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 $\rightarrow$ 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

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 iid’s
  - Need indexes for efficiency, e.g., Element(tag), Text(value)

Node/edge-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>…
</bibliography>
```
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 e2.eid = c3.eid AND c3.child = t.tid
                AND t.value = 'Abiteboul')
    AND a.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

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

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

Interval-based: example

```xml
<library>
  <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>
  ...
</library>
```

- Where did ElementChild go?
  - E1 is the parent of E2 iff:

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**
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
- \( \text{Element}(\text{pathid, left, right, ...}) \), \( \text{Path}(\text{pathid, path}) \), ... 
  - \( \text{path} \) is a label path starting from the root 
  - Why are \( \text{left} \) and \( \text{right} \) still needed?

<table>
<thead>
<tr>
<th>pathid</th>
<th>left</th>
<th>right</th>
<th>path</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>999</td>
<td>/bibliography</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>21</td>
<td>/bibliography/book</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>8</td>
<td>/bibliography/book/title</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>8</td>
<td>/bibliography/book/author</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>11</td>
<td>/bibliography/book/author</td>
</tr>
<tr>
<td>6</td>
<td>12</td>
<td>14</td>
<td>/bibliography/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 path’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"

![Diagram of Dewey-order encoding]

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?

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) ...>
<!ELEMENT PCDATA (#PCDATA) ...>
```
Handling * and + in DTD

- What if an element can have any number of children?
- Example: Book can have multiple authors
  - book(ISBN, price, title_id, author_id, publisher_id, year_id)?
  - 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
  - <book ISBN="…">…
    <publisher>…
      <name>…</name>
      <address>…</address>
    </publisher>…
  </book>
- With no inlining at all
    publisher(id, name_id, address_id)
    name(id, PCDATA_id)
    address(id, PCDATA_id)
- With inlining
  - book(ISBN, publisher_name_PCDATA_value, publisher_address_PCDATA_value)
Queries

  \texttt{book\_author(ISBN,\textit{author})}, \texttt{book\_section(ISBN,\textit{section}id), section(id,\textit{title},\textit{text}), section\_section(id,\textit{section}pos,\textit{section}id)}

- \texttt{//title}
  \quad \begin{align*}
  & \text{(SELECT title FROM book) UNION ALL} \\
  & \text{(SELECT title FROM section)};
  \\
  \text{These queries only work for the given DTD}
  \\
  & \text{//section/title}
  \\
  & \text{SELECT title FROM section;}
  \\
  & \text{//bibliography/book[author="Abiteboul"]/@price}
  \\
  & \text{SELECT price FROM book, book\_author}
  \\
  \\
  & \text{//book/title}
  \\
  & \text{(SELECT title FROM book) UNION ALL} \\
  & \text{(SELECT title FROM section)}
  \\
  \end{align*}

Pros and cons of inlining

- Not always applicable
  - * and +, recursive schema (e.g., \textit{section})
- Fewer joins
- More "scattering" (e.g., there is no longer any table containing all titles; author information is scattered \textit{across book, section, etc.})
  \quad \begin{align*}
  & \text{Heuristic: do not inline elements that can be shared}
  \\
  \end{align*}

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 \rightarrow bad
  - Option 2: return two tables, one with authors and the other with references \rightarrow join is done as post processing
  - Option 3: return one table with all author and reference columns; pad with \texttt{NULL}’s; order determines grouping \rightarrow messy

\quad \begin{align*}
  & \text{Simple results are fine}
  \\
  & \text{Simple grouping is fine (e.g., books with multiple authors)}
  \\
  & \text{Complex results are problematic (e.g., books with multiple authors and multiple references)}
  \\
  & \text{One SQL query returns one table, whose columns cannot store sets}
  \\
  & \text{Option 1: return one table with all combinations of authors and references \rightarrow bad}
  \\
  & \text{Option 2: return two tables, one with authors and the other with references \rightarrow join is done as post processing}
  \\
  & \text{Option 3: return one table with all author and reference columns; pad with \texttt{NULL}’s; order determines grouping \rightarrow messy}
  \\
\end{align*}
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