<td>book</td>
<td>title</td>
</tr>
<tr>
<td>3</td>
<td>author</td>
<td>author</td>
</tr>
<tr>
<td>4</td>
<td>publisher</td>
<td>year</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Dewey-order encoding: queries

- `//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?

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

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”

<table>
<thead>
<tr>
<th>Dewey pid</th>
<th>Tag</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>biblio</td>
<td>ggraph</td>
</tr>
<tr>
<td>2</td>
<td>book</td>
<td>title</td>
</tr>
<tr>
<td>3</td>
<td>author</td>
<td>author</td>
</tr>
<tr>
<td>4</td>
<td>publisher</td>
<td>year</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Book (ISBN, price, title_id, ...)</th>
<th>Title (id, PCDATA_id)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCDATA(id, value)</td>
<td></td>
</tr>
</tbody>
</table>
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!
- 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_pos, 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
  - <book ISBN="...">...
  - <publisher>
    - <name>/</name><address>...</address>
  - </publisher>
  - </book>
- With no inlining at all
  - book(ISBN, publisher_id, publisher_name, publisher_address)
- With inlining
  - book(ISBN, name_id, PCDATA_value, publisher_address_PCDATA_value)

Queries

- book(ISBN, price, title, publisher, year),
  - book_author(ISBN, author_pos, author_id),
  - book_section(ISBN, section_pos, section_id),
  - 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
  - //book/[section
    - SELECT title FROM book UNION ALL
    - SELECT title FROM section

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

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 NULL’s; order determines grouping → messy
## 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