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 tree-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, pos, child)
  - pos specifies the ordering of children
  - child references either Element(eid) or Text(tid)
  - Keys: (eid, pos), (child)
- 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)

Mapping data

<table>
<thead>
<tr>
<th>Bibliography</th>
<th>Element</th>
<th>ElementChild</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>tag</td>
<td>eid</td>
</tr>
<tr>
<td>40</td>
<td>Bibliography</td>
<td>41</td>
</tr>
<tr>
<td>41</td>
<td>movie</td>
<td>42</td>
</tr>
<tr>
<td>42</td>
<td>title</td>
<td>43</td>
</tr>
<tr>
<td>43</td>
<td>author</td>
<td>44</td>
</tr>
<tr>
<td>44</td>
<td>ISBN</td>
<td>45</td>
</tr>
<tr>
<td>45</td>
<td>pubDate</td>
<td>46</td>
</tr>
<tr>
<td>46</td>
<td>publisher</td>
<td>47</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attribute</th>
<th>id</th>
<th>attrName</th>
<th>attrValue</th>
</tr>
</thead>
<tbody>
<tr>
<td>48</td>
<td>ISBN-10</td>
<td>price</td>
<td>80.00</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Text</th>
<th>id</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>49</td>
<td>Foundations of Databases</td>
<td>50</td>
</tr>
<tr>
<td>51</td>
<td>Hull</td>
<td>52</td>
</tr>
<tr>
<td>52</td>
<td>slope</td>
<td>53</td>
</tr>
<tr>
<td>53</td>
<td>VeriSign Security Solutions</td>
<td>54</td>
</tr>
<tr>
<td>54</td>
<td>Addison Wesley</td>
<td>55</td>
</tr>
<tr>
<td>55</td>
<td>1995</td>
<td>56</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, ElementChild c2,
     Element e3,
     Attribute a
WHERE e1.tag = 'bibliography'
AND e2.tag = 'book'
AND e3.tag = 'author'
AND a.attrName = 'price'
AND e1.eid = c1.eid AND c1.child = e2.eid
AND e2.eid = c2.eid AND c2.child = e3.eid
AND e2.eid = a.eid
AND e3.eid IN (SELECT eid FROM ElementChild, Text
WHERE child = tid
AND value = 'Abiteboul');
```

Mapping //

```
//book/title
SELECT * FROMReachableFromBook r, ElementChild c
WHERE r.eid = c.eid
AND tag = 'title';
```

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

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

```
book(IsBN, price, title_id, …)
title_id, PCDATA_id)
PCDATA_id(value)
```

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
  - book(IsBN, price, title_id, publisher_id, year_id)
  - book_author(IsBN, author_id)
  - BCNF decomposition in action!
- A further optimization: merge book_author into author
- Need to add position information if ordering is important
  - book_author(IsBN, author_pos, author_id)
  - How about book?

Inlining

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

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

- With no inlining at all
- With inlining
  - book(IsBN, publisher_id)
  - publisher_id, name_id, address_id)
  - publisher_name_PCDATA_value,
  - name_id, PCDATA_id)
  - publisher_address_PCDATA_value
  - address_id, PCDATA_id)
Queries

- $\text{book(ISBN, price, title, publisher, year),}\,$
  $\text{book\_author(ISBN, author),}\,$
  $\text{book\_section(ISBN, section\_id),}\,$
  $\text{section(id, title, text),}\,$
  $\text{section\_section(id, section\_pos, section\_id).}\,$

//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//title
- (SELECT title FROM book) UNION ALL
- (SELECT title FROM section);

Comparison of approaches

- **Generic relational schema**
  - Flexible; no DTD needed
  - Queries easy to formulate
    - Translation from XPath can be easily automated
    - Queries involve lots of join 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