



## Lecture 23 of 42

# Extensible Markup Language (XML) Discussion: Semistructured Data

Wednesday, 18 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:

First half of Chapter 10, Silberschatz *et al.*, 5<sup>th</sup> 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



## XML Introduction (Cont.)

- The ability to specify new tags, and to create nested tag structures make XML a great way to exchange **data**, not just documents.
  - \* Much of the use of XML has been in data exchange applications, not as a replacement for HTML
- Tags make data (relatively) self-documenting
  - \* E.g.

```
<bank>
  <account>
    <account_number> A-101 </account_number>
    <branch_name> Downtown </branch_name>
    <balance> 500 </balance>
  </account>
  <depositor>
    <account_number> A-101 </account_number>
    <customer_name> Johnson </customer_name>
  </depositor>
</bank>
```





## Document Type Definition (DTD)

- The type of an XML document can be specified using a DTD
- DTD constrains structure of XML data
  - \* What elements can occur
  - \* What attributes can/must an element have
  - \* What subelements can/must occur inside each element, and how many times.
- DTD does not constrain data types
  - \* All values represented as strings in XML
- DTD syntax
  - \* `<!ELEMENT element (subelements-specification) >`
  - \* `<!ATTLIST element (attributes) >`



## Element Specification in DTD

- Subelements can be specified as
  - \* names of elements, or
  - \* #PCDATA (parsed character data), i.e., character strings
  - \* EMPTY (no subelements) or ANY (anything can be a subelement)
- Example
  - `<! ELEMENT depositor (customer_name account_number)>`
  - `<! ELEMENT customer_name (#PCDATA)>`
  - `<! ELEMENT account_number (#PCDATA)>`
- Subelement specification may have regular expressions
  - `<!ELEMENT bank ( ( account | customer | depositor)+>`
  - ⇒ Notation:
    - ◆ "[]" - alternatives
    - ◆ "+" - 1 or more occurrences
    - ◆ "\*" - 0 or more occurrences

get  
regular  
expression  
A.G.R.T





## Bank DTD

```
<!DOCTYPE bank [  
  <!ELEMENT bank ( ( account | customer | depositor)+)>  
  <!ELEMENT account (account_number branch_name balance)>  
  <!ELEMENT customer(customer_name customer_street  
                      customer_city)>  
  <!ELEMENT depositor (customer_name account_number)>  
  <!ELEMENT account_number (#PCDATA)>  
  <!ELEMENT branch_name (#PCDATA)>  
  <!ELEMENT balance(#PCDATA)>  
  <!ELEMENT customer_name(#PCDATA)>  
  <!ELEMENT customer_street(#PCDATA)>  
  <!ELEMENT customer_city(#PCDATA)>  
>
```



## Attribute Specification in DTD

- Attribute specification : for each attribute
  - \* Name
  - \* Type of attribute
    - ⇒ CDATA
    - ⇒ ID (identifier) or IDREF (ID reference) or IDREFS (multiple IDREFs)
      - ◆ more on this later
  - \* Whether
    - ⇒ mandatory (#REQUIRED)
    - ⇒ has a default value (value),
    - ⇒ or neither (#IMPLIED)
- Examples
  - \* <!ATTLIST account acct-type CDATA "checking">
  - \* <!ATTLIST customer  
customer\_id ID # REQUIRED  
accounts IDREFS # REQUIRED >



## IDs and IDREFs

- An element can have at most one attribute of type ID
- The ID attribute value of each element in an XML document must be distinct
  - \* Thus the ID attribute value is an object identifier
- An attribute of type IDREF must contain the ID value of an element in the same document
- An attribute of type IDREFS contains a set of (0 or more) ID values. Each ID value must contain the ID value of an element in the same document



## Bank DTD with Attributes

- Bank DTD with ID and IDREF attribute types.

```
<!DOCTYPE bank-2[
  <!ELEMENT account (branch, balance)>
  <!ATTLIST account
    account_number ID # REQUIRED
    owners IDREFS # REQUIRED>
  <!ELEMENT customer(customer_name, customer_street,
    customer_city)>
  <!ATTLIST customer
    customer_id ID # REQUIRED
    accounts IDREFS # REQUIRED>
  ... declarations for branch, balance, customer_name,
    customer_street and customer_city
]>
```





## XML data with ID and IDREF attributes

```

<bank-2>
  <account account_number="A-401" owners="C100 C102">
    <branch_name> Downtown </branch_name>
    <balance> 500 </balance>
  </account>
  <customer customer_id="C100" accounts="A-401">
    <customer_name>Joe </customer_name>
    <customer_street> Monroe </customer_street>
    <customer_city> Madison</customer_city>
  </customer>
  <customer customer_id="C102" accounts="A-401 A-402">
    <customer_name> Mary </customer_name>
    <customer_street> Erin </customer_street>
    <customer_city> Newark </customer_city>
  </customer>
</bank-2>
  
```



## Limitations of DTDs

- No typing of text elements and attributes
  - \* All values are strings, no integers, reals, etc.
- Difficult to specify unordered sets of subelements
  - \* Order is usually irrelevant in databases (unlike in the document-layout environment from which XML evolved)
  - \* (A | B)\* allows specification of an unordered set, but
    - ⇒ Cannot ensure that each of A and B occurs only once
- IDs and IDREFs are untyped
  - \* The *owners* attribute of an account may contain a reference to another account, which is meaningless
    - ⇒ *owners* attribute should ideally be constrained to refer to customer elements



## XML Schema

Account of string \* int  
Amount acct ID

- XML Schema is a more sophisticated schema language which addresses the drawbacks of DTDs. Supports
  - \* Typing of values
    - ⇒ E.g. integer, string, etc
    - ⇒ Also, constraints on min/max values
  - \* User-defined, complex types
  - \* Many more features, including
    - ⇒ uniqueness and foreign key constraints, inheritance
- XML Schema is itself specified in XML syntax, unlike DTDs
  - \* More-standard representation, but verbose
- XML Schema is integrated with namespaces
- BUT: XML Schema is significantly more complicated than DTDs.



## XML Schema Version of Bank DTD

```
<xs:schema xmlns:xs=http://www.w3.org/2001/XMLSchema>
<xs:element name="bank" type="BankType"/>
<xs:element name="account">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="account_number" type="xs:string"/>
      <xs:element name="branch_name" type="xs:string"/>
      <xs:element name="balance" type="xs:decimal"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
.... definitions of customer and depositor ....
<xs:complexType name="BankType">
  <xs:sequence>
    <xs:element ref="account" minOccurs="0" maxOccurs="unbounded"/>
    <xs:element ref="customer" minOccurs="0" maxOccurs="unbounded"/>
    <xs:element ref="depositor" minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
</xs:schema>
```



## XML Schema Version of Bank DTD

- Choice of “xs:” was ours -- any other namespace prefix could be chosen
- Element “bank” has type “BankType”, which is defined separately
  - \* `xs:complexType` is used later to create the named complex type “BankType”
- Element “account” has its type defined in-line



## More features of XML Schema

- Attributes specified by `xs:attribute` tag:
  - \* `<xs:attribute name = “account_number”/>`
  - \* adding the attribute use = “required” means value must be specified
- Key constraint: “account numbers form a key for account elements under the root bank element:

```
<xs:key name = “accountKey”>
  <xs:selector xpath = “]bank/account”/>
  <xs:field xpath = “account_number”/>
</xs:key>
```
- Foreign key constraint from depositor to account:

```
<xs:keyref name = “depositorAccountKey”
refer=“accountKey”>
  <xs:selector xpath = “]bank/account”/>
  <xs:field xpath = “account_number”/>
</xs:keyref>
```





## Querying and Transforming XML Data

- Translation of information from one XML schema to another
- Querying on XML data
- Above two are closely related, and handled by the same tools
- Standard XML querying/translation languages
  - \* XPath
    - ⇒ Simple language consisting of path expressions
  - \* XSLT
    - ⇒ Simple language designed for translation from XML to XML and XML to HTML
  - \* XQuery
    - ⇒ An XML query language with a rich set of features



## Tree Model of XML Data

- Query and transformation languages are based on a **tree model** of XML data
- An XML document is modeled as a tree, with **nodes** corresponding to elements and attributes
  - \* Element nodes have child nodes, which can be attributes or subelements
  - \* Text in an element is modeled as a text node child of the element
  - \* Children of a node are ordered according to their order in the XML document
  - \* Element and attribute nodes (except for the root node) have a single parent, which is an element node
  - \* The root node has a single child, which is the root element of the document





## XPath

- XPath is used to address (select) parts of documents using **path expressions**
- A path expression is a sequence of steps separated by “/”
  - \* Think of file names in a directory hierarchy
- Result of path expression: set of values that along with their containing elements/attributes match the specified path
- E.g. `/bank-2/customer/customer_name` evaluated on the bank-2 data we saw earlier returns

```
<customer_name>Joe</customer_name>
<customer_name>Mary</customer_name>
```
- E.g. `/bank-2/customer/customer_name/text( )` returns the same names, but without the enclosing tags



## XPath (Cont.)

- The initial “/” denotes root of the document (above the top-level tag)
- Path expressions are evaluated left to right
  - \* Each step operates on the set of instances produced by the previous step
- Selection predicates may follow any step in a path, in [ ]
  - \* E.g. `/bank-2/account[balance > 400]`
    - ⇒ returns account elements with a balance value greater than 400
    - ⇒ `/bank-2/account[balance]` returns account elements containing a balance subelement
- Attributes are accessed using “@”
  - \* E.g. `/bank-2/account[balance > 400]/@account_number`
    - ⇒ returns the account numbers of accounts with balance > 400
  - \* IDREF attributes are not dereferenced automatically (more on this later)





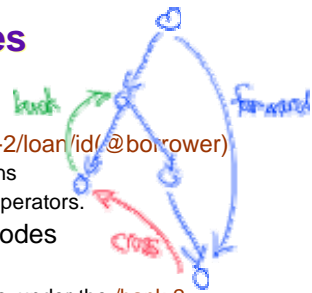
## Functions in XPath

- XPath provides several functions
  - \* The function `count()` at the end of a path counts the number of elements in the set generated by the path
    - ⇒ E.g. `/bank-2/account[count(./customer) > 2]`
      - ◆ Returns accounts with > 2 customers
    - \* Also function for testing position (1, 2, ..) of node w.r.t. siblings
  - Boolean connectives `and` and `or` and function `not()` can be used in predicates
  - IDREFs can be referenced using function `id()`
    - \* `id()` can also be applied to sets of references such as IDREFS and even to strings containing multiple references separated by blanks
    - \* E.g. `/bank-2/account/id(@owner)`
      - ⇒ returns all customers referred to from the owners attribute of account elements.



## More XPath Features

- Operator `"|"` used to implement union
  - \* E.g. `/bank-2/account/id(@owner) | /bank-2/loan/id(@borrower)`
    - ⇒ Gives customers with either accounts or loans
    - ⇒ However, `"|"` cannot be nested inside other operators.
- `"//"` can be used to skip multiple levels of nodes
  - \* E.g. `/bank-2//customer_name`
    - ⇒ finds any `customer_name` element *anywhere* under the `/bank-2` element, regardless of the element in which it is contained.
- A step in the path can go to parents, siblings, ancestors and descendants of the nodes generated by the previous step, not just to the children
  - \* `"//"`, described above, is a short form for specifying "all descendants"
  - \* `".."` specifies the parent.
- `doc(name)` returns the root of a named document





## XQuery

- XQuery is a general purpose query language for XML data
- Currently being standardized by the World Wide Web Consortium (W3C)
  - \* The textbook description is based on a January 2005 draft of the standard. The final version may differ, but major features likely to stay unchanged.
- XQuery is derived from the Quilt query language, which itself borrows from SQL, XQL and XML-QL
- XQuery uses a **for ... let ... where ... order by ... result ...** syntax
  - for** ⇔ SQL **from**
  - where** ⇔ SQL **where**
  - order by** ⇔ SQL **order by**
  - result** ⇔ SQL **select**
  - let** allows temporary variables, and has no equivalent in SQL



## FLWOR Syntax in XQuery

- For clause uses XPath expressions, and variable in for clause ranges over values in the set returned by XPath
- Simple FLWOR expression in XQuery
  - \* find all accounts with balance > 400, with each result enclosed in an <account\_number> .. </account\_number> tag
 

```

for   $x in /bank-2/account
let   $acctno := $x/@account_number
where $x/balance > 400
return <account_number> { $acctno } </account_number>
          
```
  - \* Items in the **return** clause are XML text unless enclosed in {}, in which case they are evaluated
- Let clause not really needed in this query, and selection can be done in XPath. Query can be written as:
 

```

for $x in /bank-2/account[balance>400]
return <account_number> { $x/@account_number }
          </account_number>
      
```



## Joins

- Joins are specified in a manner very similar to SQL
 

```

for $a in /bank/account,
    $c in /bank/customer,
    $d in /bank/depositor
where $a/account_number = $d/account_number
and $c/customer_name = $d/customer_name
return <cust_acct> { $c $a } </cust_acct>
      
```
- The same query can be expressed with the selections specified as XPath selections:
 

```

for $a in /bank/account
    $c in /bank/customer
    $d in /bank/depositor[
      account_number = $a/account_number and
      customer_name = $c/customer_name]
return <cust_acct> { $c $a } </cust_acct>
      
```



## Nested Queries

- The following query converts data from the flat structure for `bank` information into the nested structure used in `bank-1`

```

<bank-1> {
  for $c in /bank/customer
  return
    <customer>
      { $c/* }
      { for $d in /bank/depositor[customer_name = $c/customer_name],
        $a in /bank/account[account_number=$d/account_number]
        return $a }
    </customer>
} </bank-1>
      
```
- `$c/*` denotes all the children of the node to which `$c` is bound, without the enclosing top-level tag
- `$c/text()` gives text content of an element without any subelements / tags





## Sorting in XQuery

- The **order by** clause can be used at the end of any expression. E.g. to return customers sorted by name

```
for $c in /bank/customer
order by $c/customer_name
return <customer> { $c/* } </customer>
```

- Use **order by** \$c/customer\_name to sort in descending order

- Can sort at multiple levels of nesting (sort by customer\_name, and by account\_number within each customer)

```
<bank-1> {
  for $c in /bank/customer
  order by $c/customer_name
  return
    <customer>
      { $c/* }
      { for $d in
        /bank/depositor[customer_name=$c/customer_name],
          $a in
            /bank/account[account_number=$d/account_number] }
        order by $a/account_number
      return <accounts> $a/* </accounts>
    }
}</bank-1>
```



## Functions and Other XQuery Features

- User defined functions with the type system of XMLSchema

```
function balances(xs:string $c) returns list(xs:decimal*) {
  for $d in /bank/depositor[customer_name = $c],
    $a in /bank/account[account_number = $d/account_number]
  return $a/balance
}
```

- Types are optional for function parameters and return values
- The \* (as in decimal\*) indicates a sequence of values of that type
- Universal and existential quantification in where clause predicates
  - \* **some** \$e in path satisfies P
  - \* **every** \$e in path satisfies P
- XQuery also supports If-then-else clauses



## XSLT

- A **stylesheet** stores formatting options for a document, usually separately from document
  - \* E.g. an HTML style sheet may specify font colors and sizes for headings, etc.
- The **XML Stylesheet Language (XSL)** was originally designed for generating HTML from XML
- XSLT is a general-purpose transformation language
  - \* Can translate XML to XML, and XML to HTML
- XSLT transformations are expressed using rules called **templates**
  - \* Templates combine selection using XPath with construction of results



## XSLT Templates

- Example of XSLT template with **match** and **select** part

```
<xsl:template match="/bank-2/customer">
  <xsl:value-of select="customer_name"/>
</xsl:template>
<xsl:template match="*">
```
- The match attribute of xsl:template specifies a pattern in XPath
- Elements in the XML document matching the pattern are processed by the actions within the **xsl:template** element
  - \* **xsl:value-of** selects (outputs) specified values (here, **customer\_name**)
- For elements that do not match any template
  - \* Attributes and text contents are output as is
  - \* Templates are recursively applied on subelements
- The **<xsl:template match="\*">** template matches all elements that do not match any other template
  - \* Used to ensure that their contents do not get output.
- If an element matches several templates, only one is used based on a complex priority scheme/user-defined priorities





## 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>
```

