Announcements (March 18)

- Midterm sample solution available outside my office
- Course project milestone 2 due March 30
- Homework #3 due April 6
- Talk by Amol Deshpande
  - Adaptive Query Processing to Handle Estimation Errors
  - Monday, 11:30am-12:30pm, D106
- Reading assignment due next Monday
  - Two VLDB papers on native XML databases

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 $\rightarrow$ generic relational schema
    - Node/edge-based mapping for graphs
    - Interval-based mapping for trees
    - Path-based mapping for trees
  - Schema-aware mapping:
    - valid XML $\rightarrow$ special relational schema based on DTD

Node/edge-based: schema

- $\text{Element}(\text{eid}, \text{tag})$
- $\text{Attribute}(\text{eid}, \text{attrName}, \text{attrValue})$
  - Key: $(\text{sid}, \text{attrName})$
  - Attribute order does not matter
- $\text{ElementChild}(\text{eid}, \text{pos}, \text{child})$
  - $(\text{pos})$ specifies the ordering of children
  - $\text{child}$ references either $\text{Element}(\text{sid})$ or $\text{Text}(\text{tid})$
- $\text{Text}(\text{tid}, \text{value})$
  - $\text{tid}$ cannot be the same as any $\text{sid}$
  - Need to “invent” lots of $\text{sid}$’s
  - Need indexes for efficiency, e.g., $\text{Element}(\text{tag})$, $\text{Text}(\text{value})$

Node/edge-based: example

XML-Relational Mapping

CPS 216
Advanced Database Systems
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 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: example

```xml
title author author author publisher year
```

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!
How about XQuery?
DeHaan et al. SIGMOD 2003

- Evaluating an XQuery expression results in a sequence of environments
  - An environment \( E \) maps each query variable \( v \) to its value: a forest of XML trees (a node-set) \( f_v \)
- Encode using tables with "dynamic intervals"
  - Table \( F \): increasing sequence of integers, one per environment
  - For each query variable \( v \), create a table \( T_v \) representing the value of \( v \) in all environments
    - Sorted on \( l \) to support efficient processing
    - Different environments form non-overlapping regions

Translating /
- Given \( T_v \) for values of \( v \), compute \( v/\text{name} \)
  - Compute \( v/\text{name} \)
    
    \[
    \text{CREATE VIEW T1 AS}
    \text{SELECT * FROM T_v t}
    \text{WHERE EXISTS(SELECT * FROM T_v WHERE l<t.l AND t.r<r);}
    \]
  - Compute \( \text{name} \) roots of \( v/\text{name} \)
    
    \[
    \text{CREATE VIEW T2 AS}
    \text{SELECT * FROM T1 t}
    \text{WHERE s = \text{name}}
    \text{AND NOT EXISTS(SELECT * FROM T1 WHERE l<t.l AND t.r<r);}
    \]
  - Compute \( v/\text{name} \)
    
    \[
    \text{CREATE VIEW T3 AS}
    \text{SELECT * FROM T2 t}
    \text{WHERE EXISTS(SELECT * FROM T2 WHERE l<t.l AND t.r<r);}
    \]

Translating //
- Given \( T_v \) for values of \( v \), compute \( v//\text{*} \)
  - How about:
    
    \[
    \text{CREATE VIEW T1 AS}
    \text{SELECT * FROM T_v t1, T_v t2}
    \text{WHERE t1.1<=t2.1 AND t2.r<=t.t1;}
    \]
  - Fix: let \( w = \max\{t.r | t \in T_v\} \)
    
    \[
    \text{CREATE VIEW T1 AS}
    \text{SELECT t2.s, t1.l*w + t2.l, t1.l*w + t2.r}
    \text{FROM T_v t1, T_v t2}
    \text{WHERE t1.1<=t2.1 AND t2.r<=t2.1;}
    \]
  - What would this do to the size of \( T_v \)?

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
  - 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

- \text{Element(pathid, left, right, value), Path(pathid, path)}
  - \text{path} is a label path starting from the root
  - Why are \text{left} and \text{right} still needed? To preserve structure

<table>
<thead>
<tr>
<th>Element</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<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>
</tbody>
</table>

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”

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

Dewey-order encoding: queries

- Examples:
  - //title
  - //section/title
  - //book/title
  - //book[publisher='Prentice Hall']/title
  - Works similarly as interval-based mapping
  - 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 \rightarrow table name
  - Attributes \rightarrow columns
    - If one exists, ID attribute \rightarrow key column; otherwise, need to “invent” a key
    - IDREF attribute \rightarrow foreign key column
  - Children of the element \rightarrow foreign key columns
    - Ordering of columns encodes ordering of children

```
<!DOCTYPE bibliography [ ...
  <!ELEMENT book (title_id, ...)
  <!ELEMENT title (PCDATA)...
  <!ELEMENT author (PCDATA)...
]>

  ABCNF?

Idea: create another table to track such relationships
  ABCNF decomposition in action!
  A further optimization: merge book_author into author

Need to add position information if ordering is important
```

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?
    ABCNF?
  
  Idea: create another table to track such relationships
    ABCNF decomposition in action!
  
  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>...<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)
  ```

Queries

```sql
//title
(SELECT title FROM book) UNION ALL
(SELECT title FROM section);
```

```sql
//section/title
SELECT title FROM section;
```

```sql
//bibliography/book[author="Abiteboul"]/@price
(SELECT price FROM book, book_author
```

```sql
//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 can only return a single 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

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