XML-Relational Mapping

CPS 216
Advanced Database Systems

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 → 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

```
<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

eid	*tag*
e0	bibliography
e1	book
e2	title
e3	author
e4	author
e5	author
e6	publisher
e7	year

ElementChild

eid	*pos*	*child*
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

Attribute

eid	*attrName*	*attrValue*
e1	ISBN	ISBN-10
e1	price	80

Text

tid	*value*
t0	Foundations of Databases
t1	Abiteboul
t2	Hull
t3	Vianu
t4	Addison Wesley
t5	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"]/@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

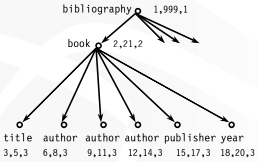
  -
    ```
    WITH ReachableFromBook(id) AS
     ((SELECT eid FROM Element WHERE tag = 'book')
      UNION ALL
      (SELECT c.child
       FROM ReachableFromBook r, ElementChild c
       WHERE r.eid = c.eid))
    SELECT eid
    FROM Element
    WHERE eid IN (SELECT * FROM ReachableFromBook)
    AND 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:
    $[E1.left, E1.right] \supset [E2.left, E2.right]$, and
    $E1.level = E2.level - 1$

Interval-based: example

```
1<bibliography>
   2<book ISBN="ISBN-10" price="80.00">
      3<title>4Foundations of Databases</title>5
      6<author>7Abiteboul</author>8
      9<author>10Hull</author>11
      12<author>13Vianu</author>14
      15<publisher>16Addison Wesley</publisher>17
      18<year>191995</year>20
    </book>21…
  </bibliography>999
```



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`

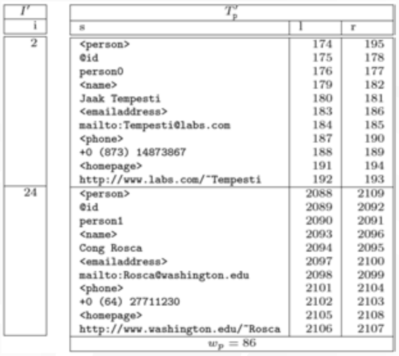
  -
    ```
    SELECT e2.left
    FROM Element e1, Element e2
    WHERE e1.tag = 'book' AND e2.tag = 'section'
    AND e1.left < e2.left AND e2.right < e1.right;
    ```
  - ☞ No recursion!

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 I: increasing sequence of integers, one per environment
  - For each query variable v, create a table T_v(*s(tring)*, *l(eft)*, *r(ight)*) representing the value of v in all environments
    - Sorted on l to support efficient processing
    - Different environments form non-overlapping regions

Example T_v



I'
i
2
24

T'_p		
s	l	r
`<person>`	174	195
@id	175	178
person0	176	177
`<name>`	179	182
Jaak Tempesti	180	181
`<emailaddress>`	183	186
mailto:Tempesti@labs.com	184	185
`<phone>`	187	190
+0 (873) 14873867	188	189
`<homepage>`	191	194
http://www.labs.com/~Tempesti	192	193
`<person>`	2088	2109
@id	2089	2092
person1	2090	2091
`<name>`	2093	2096
Cong Rosca	2094	2095
`<emailaddress>`	2097	2100
mailto:Rosca@washington.edu	2098	2099
`<phone>`	2101	2104
+0 (64) 27711230	2102	2103
`<homepage>`	2105	2108
http://www.washington.edu/~Rosca	2106	2107
$w_p = 86$		

Translating /

- Given T_v for values of v, compute v/`name`
  - Compute v/*

```
CREATE VIEW T1 AS
SELECT * FROM Tv t
WHERE EXISTS(SELECT * FROM Tv WHERE l<t.l AND t.r<r);
```

  - Compute `name` roots of v/*

```
CREATE VIEW T2 AS
SELECT * FROM T1 t
WHERE s = 'name'
AND NOT EXISTS(SELECT * FROM T1 WHERE l<t.l AND t.r<r);
```

  - Compute v/`name`

```
CREATE VIEW T3 AS
SELECT * FROM Tv t
WHERE EXISTS(SELECT * FROM T2 WHERE l<=t.l AND t.r<=r);
```

Translating //

- Given T_v for values of v, compute v//*
  - How about:

```
CREATE VIEW T1 AS
SELECT t2.* FROM Tv t1, Tv t2
WHERE t1.l<=t2.l AND t2.r<=t2.l;
```

    - Almost there, but environments overlap now
  - Fix: let $w = \max\{t.r \mid t \in T_v\}$

```
CREATE VIEW T1 AS
SELECT t2.s, t1.l*w + t2.l, t1.l*w + t2.r
FROM Tv t1, Tv t2
WHERE t1.l<=t2.l AND t2.r<=t2.l;
```

  - What would this do to the size of T_v?

Translating `for`

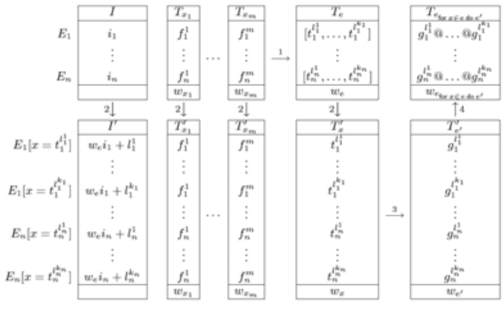



Figure 6: TRANSLATION OF "**for** $x \in e$ **do** e'" IN THE ENVIRONMENT FOR $x_1, \ldots, x_m$.

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

- ❖ *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

Element

pathid	*left*	*right*	…
1	1	999	…
2	2	21	…
3	3	5	…
4	6	8	…
4	9	11	…
4	12	14	…
…	…	…	…

Path

pathid	*path*
1	/bibliography
2	/bibliography/book
3	/bibliography/book/title
4	/bibliography/book/author
…	…

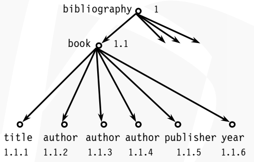
Label-path encoding: queries

- ❖ 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`

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"



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 → 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*)

Inlining

- An `author` element just has a `PCDATA` child
- Instead of using foreign keys
  - *book_author*(*ISBN*, *author_id*)
  - *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>
  </publisher>…
</book>
```

- With no inlining at all

  book(*ISBN*, *publisher_id*)
  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

- *book*(*ISBN*, *price*, *title*, *publisher*, *year*), *book_author*(*ISBN*, *author*), *book_section*(*ISBN*, *section_id*), *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`
  - `SELECT price FROM book, book_author WHERE book.ISBN = book_author.ISBN AND author = 'Abiteboul';`
- `//book//title`
  - `(SELECT title FROM book) UNION ALL (SELECT title FROM section)`

These queries only work for the given DTD

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