Supporting XML in Relational Databases

CPS 196.3
Introduction to Database Systems

Approaches to supporting XML

- Text files
- Specialized XML DBMS
  - Lore (Stanford), Strudel (AT&T), eXist (open-source), Tamino/Quip (Software AG), 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

- Just store XML text 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?
  - Well-formed XML $\rightarrow$ generic relational schema for graph-structured data
  - Valid XML $\rightarrow$ special relational schema based on DTD

Storing well-formed XML

- Element(eid, tag)
- Attribute(eid, attrName, attrValue)
  - Attribute order does not matter
  - Key: (eid, attrName)
- ElementChild(eid, pid, child)
  - pid specifies the ordering of children
  - child references either Element(eid) or Text(tid)
  - Keys: (eid, pid), (child)
- 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)

Mapping data

<table>
<thead>
<tr>
<th>Element</th>
<th>ElementChild</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>Tag</td>
</tr>
<tr>
<td>41</td>
<td>Foundations of Database</td>
</tr>
<tr>
<td>42</td>
<td>ISBN-10</td>
</tr>
<tr>
<td>43</td>
<td>price</td>
</tr>
<tr>
<td>44</td>
<td>Title</td>
</tr>
<tr>
<td>45</td>
<td>Author</td>
</tr>
<tr>
<td>46</td>
<td>Publisher</td>
</tr>
<tr>
<td>47</td>
<td>Year</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attribute</th>
<th>oid</th>
<th>attrName</th>
<th>attrValue</th>
</tr>
</thead>
<tbody>
<tr>
<td>41</td>
<td>isbn</td>
<td>ISBN-10</td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>price</td>
<td>8.0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Text</th>
<th>oid</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>41</td>
<td>Foundations of Database</td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>ISBN-10</td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>price</td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>Title</td>
<td></td>
</tr>
<tr>
<td>45</td>
<td>Author</td>
<td></td>
</tr>
<tr>
<td>46</td>
<td>Publisher</td>
<td></td>
</tr>
<tr>
<td>47</td>
<td>Year</td>
<td></td>
</tr>
</tbody>
</table>
Mapping queries

- //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

Another query mapping example

- //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 e2.eid = a.eid
    AND a.attrName = 'price';

Storing valid XML

- 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

们

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
    - How about book?

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(id, value)
  - Pros and cons of inlining
    - Fewer joins!
    - May create “scattering”: There is no longer any table containing
      all authors; author information is scattered across book, article, etc.
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:

```xml
<book ISBN="...">...
<publisher>
  <name>...</name>
  <address>...</address>
</publisher>...
</book>
```

- With no inlining at all:

```sql
publisher(id, name_id, address_id)
name(id, PCDATA_id)
address(id, PCDATA_id)
```

- With inlining:

```sql
book(ISBN, publisher_name_PCDATA_value, publisher_address_PCDATA_value)
publisher(name_id, address_id)
name(id, PCDATA_id)
address(id, PCDATA_id)
```

Queries

  - `//title` (SELECT title FROM book) UNION ALL (SELECT title FROM section);
    - These queries only work for the given DTD
  - `//title` (SELECT title FROM book) UNION ALL (SELECT title FROM section);
  - `//book//title` (SELECT title FROM book) UNION ALL (SELECT title FROM section);

Comparison of approaches

- Generic relational schema
  - Flexible; no DTD needed
  - Queries are easy to formulate
    - Translation from XPath can be easily automated
  - Queries involve lots of joins and are expensive
- DTD-based relational schema
  - Need to know DTD to design the relational schema
  - Query formulation requires knowing DTD and schema
  - Queries are more efficient