TDDD43

Theme 1.3: XML query processing

Fang Wei-Kleiner

http://www.ida.liu.se/~TDDD43

Announcement

- Please register for the lab! (deadline: 5.Sep.2012)
- Lab will start next week.

Approaches of XML query 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)*
  - 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 *eid*

✓ Need to "invent" lots of *id*'s

✓ Need indexes for efficiency, e.g., *Element(tag), Text(value)*

```
<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

Text

tid	value
t0	Foundation of databases
t1	Abiteboul
t2	Hull
t3	Vianu
t4	Addison Wesley
t5	1995

Attribute

eid	attrName	attrValue
e1	ISBN	ISBN-10
e1	price	80

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

More queries

- //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 e3.eid = c3.eid AND c3.child = t.tid
                AND t.value = 'Abiteboul')
AND e2.eid = a.eid
AND a.attrName = 'price';
```

Descendant-or-self

- //book//title
- Requires SQL3 recursion

```
WITH RECURSIVE 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 approach

- *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
  - Key is *left*

- *Text(left, right, level, value)*
  - Key is *left*

- *Attribute(left, attrName, attrValue)*
  - Key is *(left, attrName)*

```
<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

left	right	level	tag
1	999	1	bibliography
2	21	2	book
3	5	3	title
6	8	3	author
9	11	3	author
12	14	3	author
15	17	3	publisher
18	20	3	year
…	…	…	…

Text

left	right	level	tag
4	4	4	Foundation of databases
7	7	4	Abiteboul
10	10	4	Hull
13	13	4	Vianu
16	16	4	Addison Wesley
19	19	4	1995

Attribute

left	attrName	attrValue
2	ISBN	ISBN-10
2	price	80

```
<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

left	right	level	tag
1	999	1	bibliography
2	21	2	book
3	5	3	title
6	8	3	author
9	11	3	author
12	14	3	author
15	17	3	publisher
18	20	3	year
…	…	…	…

- Where did *ElementChild* go?

E1 is the parent of *E2* iff:

- *E1.left* < *E2.left*, *E1.right* > *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!

Interval based mapping

- Path expression steps become containment joins
- No recursion needed for descendent-or-self
- Comprehensive XQuery-SQL translation is possible

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

Element

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

Path

pathId	path
1	/bibliography
2	/bibliography/book
3	/bibliography/book/title
4	/bibliography/book/author
…	…

Path based mapping

- //book//title

SELECT E.left

FROM Element E, Path P

WHERE P.path LIKE '%/book%/title' AND E.pathId = P.pathId

✓ Perform string matching on Path
✓ Join qualified pathid's with Element

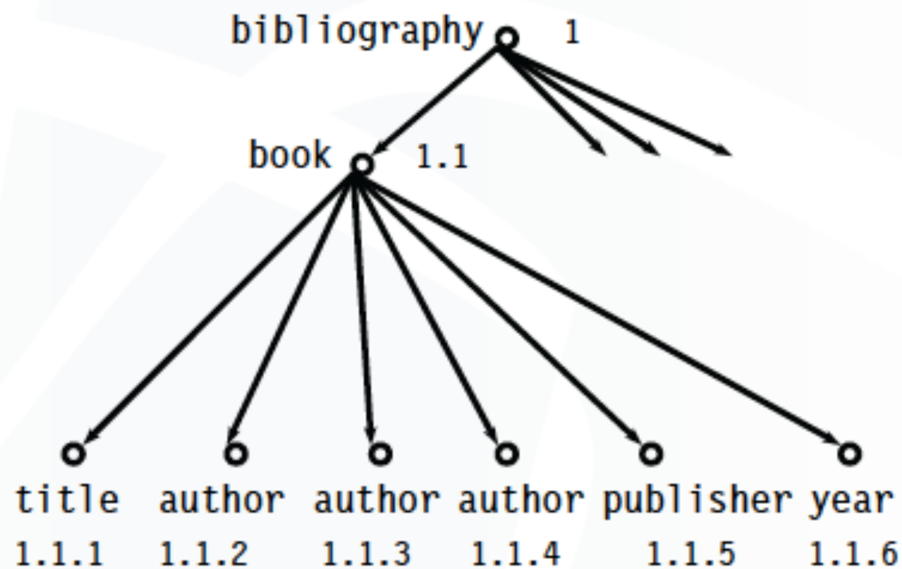
- //book[publisher='Prentice Hall']/title

- ✓ Evaluate //book/title
- ✓ Evaluate //book/publisher[text()='Prentice Hall']
- ✓ How to ensure title and publisher belong to the same book?

```
SELECT E2.left
FROM Path P1, Path P2, Path P3,Element E1, Element E2, Text T
WHERE P1.path LIKE '%/book'
AND P2.path LIKE '%/book/publisher'
AND P3.path LIKE '%/book/title'
AND E1.pathId = P1.pathId
AND E2.PathId = P3.PathId
AND T.PathId = P2.PathId
AND E1.left < T.left AND E1.right > T.right
AND E1.left < E2.left AND E1.right > E2.right
AND T.value = 'Prentice Hall'
```

Dewey order mapping

- Dewey-order encoding
- Each component of the id represents the order of the child within its parent
- Unlike label-path, this encoding "lossless"



Element(dewey_pid, tag)
Text(dewey_pid, value)
Attribute(dewey_pid, attrName, attrValue)

Dewey ordering

- 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

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
- 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!

- Need to add position information if ordering is important
- *book_author(ISBN, author_pos, author_id)*

Example

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; ← works only with this DTD

- //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)

Comparison

- 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