Class Agenda

• Last time:
  – Views and Relational Encodings of XML (1)

• Today:
  – Views and Relational Encodings of XML (2)
  – Cookbook Session: Query Evaluation Plans

• Reading:
  – none
Views and Relational Encodings of XML (2)
The Simplest Way to Encode a Tree: Tree Tables

• Suppose we had:

•

  <tree id="0">
  <content id="1">
  <sub-content>XYZ</sub-content>
  <i-content>14</i-content>
  </content>
  </tree>

• Where we have no IDs, invent values

What are the shortcomings of this approach?
Florescu/Kossmann Improved Tree Approach

- Consider order, typing; separate the values

### Tree

<table>
<thead>
<tr>
<th>parent</th>
<th>ord</th>
<th>label</th>
<th>flag</th>
<th>target</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>1</td>
<td>tree</td>
<td>ref</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>content</td>
<td>ref</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>sub-content</td>
<td>str</td>
<td>v2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>i-content</td>
<td>int</td>
<td>v3</td>
</tr>
</tbody>
</table>

### Vint

<table>
<thead>
<tr>
<th>vid</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>v3</td>
<td>14</td>
</tr>
</tbody>
</table>

### Vstring

<table>
<thead>
<tr>
<th>vid</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>v2</td>
<td>XYZ</td>
</tr>
</tbody>
</table>
A Relation that Mirrors the XML Hierarchy

Output relation, encoding this tree, would look like:

```
<tree id="0">
  <content id="1">
    <sub-content>XYZ</sub-content>
    <i-content>14</i-content>
  </content>
</tree>
```

<table>
<thead>
<tr>
<th>rLabel</th>
<th>rid</th>
<th>rOrd</th>
<th>clabel</th>
<th>cid</th>
<th>cOrd</th>
<th>sLabel</th>
<th>sid</th>
<th>sOrd</th>
<th>str</th>
<th>int</th>
</tr>
</thead>
<tbody>
<tr>
<td>tree</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>1</td>
<td>content</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>sub-content</td>
<td>2</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>2</td>
<td>1</td>
<td>XYZ</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>1</td>
<td>2</td>
<td>i-content</td>
<td>3</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>1</td>
<td>2</td>
<td>-</td>
<td>3</td>
<td>1</td>
<td>-</td>
<td>14</td>
</tr>
</tbody>
</table>
“Inlining” Techniques

• Folks at Wisconsin noted we can exploit the “structured” aspects of semi-structured XML
  – If we’re given a DTD, often the DTD has a lot of required (and often singleton) child elements
    • Book(title, author*, publisher)
  – Recall how normalization works in a traditional DBMS:
    • Decompose until we have everything in a relation determined by the keys
    • ... But don’t decompose any further than that
  – Shanmugasundaram et al. try not to decompose XML beyond the point of singleton children
Once we've encoded the XML in relational form, would like to query it! (Using XQuery, of course)

For limited fragments of XQuery, this is possible, via translation to SQL

Details of translation depend heavily on the encoding scheme

We'll look at 2 examples: one for XPERANTO-style encoding, and one for an inlined encoding
Running Example for XQuery -> SQL Translation

**XQuery:**
for $X in document("mydoc")/tree/content
where $X/sub-content = "XYZ"
return $X

**Source document:**
<tree id="0">
  <content id="1">
    <sub-content>XYZ</sub-content>
    <i-content>14</i-content>
  </content>
</tree>

**Query output:**
<content id="1">
  <sub-content>XYZ</sub-content>
  <i-content>14</i-content>
</content>
## EXPERANTO-Style Encoding: Input

**Source document:**

```xml
<tree id="0">
    <content id="1">
        <sub-content>XYZ</sub-content>
        <i-content>14</i-content>
    </content>
</tree>
```

<table>
<thead>
<tr>
<th>rLabel</th>
<th>rid</th>
<th>rOrd</th>
<th>clabel</th>
<th>cid</th>
<th>cOrd</th>
<th>sLabel</th>
<th>sid</th>
<th>sOrd</th>
<th>str</th>
<th>int</th>
</tr>
</thead>
<tbody>
<tr>
<td>tree</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>1</td>
<td>content</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>sub-content</td>
<td>2</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>2</td>
<td>1</td>
<td>XYZ</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>1</td>
<td>2</td>
<td>i-content</td>
<td>3</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>1</td>
<td>2</td>
<td>-</td>
<td>3</td>
<td>1</td>
<td>-</td>
<td>14</td>
</tr>
</tbody>
</table>
EXPERANTO-Style Encoding: Output

**Query output:**
<content id="1">
  <sub-content>XYZ</sub-content>
  <i-content>14</i-content>
</content>

<table>
<thead>
<tr>
<th>rlabel</th>
<th>rid</th>
<th>rOrd</th>
<th>cLabel</th>
<th>cid</th>
<th>cOrd</th>
<th>str</th>
<th>int</th>
</tr>
</thead>
<tbody>
<tr>
<td>content</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>1</td>
<td>sub-content</td>
<td>2</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>2</td>
<td>1</td>
<td>XYZ</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>1</td>
<td>i-content</td>
<td>3</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>3</td>
<td>1</td>
<td>-</td>
<td>14</td>
</tr>
</tbody>
</table>
EXPERANTO-Style Encoding: Query

**XQuery:**
for $X$ in document(“mydoc”)/tree/content
where $X$/sub-content = “XYZ”
return $X$

**SQL:**
select T5.clabel as rlabel, T5.cid as rid, T5.cOrd as rOrd,
    T5.sLabel as cLabel, T5.sid as cid, T5.sOrd as cOrd,
    T5.str as str, T5.int as int
from Tree T1, Tree T2, Tree T3, Tree T4, Tree T5
where T1.rLabel = "tree" and
    T2.rid = T1.rid and T2.rOrd = T1.rOrd and T2.clabel = "content" and
    T3.rid = T2.rid and T3.rOrd = T2.rOrd and T3.cid = T2.cid and
    T3.cOrd = T2.cOrd and T3.label = "sub-content" and
    T4.rid = T3.rid and ... T4.sid = T3.sid and T4.str = "XYZ" and
    T5.rid = T2.rid and ... and T5.cOrd = T2.cOrd
EXPERANTO-Style Encoding: Query (2)

```sql
select T5.clabel as rlabel, ..., T5.int as int
from Tree T1, Tree T2, Tree T3, Tree T4, Tree T5
where T1.rLabel = "tree" and T2.rid = T1.rid and ... and T2.clabel = "content" and T3.rid = T2.rid and ... and T3.label = "sub-content" and T4.rid = T3.rid and ... and T4.str = "XYZ" and T5.rid = T1.rid and ... and T5.cid = T2.cid
```
Inlined Encoding: Input Tables

**Source document:**

```xml
<tree id="0">
  <content id="1">
    <sub-content>XYZ</sub-content>
    <i-content>14</i-content>
  </content>
</tree>
```

- Suppose we know from DTD or XML-Schema that:
  - root node is "tree";
  - "content" node only occurs under "tree";
  - "content" node contains ID attribute "id" and exactly two children: "sub-content", with text data, and "i-content", with integer-valued data

- Can encode the tree as follows:

<table>
<thead>
<tr>
<th>Tree</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>parentID</th>
<th>id</th>
<th>ord</th>
<th>sub-content</th>
<th>i-content</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>XYZ</td>
<td>14</td>
</tr>
</tbody>
</table>
Inlined Encoding: Query

**XQuery:**
for $X$ in document("mydoc")/tree/content
where $X/sub-content = "XYZ"
return $X

**SQL:**
select Content.*
from Tree, Content
where Tree.id = Content.parentID and
  Content.sub-content = "XYZ"

Output is another inline-encoded relation:

<table>
<thead>
<tr>
<th>parentID</th>
<th>id</th>
<th>ord</th>
<th>sub-content</th>
<th>i-content</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>XYZ</td>
<td>14</td>
</tr>
</tbody>
</table>

*2-way join, instead of a 5-way join!*
Summary: Relational "Views" of XML

• We’ve seen that views are useful things

• Allow us to store and refer to the results of a query

• We’ve seen examples of relational "views" of XML, and SQL translations of XQuery over such views
  
  – Current versions of the major DBMSs support XML and (fragments of) XQuery this way
  
  – Challenge: limiting the number of joins required in the translated queries
And Now For Something Completely Different

Query Engine Sneak Preview and Cookbook Session
DavisDB, Part 4: Query Engine

• Culmination of the project: you'll implement a query engine for a fragment of SQL
  – Queries: `select-from-where`
  – Updates: `insert into`, `delete from`, `update`

• As in Part 3, `SystemParser` handles the front-end, you implement the back-end: a class called `QueryEngine`

• Relatively low bar for getting full credit
  – Optimization not required
  – Starter code and architectural template provided

• Opportunities for extra credit
  – e.g., Query optimizer
Query Engine API

• (cf. the Doxygen docs...)
How to Implement an Execution Engine?

Physical query plan

C++ implementation
How to Implement an Execution Engine?

Physical query plan

\[
\pi_{\text{sname}} \\
\sigma_{\text{bid}=100 \ \text{AND} \ \text{rating}>5} \\
&_{\text{sid} = \text{sid}} \\
\text{Reserves} \ \text{and} \ \text{Sailors}
\]

C++ implementation

\[
\pi_{\text{sname}} \\
\sigma_{\text{bid}=100 \ \text{AND} \ \text{rating}>5} \\
&_{\text{sid} = \text{sid}} \\
\text{Reserves} \ \text{and} \ \text{Sailors}
\]
IQueryOperator: an Abstract Interface

• What is an "abstract interface" in C++?

• A base class with only **abstract virtual** methods

  ```cpp
  virtual ReturnCode getNextRecord(char* data) = 0;
  ```

• Other classes inherit from this base class ("implement" the interface) and fill in the method implementations

• One technical exception: virtual destructor must have implementation, but can be empty

  ```cpp
  virtual ~IQueryOperator() {};
  ```
What's a Virtual Method?

- C++ versus Java: in Java, **all** methods are virtual!

```cpp
class A {
    void foo() { printf("A foo"); }
    virtual void bar() { printf("A bar"); }
}

class B : A {
    void foo() { printf("B foo"); }
    virtual void bar() { printf("B bar"); }
}

void biz(A* a, B* b) {
    A* a, B* b; A* c = (A*)b;
    a->foo();
    a->bar();
    b->foo();
    b->bar();
    c->foo();
    c->bar();
}
```

**OUTPUT of call to biz(a,b):**

- **A foo**
- **A bar**
- **B foo**
- **B bar**

**QUESTION:** why declare destructors virtual?
Why Use Interfaces in the Execution Engine?

• An operator shouldn't have to know about all the different physical operators that might be below it in the tree!
IQueryOperator

• (cf. code in Xcode...)

"Canonical" Execution Plans

• Given a select-from-where query, will only be required to build a "canonical" execution plan
  
  – A plan fully determined by the order of relations in the "from" clause and availability of indices

• Extra credit: build an optimizer to explore other plans
  
  – Based on heuristics or statistics
Putting it All Together

• Let's go to the videotape!