**Announcements (March 29)**
- Homework #3 assigned today
  - Due in two weeks (April 12)
- Reading assignment due this Wednesday
  - Two VLDB papers on native XML databases
- Course project milestone 2 due this Thursday

**Approaches to XML processing**
- Text files (!)
- Specialized XML DBMS
  - Lore (Stanford), Strudel (AT&T), Tamino/QuiP (Software AG), X-Hive, Timber (Michigan), 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**
- 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
  - 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**

```sql
<table>
<thead>
<tr>
<th>Bibliography</th>
<th>Element</th>
<th>Text</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>eid</td>
<td>tid</td>
</tr>
<tr>
<td></td>
<td>00</td>
<td></td>
</tr>
<tr>
<td></td>
<td>01</td>
<td></td>
</tr>
<tr>
<td></td>
<td>02</td>
<td></td>
</tr>
<tr>
<td></td>
<td>03</td>
<td></td>
</tr>
<tr>
<td></td>
<td>04</td>
<td></td>
</tr>
<tr>
<td></td>
<td>05</td>
<td></td>
</tr>
<tr>
<td></td>
<td>06</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bibliography</th>
<th>Element</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>eid</td>
<td>attrName</td>
</tr>
<tr>
<td></td>
<td>00</td>
<td>ISBN-10</td>
</tr>
<tr>
<td></td>
<td>01</td>
<td>price</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bibliography</th>
<th>ElementChild</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>eid</td>
<td>attrName</td>
</tr>
<tr>
<td></td>
<td>00</td>
<td>ISBN-10</td>
</tr>
<tr>
<td></td>
<td>01</td>
<td>price</td>
</tr>
<tr>
<td></td>
<td>02</td>
<td>value</td>
</tr>
</tbody>
</table>
```
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, Text t
      WHERE e2.eid = c2.eid
      AND c2.child = e3.eid
      AND e3.tag = 'author'
      AND e2.eid = c1.eid
      AND c1.child = t.tid
      AND t.value = 'Abiteboul')
    AND e2.eid = a.eid
    AND a.attrName = 'price';

Node/edge-based: descendant-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 left
- Attribute(left, attrName, attrValue)
- Text(left, level, value)

  Where did ElementChild go?
  * E1 is the parent of E2 iff:
    [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 = 'section'
    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 with "dynamic interval encoding"
  - DeHaan et al. SIGMOD 2003
  - Looks hairy, but with some special tweaks to the relational engine, it actually performs better than many of the currently available native XQuery products!
  - Set-oriented processing helps!

A path-based mapping

Label-path encoding

- `Element(pathid, left, right, value), 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>pathid</th>
<th>left</th>
<th>right</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>999</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>1</td>
<td>21</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>8</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
<td>8</td>
<td>11</td>
</tr>
</tbody>
</table>

DeHaan et al. SIGMOD 2003

Label-path encoding: queries

- Simple path expressions with no conditions
  
  - //book/title
  - Perform string matching on Path
  - Join qualified pathid's with Element

- Path expression with attached conditions needs to be broken down, processed separately, and joined back
  
  - //book[publisher='Prentice Hall']/title
  - Evaluate //book/title
  - Evaluate //book/publisher[text()='Prentice Hall']
  - Join to ensure title and publisher belong to the same book

How?

Label-path encoding: queries

- //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

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>bibliogaphy</th>
<th>book</th>
<th>title</th>
<th>author</th>
<th>author</th>
<th>publisher</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.1</td>
<td>1.1.1</td>
<td>1.1.2</td>
<td>1.1.3</td>
<td>1.1.4</td>
<td>1.1.5</td>
</tr>
<tr>
<td>2</td>
<td>1.2</td>
<td>1.2.1</td>
<td>1.2.2</td>
<td>1.2.3</td>
<td>1.2.4</td>
<td>1.2.5</td>
</tr>
<tr>
<td>3</td>
<td>1.3</td>
<td>1.3.1</td>
<td>1.3.2</td>
<td>1.3.3</td>
<td>1.3.4</td>
<td>1.3.5</td>
</tr>
<tr>
<td>4</td>
<td>1.4</td>
<td>1.4.1</td>
<td>1.4.2</td>
<td>1.4.3</td>
<td>1.4.4</td>
<td>1.4.5</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?

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>bibliogaphy</th>
<th>book</th>
<th>title</th>
<th>author</th>
<th>author</th>
<th>publisher</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.1</td>
<td>1.1.1</td>
<td>1.1.2</td>
<td>1.1.3</td>
<td>1.1.4</td>
<td>1.1.5</td>
</tr>
<tr>
<td>2</td>
<td>1.2</td>
<td>1.2.1</td>
<td>1.2.2</td>
<td>1.2.3</td>
<td>1.2.4</td>
<td>1.2.5</td>
</tr>
<tr>
<td>3</td>
<td>1.3</td>
<td>1.3.1</td>
<td>1.3.2</td>
<td>1.3.3</td>
<td>1.3.4</td>
<td>1.3.5</td>
</tr>
<tr>
<td>4</td>
<td>1.4</td>
<td>1.4.1</td>
<td>1.4.2</td>
<td>1.4.3</td>
<td>1.4.4</td>
<td>1.4.5</td>
</tr>
</tbody>
</table>

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

- Example: DTD for books

  ```xml
  <!DOCTYPE bibliography[
    <!ELEMENT bibliography (book)*>
    <!ELEMENT book (title, [author]+, ...)
    <!ELEMENT title (#PCDATA)>
    <!ELEMENT booktitle PCDATA>
    <!ELEMENT author PCDATA>
    <!ELEMENT publisher 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

**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

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

- With no inlining at all
- With inlining

```sql
WITH book(ISBN, publisher_id) AS
  SELECT ISBN, publisher_name, PCDATA_value, publisher_address PCDATA_value
FROM book
JOIN publisher ON book.publisher_id = publisher.id
JOIN PCDATA ON book.PCDATA_id = PCDATA.id
```

**Queries**

- book(ISBN, price, title, publisher, year),
- section(id, title, text), section_section(id, section_pos, section_id)

```sql
//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))
```

**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: one SQL query only returns a single table; columns cannot contains sets or structures
  - E.g., books with multiple authors and multiple references
  - Option 1: one table with all combo of authors/references ⇒ bad
  - Option 2: two tables, one w/ authors and the other w/ references ⇒ join is done as post processing
  - Option 3: sorted “union” of NULL-padded authors and references
Comparison of approaches

- **Schema-oblivious**
  - Flexible and adaptable; no DTD needed
  - Queries are easy to formulate
    - Translation from XPath/XQuery 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