



Lecture 24 of 42

XML Structure and Document Schemas Discussion: Indexing

Friday, 20 October 2006

William H. Hsu
Department of Computing and Information Sciences, KSU

KSOL course page: <http://snipurl.com/va60>

Course web site: <http://www.kddresearch.org/Courses/Fall-2006/CIS560>

Instructor home page: <http://www.cis.ksu.edu/~bhsu>

Reading for Next Class:

Second half of Chapter 10, Silberschatz *et al.*, 5th edition



XML

- Structure of XML Data
- XML Document Schema
- Querying and Transformation
- Application Program Interfaces to XML
- Storage of XML Data
- XML Applications





Introduction

- XML: Extensible Markup Language
- Defined by the WWW Consortium (W3C)
- Derived from SGML (Standard Generalized Markup Language), but simpler to use than SGML
- Documents have tags giving extra information about sections of the document
 - * E.g. `<title> XML </title> <slide> Introduction ...</slide>`
- **Extensible**, unlike HTML
 - * Users can add new tags, and *separately* specify how the tag should be handled for display



Creating XML Output

- Any text or tag in the XSL stylesheet that is not in the xsl namespace is output as is
- E.g. to wrap results in new XML elements.

```
<xsl:template match="/bank-2/customer">
  <customer>
    <xsl:value-of select="customer_name"/>
  </customer>
</xsl:template>
<xsl:template match="*" />
```

 - * Example output:

```
<customer> Joe </customer>
<customer> Mary </customer>
```





Creating XML Output (Cont.)

- Note: Cannot directly insert a `xsl:value-of` tag inside another tag
 - * E.g. cannot create an attribute for `<customer>` in the previous example by directly using `xsl:value-of`
 - * XSLT provides a construct `xsl:attribute` to handle this situation
 - ⇒ `xsl:attribute` adds attribute to the preceding element
 - ⇒ E.g.

```
<customer>
  <xsl:attribute name="customer_id">
    <xsl:value-of select="customer_id"/>
  </xsl:attribute>
</customer>
```

 results in output of the form

```
<customer customer_id="..."> ....
```
- `xsl:element` is used to create output elements with computed names



Structural Recursion

- Template action can apply templates recursively to the contents of a matched element


```
<xsl:template match="/bank">
  <customers>
    <xsl:template apply-templates/>
  </customers >
</xsl:template>
<xsl:template match="/customer">
  <customer>
    <xsl:value-of select="customer_name"/>
  </customer>
</xsl:template>
<xsl:template match="*" />
```
- Example output:


```
<customers>
  <customer> John </customer>
  <customer> Mary </customer>
</customers>
```





Joins in XSLT

- XSLT keys allow elements to be looked up (indexed) by values of subelements or attributes
 - Keys must be declared (with a name) and, the key() function can then be used for lookup. E.g.


```
<xsl:key name="acctno" match="account"
            use="account_number"/>
<xsl:value-of select=key("acctno", "A-101")
```
- Keys permit (some) joins to be expressed in XSLT


```
<xsl:key name="acctno" match="account" use="account_number"/>
<xsl:key name="custno" match="customer" use="customer_name"/>
<xsl:template match="depositor">
  <cust_acct>
    <xsl:value-of select=key("custno", "customer_name")/>
    <xsl:value-of select=key("acctno", "account_number")/>
  </cust_acct>
</xsl:template>
<xsl:template match="*" />
```



Sorting in XSLT

- Using an `xsl:sort` directive inside a template causes all elements matching the template to be sorted
 - * Sorting is done before applying other templates

```
<xsl:template match="/bank">
  <xsl:apply-templates select="customer">
    <xsl:sort select="customer_name"/>
  </xsl:apply-templates>
</xsl:template>
<xsl:template match="customer">
  <customer>
    <xsl:value-of select="customer_name"/>
    <xsl:value-of select="customer_street"/>
    <xsl:value-of select="customer_city"/>
  </customer>
</xsl:template>
<xsl:template match="*" />
```





Application Program Interface

- There are two standard application program interfaces to XML data:
 - * **SAX** (Simple API for XML)
 - ⇒ Based on parser model, user provides event handlers for parsing events
 - ◆ E.g. start of element, end of element
 - ◆ Not suitable for database applications
 - * **DOM** (Document Object Model)
 - ⇒ **XML** data is parsed into a tree representation
 - ⇒ Variety of functions provided for traversing the DOM tree
 - ⇒ E.g.: Java DOM API provides Node class with methods
 - getParentNode(), getFirstChild(), getNextSibling()
 - getAttribute(), getData() (for text node)
 - getElementsByTagName(), ...
 - ⇒ Also provides functions for updating DOM tree



Storage of XML Data

- XML data can be stored in
 - * **Non-relational data stores**
 - ⇒ Flat files
 - ◆ Natural for storing XML
 - ◆ But has all problems discussed in Chapter 1 (no concurrency, no recovery, ...)
 - ⇒ XML database
 - ◆ Database built specifically for storing XML data, supporting DOM model and declarative querying
 - ◆ Currently no commercial-grade systems
 - * **Relational databases**
 - ⇒ Data must be translated into relational form
 - ⇒ Advantage: mature database systems
 - ⇒ Disadvantages: overhead of translating data and queries





Storage of XML in Relational Databases

- Alternatives:
 - * String Representation
 - * Tree Representation
 - * Map to relations



String Representation

- Store each top level element as a string field of a tuple in a relational database
 - * Use a single relation to store all elements, or
 - * Use a separate relation for each top-level element type
 - ⇒ E.g. account, customer, depositor relations
 - ◆ Each with a string-valued attribute to store the element
- Indexing:
 - * Store values of subelements/attributes to be indexed as extra fields of the relation, and build indices on these fields
 - ⇒ E.g. customer_name or account_number
 - * Some database systems support **function indices**, which use the result of a function as the key value.
 - ⇒ The function should return the value of the required subelement/attribute





String Representation (Cont.)

- Benefits:
 - * Can store any XML data even without DTD
 - * As long as there are many top-level elements in a document, strings are small compared to full document
 - ⇒ Allows fast access to individual elements.
- Drawback: Need to parse strings to access values inside the elements
 - * Parsing is slow.

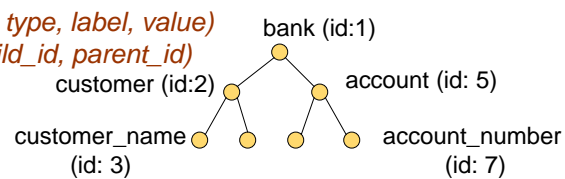


Tree Representation

- **Tree representation:** model XML data as tree and store using relations

nodes(*id*, *type*, *label*, *value*)

child (*child_id*, *parent_id*)



- Each element/attribute is given a unique identifier
- Type indicates element/attribute
- Label specifies the tag name of the element/name of attribute
- Value is the text value of the element/attribute
- The relation *child* notes the parent-child relationships in the tree
 - * Can add an extra attribute to *child* to record ordering of children





Tree Representation (Cont.)

- Benefit: Can store any XML data, even without DTD
- Drawbacks:
 - * Data is broken up into too many pieces, increasing space overheads
 - * Even simple queries require a large number of joins, which can be slow



Mapping XML Data to Relations

- Relation created for each element type whose schema is known:
 - * An id attribute to store a unique id for each element
 - * A relation attribute corresponding to each element attribute
 - * A parent_id attribute to keep track of parent element
 - ⇒ As in the tree representation
 - ⇒ Position information (ith child) can be store too
- All subelements that occur only once can become relation attributes
 - * For text-valued subelements, store the text as attribute value
 - * For complex subelements, can store the id of the subelement
- Subelements that can occur multiple times represented in a separate table
 - * Similar to handling of multivalued attributes when converting ER diagrams to tables





Storing XML Data in Relational Systems

- *Publishing*: process of converting relational data to an XML format
- *Shredding*: process of converting an XML document into a set of tuples to be inserted into one or more relations
- XML-enabled database systems support automated publishing and shredding
- Some systems offer *native storage* of XML data using the **xml** data type. Special internal data structures and indices are used for efficiency



SQL/XML

- New standard SQL extension that allows creation of nested XML output
 - * Each output tuple is mapped to an XML element *row*
- ```
<bank>
 <account>
 <row>
 <account_number> A-101 </account_number>
 <branch_name> Downtown </branch_name>
 <balance> 500 </balance>
 </row>
 more rows if there are more output tuples ...
 </account>
</bank>
```





## SQL Extensions

- **xmlelement** creates XML elements
- **xmlattributes** creates attributes

```
select xmlelement (name "account",
 xmlattributes (account_number as account_number),
 xmlelement (name "branch_name", branch_name),
 xmlelement (name "balance", balance))
from account
```



## Web Services

- The Simple Object Access Protocol (SOAP) standard:
  - \* Invocation of procedures across applications with distinct databases
  - \* XML used to represent procedure input and output
- A *Web service* is a site providing a collection of SOAP procedures
  - \* Described using the Web Services Description Language (WSDL)
  - \* Directories of Web services are described using the Universal Description, Discovery, and Integration (UDDI) standard

