XML-Relational Mapping

CPS 116
Introduction to Database Systems

Announcements (Tue. Nov. 1)

- Homework #3 deadline extended to next Tuesday
- Project milestone #2 due in 1½ weeks

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(eid, tag)
- Attribute(eid, attrName, attrValue) Key: (eid, attrName)
  - 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 iid
  - Need to “invent” lots of id’s
  - Need indexes for efficiency, e.g., Element(tag), Text(value)

Node/edge-based: example

```xml
<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>
```

Element | ElementChild
--------|------------------
| | e0 1 e1
| | e1 1 e2
| | e1 2 e3
| | e1 3 e4
| | e1 4 e5
| | e1 5 e6
| | e1 6 e7
| | e2 1 t0
| | e3 1 t1
| | e4 1 t2
| | e5 1 t3
| | e6 1 t4
| | e7 1 t5

Element | Tag | Value
--------|-----|------
e0 | bibliography |
e1 | book | ISBN-10
| | price | 80.00
| | author | Abiteboul
| | author | Hull
| | author | Vianu
| | publisher | Addison Wesley
| | year | 1995

e1 | title | Foundations of Databases
| | author | Abiteboul
| | author | Hull
| | author | Vianu

```

Attribute | Value
--------|------
| ISBN-10 |
| price | 80

Text | Value
--------|------
| Foundations of Databases
| Addison Wesley
| 1995
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"]/title
  - 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 a.eid = e2.eid
    AND a.attrName = 'price';

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

- Where did ElementChild go?
  - E1 is the parent of E2 if:
    [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>Element</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>pathid</td>
<td>pathid</td>
</tr>
<tr>
<td>left</td>
<td>path</td>
</tr>
<tr>
<td>right</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

- 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>Element (dewey_pid, tag)</th>
<th>Text (dewey_pid, value)</th>
<th>Attribute (dewey_pid, attrName, attrValue)</th>
</tr>
</thead>
<tbody>
<tr>
<td>bibilography</td>
<td>book</td>
<td>title author author author publisher year</td>
</tr>
<tr>
<td></td>
<td></td>
<td>title_id</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PCDATA_id</td>
</tr>
<tr>
<td></td>
<td></td>
<td>value</td>
</tr>
</tbody>
</table>

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)>…
]> 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
  - 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
  - With no inlining at all
    - `book(ISBN, publisher_name_PCDATA_value, publisher_address_PCDATA_value)`
  - With inlining
    - `book(ISBN, publisher_name_PCDATA_value, publisher_address_PCDATA_value)`

Queries

  - //title
    - `[SELECT title FROM book] UNION ALL [SELECT title FROM section]`
  - //section/title
    - `[SELECT title FROM section]`
  - //bibliography/book[author="Abiteboul"]/@price
  - //book/title
    - `[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