XML and Relational Databases

© Leonidas Fegaras
University of Texas at Arlington
Two Approaches

- **XML Publishing**
  - treats existing relational data sets as if they were XML data
  - defines an XML view of the relational data
  - poses XML queries over this view
  - similar to schema integration
    - global as view (GAV) vs local as view (LAV)
  - materializing (parts of) the view

- **XML Storage**
  - uses an RDBMS to store and query existing XML data
    - need to choose a relational schema for storing XML data
    - translate XML queries to SQL
Publishing without Views

- Constructs XML data in main memory on the fly
- Based on language extensions to SQL and modified query engine
- Requires user-defined functions for XML element construction
- Example:
  
  ```sql
  define XML constructor ARTICLE ( artId:integer, title:varchar(20), authorList:xml ) AS {
    <article id=$artId>
      <title>$title</title>
      <authors>$authorList</authors>
    </article>
  }
  
  Special function to concatenate input fragments
    - Problem: list vs set
Publishing with Support for Views

- Provides XML views over relational data
  - a view is not necessarily materialized
- Queries are XML queries over these views
  - goal: retrieve only the required fragments of relational data by pushing the computation into the relational engine as much as possible
  - we don't want to reconstruct the entire XML document from all the relational data and then extract the answer from the document
Case Study: XPERANTO

- Automatically creates a default XML view from relational tables
  - top-level elements correspond to table names
  - row elements are nested under the table elements
  - for each row element, a column corresponds to an element whose tag name is the column name and text is the column value

- Example

  Relation schema:
  Department (deptno, dname, address)
  Employee (ssn, dno, name, phone, salary)

**DTD of the default view:**

```xml
<!ELEMENT db (Department*, Employee*)>
<!ELEMENT Department (deptno, dname, address)>
<!ELEMENT Employee (ssn, dno, name, phone, salary)>
<!ELEMENT deptno (PCDATA)>
<!ELEMENT dname (PCDATA)>
...
```
The default view may be refined by a user view
- the view is defined using an XQuery

\[
\text{<info>}\{
  \text{for } \$d \text{ in view("default")/db/Departments}
  \text{for } \$e \text{ in view("default")/db/Employees[dno=\$d/deptno]}
  \text{return } \text{<employee ssn="\$e/ssn">}\$e/name,\$d/dname</employee>
\}\text{</info>}
\]

Then the actual query can be on the user view
- for $e$ in view("view")/info/employee[@ssn="123"]
  return $e/name
XPERANTO (cont.)

- It uses the XML Query Graph Model (XQGM) as internal representation
  - enables the translation from XQuery to SQL
  - exploits an XML query algebra
- It removes all XML navigation operators
  - to avoid intermediate results
- It pushes joins and selections down to the relational query engine
  - query decorrelation
Relational Schemas for XML

- Various approaches
  - **generic mapping** regardless of any schema or data knowledge
    - same for all kinds of XML data
  - **user-defined mapping** from XML to relational tables
    - mapping is inferred from DTD or XML Schema
    - mapping is derived from conceptual model
    - mapping is deduced from ontologies or domain knowledge
    - mapping is derived from query workload
Generic Mapping

- XML data can be seen as a graph
  - A graph node corresponds to an XML element
  - An edge indicates a child element (the edge label is the child tagname)
  - id/idrefs can make the XML tree into a graph

- Three ways of storing graph edges:
  - edge approach: store all edges in a single table
  - binary approach: group all edges with the same label into a separate table
  - universal table: an outer join between all tables from the binary approach

- Two ways of mapping values:
  - using a separate value table
  - inlining the values into the edge table(s)

- Usually binary approach with inlining
**A Single Table**

```sql
create table element
    ( tagname   varchar(20),
      content   varchar(100),
      begin     int      not null,
      end       int      not null,
      level     int      not null
    )

<A><B>text1</B><B>text2</B></A>

<table>
<thead>
<tr>
<th>tagname</th>
<th>content</th>
<th>begin</th>
<th>end</th>
<th>level</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>null</td>
<td>0</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>null</td>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>null</td>
<td>4</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>null</td>
<td>text1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>null</td>
<td>text2</td>
<td>5</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>
```

0 1 2 3 4 5 6 7  <-- begin/end positions
For example, the XPath query:
   //book/title
is translated into the following SQL query:
   select e2
   from element e1, element e2
   where e1.tagname = 'book'
   and e2.begin > e1.begin
   and e2.end < e1.end
   and e2.level = e1.level+1
   and e2.tagname = 'title'
The XPath query:

```
/books//book[author/name="Smith"]/title
```

is translated into:

```
select e6
from element e1, element e2, element e3,
    element e4, element e5, element e6
where e1.level = 0
    and e1.tagname = 'books'
    and e2.begin > e1.begin
    and e2.end < e1.end
    and e2.level > e1.level
    and e2.tagname = 'book'
    and e3.begin > e2.begin
    and e3.end < e2.end
    and e3.level = e2.level+1
    and e3.tagname = 'author'
    and e4.begin > e3.begin
    and e4.end < e3.end
    and e4.level = e3.level+1
    and e4.tagname = 'name'
    and e5.begin > e4.begin
    and e5.end < e4.end
    and e5.level = e4.level+1
    and e5.content = 'Smith'
    and e6.begin > e5.begin
    and e6.end < e5.end
    and e6.level = e5.level+1
    and e6.tagname = 'title'
```
Inferring the Relational Schema from DTD

- A DTD graph is generated from the DTD
  - one node for each DTD <!ELEMENT ... >
  - a node '*' for repetition
  - an arrow connects a parent element to a child element in DTD

- Two approaches:
  - Shared inlining
    - an element node corresponds to one relation
    - … but element nodes with one parent are inlined
    - … but nodes below a '*' node correspond to a separate relations
    - mutual recursive elements are always mapped to separate relations
  - Hybrid inlining
    - may inline elements even with multiple parents, below '*', or recursive
Example

<!ELEMENT DB (proceeding | book)*>  
<!ELEMENT proceeding (article*)>  
<!ELEMENT article (title,author)>  
<!ELEMENT book (editor,title)>

Shared inlining:
  proceeding(ID)  
  article(ID,parent,author)  
  title(ID,parent,title)  
  book(ID,editor)

Hybrid inlining:
  proceeding(ID)  
  article(ID,parent,author,title)  
  book(ID,editor,title)
XML Indexing

- Many approaches
- Data guides
  - based on a *structural summary*
    - the structural summary is the minimum graph that captures all valid paths to data
    - deterministic: from each node you can go to only one node via a tagname
  - the leaves are sets of nodes (the indexed data)
  - designed for evaluating XPath efficiently
  - may take the form of a DFA or a tree

```
  depts
     \--- department
          \--- student
              \--- name
                  \--- firstname
                      \--- lastname
              \--- gpa
          \--- faculty
              \--- name
                  \--- firstname
                      \--- lastname
              \--- salary
```
Inverted Index

- Inverted indexes are used in Information Retrieval (IR) in mapping words to sets of text documents that contain the word
  - typically implemented as a B+-tree having the word as a key
- Each XML element is assigned two numbers. Two choices:
  - (begin,end) which are the positions of the start/end tags of the element
  - (order,size) which are order=begin and size=end-begin
- We will use the following representation of an XML element:
  (docnum, begin:end, level) where level is the depth level of the element
- Words in PCData are represented by:
  (docnum, position, level)
- Two indexes:
  - E-index for indexing tagnames
  - T-index for indexing words
Example

\(<A><B>\text{Computer Science}</B><B>\text{Science and Engineering}</B></A>\)

\[
\begin{array}{cccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9
\end{array}
\]

\[\text{begin/end positions}\]

E-index:

\(<A>\{ (1,0:9,0) \}\)  
\(<B>\{ (1,1:4,1), (1,5:8,1) \}\)

E-index is implemented as a table with secondary index on tag

\(\begin{array}{|c|c|c|c|}
\hline
\text{tagname} & \text{doc} & \text{begin} & \text{end} & \text{level} \\
\hline
\text{A} & 1 & 0 & 9 & 0 \\
\text{B} & 1 & 1 & 4 & 1 \\
\text{B} & 1 & 5 & 8 & 1 \\
\hline
\end{array}\)

T-index:

Computer \{ (1,2,3) \}
Science \{ (1,3,3), (1,6,3) \}
Engineering \{ (1,7,3) \}
Containment Join

- XPath steps are evaluated using containment joins
  - a join that indicates that the inner element should be 'contained' inside the outer element
- For example, the XPath query `//book/title` is translated into the following SQL query:
  ```sql
  select e2
  from element e1, element e2
  where e1.tagname = "book"
  and e2.doc = e1.doc
  and e2.begin > e1.begin
  and e2.end < e1.end
  and e2.level = e1.level+1
  and e2.tagname = "title"
  ```
- It uses the E-index twice
Evaluating XPath Steps

- From path/A, we generate the SQL query
  ```sql
  select e2
  from PATH e1, element e2
  where e2.tagname = "A"
    and e2.doc = e1.doc
    and e2.begin > e1.begin
    and e2.end < e1.end
    and e2.level = e1.level+1
  
  where PATH is the SQL query that evaluates path
  ```

- From path//A, we get:
  ```sql
  select e2
  from PATH e1, element e2
  where e2.tagname = "A"
    and e2.doc = e1.doc
    and e2.begin > e1.begin
    and e2.end < e1.end
  ```
Problems

- **Advantages:**
  - you can use an existing relational query evaluation engine
  - the query optimizer will use the E-index

- **Disadvantages:**
  - many levels of query nesting
    - as many as the XPath steps
    - need query decorrelation
  - even after query unnesting, we get a join over a large number of tables
    - these are self joins because we are joining over the same table (element)
    - most commercial optimizers can handle up to 12 joins

- **Need a special evaluation algorithm for containment join**
  - based on sort-merge join
  - requires that the indexes deliver the data sorted by major order of docnum and minor order of begin/position
  - facilitates pipelining
Pipeline Processing of XPath Queries

- A pipeline is a sequence of iterators
  
  ```java
  class Iterator {
      Tuple current();  // current tuple from stream
      void open ();      // open the stream iterator
      Tuple next ();     // get the next tuple from stream
      boolean eos ();    // is this the end of stream?
  }
  ```

- An iterator reads data from the input stream(s) and delivers data to the output stream

- Connected through pipelines
  - an iterator (the producer) delivers a stream element to the output only when requested by the next operator in pipeline (the consumer)
  - to deliver one stream element to the output, the producer becomes a consumer by requesting from the previous iterator as many elements as necessary to produce a single element, etc, until the end of stream
Pipelines Pass one Tuple at a Time

- For XPath evaluation, a Tuple is a Fragment

```java
class Fragment {
    int document; // document ID
    short begin;  // the start position in document
    short end;    // the end position in document
    short level;  // depth of term in document
}
```

- E-index delivers Fragments sorted by major order of 'document' and minor order of 'begin'
XPath Steps are Iterators

class Child extends Iterator {
    String tag;
    Iterator input;
    IndexIterator ti;

    void open () { ti = new IndexIterator(tag); }

    Fragment next () {
        while (!ti.eos() && !input.eos()) {
            Fragment f = input.current();
            Fragment h = ti.current();
            if (f.document < h.document) input.next();
            else if (f.document > h.document) ti.next();
            else if (f.begin < h.begin && f.end > h.end && h.level == f.level+1) {
                ti.next();
                return h;
            } else if (lf.begin < h.begin) input.next();
            else ti.next();
        }
    }
}
Example

1  <a>
2       <b>
3           X
4       </b>
5       <b>
6           Y
7       </b>
8  </a>
9  <a>
10     <c>
11         <b>
12           Z
13         </b>
14     </c>
15     <b>
16           W
17     </b>
18  </a>

<table>
<thead>
<tr>
<th>&lt;a&gt;</th>
<th>&lt;b&gt;</th>
<th>&lt;c&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1,1:8,0)</td>
<td>(1,2:4,1)</td>
<td>(1,10:14,1)</td>
</tr>
<tr>
<td>(1,9:18,0)</td>
<td>(1,5:7,1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(1,11:13,2)</td>
<td>(1,15:17,1)</td>
</tr>
</tbody>
</table>

Query:  //a/b
XPath Evaluation Based on Iterators

- Iterators implement containment joins using sort-merge joins
  - they maintain the invariant that all fragments are sorted by document (major) and begin/position (minor) order

- They can support two modes for path evaluation
  1) starting from a specific document, evaluate an XPath query
     ```xml
document("book.xml")//book/author
     ```
  1) evaluate an XPath query against all indexed documents
     ```xml
document("*")//book/author
     ```

- The sorted lists derived from E-index/T-index may be very long
  - improvement:
    - jump over the list elements that do not contribute to the result
    - can be accomplished if the index is a B+-tree
A Problem

- Pure sort-merge join may not work in some extreme cases

  Example: //a/b

  1  <a>
  2     <a>
  3        <b>
  4            text1
  5        </b>
  6     </a>
  7     <b>
  8            text2
  9        </b>
 10  </a>

  \[
  \begin{array}{c|c}
  \text{<a>} & \text{<b>} \\
  (1,1:10,0) & (1,3:5,2) \\
  (1,2:6,1) & (1,7:9,1) \\
  \end{array}
  \]

  will miss \text{<b>text1</b>}

- This can be easily fixed by using a stack that holds the 'open' elements of the left input

  - when we advance from (1,1:10,0) to (1,2:6,1) we push (1,1:10,0)
  - very little space overhead: max size of stack = depth of the XML tree
Preorder/Postorder Encoding

- Each node is assigned a (pre,post) pair
  - replaces (begin,end)
  - Preorder is the document order of the opening tags
  - Postorder is the document order of the closing tags

We can now check for all XPath axes (steps) using pre, post, & level