XML-Relational Mapping

CPS 116
Introduction to Database Systems

Announcements (November 1)

- Homework #3 due next Tuesday
  - Yi will conduct a help session next Monday
  - Time/location will be announced by this weekend
- Project milestone #2 due in one week

Approaches to XML processing

- Text files (!)
- Specialized XML DBMS
  - Lore (Stanford), Strudel (AT&T), Timber (Michigan), MonetDB/XQuery (CW1, Netherlands), Tamino (Software AG), eXist, Sedna, Apache Xindice, XML:DB API initiative…
  - Still a long way to go
- Object-oriented DBMS
  - ObjectStore, ozone, …
  - Not as mature as relational DBMS
- Relational (and object-relational) DBMS
  - Middleware and/or object-relational extensions
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)** 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 id’s
  - Need indexes for efficiency, e.g., Element(tag), Text(value)

Node/edge-based: example

<table>
<thead>
<tr>
<th>Element</th>
<th>ElementChild</th>
<th>Attribute</th>
<th>Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>61</td>
<td>bibliography</td>
<td>ISBN 1559-10</td>
<td>62</td>
</tr>
<tr>
<td>61</td>
<td>bibliography</td>
<td>price 80</td>
<td>62</td>
</tr>
<tr>
<td>63</td>
<td>book</td>
<td>64</td>
<td>62</td>
</tr>
<tr>
<td>64</td>
<td>title</td>
<td>65</td>
<td>62</td>
</tr>
<tr>
<td>65</td>
<td>author</td>
<td>66</td>
<td>64</td>
</tr>
<tr>
<td>66</td>
<td>publisher</td>
<td>67</td>
<td>65</td>
</tr>
<tr>
<td>67</td>
<td>year</td>
<td>68</td>
<td>65</td>
</tr>
<tr>
<td>68</td>
<td>price</td>
<td>69</td>
<td>67</td>
</tr>
<tr>
<td>69</td>
<td>70</td>
<td>71</td>
<td>67</td>
</tr>
<tr>
<td>70</td>
<td>71</td>
<td>72</td>
<td>67</td>
</tr>
<tr>
<td>71</td>
<td>72</td>
<td>73</td>
<td>67</td>
</tr>
<tr>
<td>72</td>
<td>73</td>
<td>74</td>
<td>67</td>
</tr>
<tr>
<td>73</td>
<td>74</td>
<td>75</td>
<td>67</td>
</tr>
<tr>
<td>74</td>
<td>75</td>
<td>76</td>
<td>67</td>
</tr>
<tr>
<td>75</td>
<td>76</td>
<td>77</td>
<td>67</td>
</tr>
<tr>
<td>76</td>
<td>77</td>
<td>78</td>
<td>67</td>
</tr>
<tr>
<td>77</td>
<td>78</td>
<td>79</td>
<td>67</td>
</tr>
<tr>
<td>78</td>
<td>79</td>
<td>80</td>
<td>67</td>
</tr>
<tr>
<td>79</td>
<td>80</td>
<td>81</td>
<td>67</td>
</tr>
</tbody>
</table>

Table:

- **Element**: (tag, eid)
- **ElementChild**: (child, pos)
- **Attribute**: (attrName, attrValue)
- **Text**: (tid, value)
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 e2.eid = a.eid
    AND a.attrName = 'price';

Node/edge-based: descendent-or-self

- //book/title
  - Requires SQL3 recursion
  - WITH 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
<biblpgraphy>
  <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>
</biblpgraphy>
```

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

```
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!*

```
//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

- Element(pathid, left, right, …), Path(pathid, path), …
  - path is a label path starting from the root
  - Why are left and right still needed?

<table>
<thead>
<tr>
<th>Element</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>pathid</td>
<td>path</td>
</tr>
<tr>
<td>1</td>
<td>/bibliography</td>
</tr>
<tr>
<td>2</td>
<td>/bibliography/book</td>
</tr>
<tr>
<td>3</td>
<td>/bibliography/book/title</td>
</tr>
<tr>
<td>4</td>
<td>/bibliography/book/author</td>
</tr>
<tr>
<td>5</td>
<td>/bibliography/book/author</td>
</tr>
<tr>
<td>6</td>
<td>/bibliography/book/author</td>
</tr>
<tr>
<td>7</td>
<td>/bibliography/book/author</td>
</tr>
<tr>
<td>8</td>
<td>/bibliography/book/author</td>
</tr>
<tr>
<td>9</td>
<td>/bibliography/book/author</td>
</tr>
<tr>
<td>10</td>
<td>/bibliography/book/author</td>
</tr>
<tr>
<td>11</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 pathid’s with Element
- //book[publisher='Prentice Hall']/title
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"

\[
\text{Element}(\text{dewey_pid}, \text{tag}) \\
\text{Text}(\text{dewey_pid}, \text{value}) \\
\text{Attribute}(\text{dewey_pid}, \text{attrName}, \text{attrValue})
\]

Dewey-order encoding: queries

- Examples:
  - //title
  - //section/title
  - //book/title
  - //book[publisher='Prentice Hall']/title

- 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

\[
\text{book}(\text{ISBN}, \text{price}, \text{title}, \ldots) \\
\text{title}(\text{id}, \text{PCDATA_id}) \\
\text{PCDATA}(\text{id}, \text{value})
\]
Handling * and + in DTD

- What if an element can have any number of children?
- Example: Book can have multiple authors

- 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>
    <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)`
    - `publisher(id, name_id, address_id)`
    - `name(id, PCDATA_id)`
    - `address(id, PCDATA_id)`
Queries

- book(ISBN, price, title, publisher, year),
  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
  - //book/title

Pros and cons of inlining

- Not always applicable
  - * and +, recursive schema (e.g., section)

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