Introduction to XML

Prof. Weining Zhang

What is XML?
- XML stands for Exensible Markup Language
- XML is a markup language much like HTML.
- XML was designed to describe data
- XML tags are not predefined. You must define your own tags
- XML uses a Document Type Definition (DTD) or an XML Schema to describe the data
- XML with a DTD or XML Schema is designed to be self-descriptive
- XML is a W3C Recommendation

XML and HTML
- XML was designed to describe data and to focus on what data is.
- HTML was designed to display data and to focus on how data looks.
- Both XML and HTML use tags to mark up parts of text documents, to describe aspects of documents.
- Specially designed software is needed to process XML and HTML documents.

Why XML
- XML was created so that richly structured documents could be used over the web. The only viable alternatives, HTML and SGML, are not practical for this purpose.
  - HTML comes bound with a set of semantics and does not provide arbitrary structure.
  - SGML provides arbitrary structure, but is too difficult to implement just for a web browser.
- XML does not replace HTML or SGML.

Example: XML Data
```xml
<?xml version="1.0"?>
<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>
```
**Elements**

- All XML elements must have a closing tag
- XML tags are case sensitive
- XML Elements must be properly nested
  - Proper nesting
    - `<account> … <balance> … </balance> </account>`
  - Improper nesting
    - `<account> … <balance> … </account> </balance>`
- Every document must have a root element

**Example of Nested Elements**

```xml
<?xml version="1.0"?>
<bank>
  <customer>
    <customer-name> Hayes </customer-name>
    <customer-street> Main </customer-street>
    <customer-city> Harrison </customer-city>
    <account>
      <account-number> A-102 </account-number>
      <branch-name> Perryridge </branch-name>
      <balance> 400 </balance>
    </account>
    <account> … </account>
  </customer>
</bank>
```

**Attributes**

- Elements can have attributes
  - `<account acct-type="checking">`  
    - `<account-number> A-102 </account-number>`
    - `<branch-name> Perryridge </branch-name>`
    - `<balance> 400 </balance>`
  - `<account>`
- Attributes are specified by `name=value` pairs inside the starting tag of an element
- Values must be quoted
- An element may have several attributes, but each attribute name must be unique within the element
  - `<account acct-type = "checking" monthly-fee="5">`

**Attributes vs Elements**

- There are no fixed rules about when to use attributes to describe data, and when to use elements
  - Same information can be represented in two ways
    - `<account account-number = “A-101”> … </account>`
    - `<account-account-number> A-101 </account-account-number> </account>`
  - Suggestion: use attributes for identifiers of elements, and use nested elements for other contents

**More on XML Syntax**

- Elements without nested elements or text content can be abbreviated by ending the start tag with a `/>` and deleting the end tag
  - `<account number="A-101" branch="Perryridge" balance="200 />`
- To store string data that may contain tags, without the tags being interpreted as subelements, use CDATA as below
  - `<![CDATA[<account> … </account>]]>`
  - Here, `<account>` and `/account>` are treated as just strings

**Namespaces**

- Resolve problems caused by tag names that are defined by multiple organizations with different meanings
  - `<? xml version="1.0"?>
    <bank xmlns:FB='http://www.FirstBank.com'>
    …
    <FB:branch>
    <FB:branchname>Downtown</FB:branchname>
    <FB:branchname> Brooklyn</FB:branchname>
    </FB:branch>
    …
    </bank>`
XML Validation

- A "Well Formed" XML document is a document that conforms to the XML syntax rules
- A "Valid" XML document is a "Well Formed" XML document which conforms to the rules of a Document Type Definition (DTD).
- Standard tools have been provided to check the validity of XML documents

XML Document Schema

- Database schemas constrain what information can be stored, and the data types of stored values
- XML documents are not required to have an associated schema
- However, schemas are very important for XML data exchange
  - Otherwise, a site cannot automatically interpret data received from another site
- Two mechanisms for specifying XML schema
  - Document Type Definition (DTD). Widely used
  - XML Schema. Newer, not yet widely used

Document Type Definition (DTD)

- Used to specify the type of an XML document
- 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 are represented as strings in XML
- DTD syntax
  - `<!ELEMENT element (subelements-specification)>`
  - `<!ATTLIST element (attributes)>`

Specify Elements

```xml
<?xml version="1.0"?>
<!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)> ]>
... data goes here ...
```

Specify Attributes

```xml
<?xml version="1.0"?>
<!ELEMENT bank []
<!ATTLIST account
 acct-num ID #REQUIRED>
<!ATTLIST account
 owners IDREFS #REQUIRED>
<!ATTLIST customer
 cust-id ID #REQUIRED>
<!ATTLIST customer
 accounts IDREFS #REQUIRED>
... declarations for branch, balance, customer-name, customer-street and customer-city ...
]>
... data goes here ...
```
**Specify Attributes**

- For each attribute, specify
  - **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)

**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

---

**XML data with ID and IDREF attributes**

```xml
<bank-2>
  <account acct-num="A-401" owners="C100 C102">
    <balance>500</balance>
    <branch-name>Downtown</branch-name>
  </account>
  <customer cust-id="C100" accounts="A-401">
    <name>Joe</name>
    <street>Monroe</street>
    <city>Madison</city>
  </customer>
  <customer cust-id="C102" accounts="A-401 A-402">
    <name>Mary</name>
    <street>Erin</street>
    <city>Newark</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
  - (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**

- A more sophisticated schema language
  - Typing of values
    - E.g., integer, string, etc.
  - Also, constraints on min/max values
  - User defined types
  - Is itself specified in XML syntax, unlike DTDs
  - More standard representation, but verbose
  - Is integrated with namespaces
  - Many more features
    - List types, uniqueness and foreign key constraints, inheritance . .

---

**XML Schema of the Bank**

```
<xsd:schema xmlns:xsd=http://www.w3.org/2001/XMLSchema>
  <xsd:element name="bank" type="BankType"/>
  <xsd:element name="account">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="account-number" type="xsd:string"/>
        <xsd:element name="branch-name" type="xsd:string"/>
        <xsd:element name="balance" type="xsd:decimal"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```
XML Schema of the Bank

```xml
<xsd:schema>
  <xsd:complexType name="BankType">
    <xsd:sequence>
      <xsd:element ref="account" minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element ref="customer" minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element ref="depositor" minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

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 children 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 have a single parent, which is an element node

Tree Model of XML Data

```
   Bank-2
     |      account
     |        acct-num
     |          branch-name
     |            balance
     |              owners
     |                "Downtown"
     |                   A-401
     |                      C100
     |                           C102
     |          cust-id
     |            accounts
     |              city
     |                name
     |            street
     |              A-401
     |                "Joe"
     |                "Madison"
     |              "Monroe"
```

We use the terminology of nodes, children, parent, siblings, ancestor, descendant, etc., which should be interpreted in the above tree model of XML data.

XML APIs

- Standard tools that applications can use to process XML data
  - Simple API for XML (SAX)
    - A standard API for parsing XML documents.
    - Event based (supported by J2SDK since v1.4)
    - Application programs define handler objects to respond to events, such as
      - Begin and end of document
      - Begin and end of elements
      - Character text
      - Parser is instantiated as a XML input stream

XML APIs

- Document Object Model (DOM)
  An object-oriented API of XML documents.
  - Objects: Node, Element, Attribute, etc. are modeled as objects.
  - XML data is parsed into a tree representation
  - Methods: Each object has methods to access & modify its components. Classes have methods to create instances.
  - Many functions are provided for traversing the DOM tree
XML APIs

- Java DOM API provides Node class with methods:
  - getParentNode()
  - getFirstChild()
  - getNextSibling()
  - getAttribute()
  - getData() (for text node)
  - getElementsByTagName()
- Also provides methods for updating DOM trees, and for generating XML documents from DOM trees.

Querying and Transforming XML

- Common tasks of XML applications include:
  - Query the XML data
  - Translate XML data from one schema to another
- Standard XML query/translation languages:
  - XPath
    - Simple language consisting of path expressions
  - XQuery
    - Standard language with a rich set of features
  - XSLT
    - Simple language to translate XML to XML and XML to HTML

XPath

- XPath uses path expressions to select parts of documents.
- A path expression is a sequence of steps separated by “/”.
  - /A/B/C
- Result of path expression: a set of subtrees whose roots have a path matching the path expression.

XPath (Cont.)

- E.g. /bank-2/customer/name evaluated on the bank-2 data we saw earlier returns:
  - <name>Joe</name>
  - <name>Mary</name>
- E.g. /bank-2/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

XPath (Cont.)

- Attributes are accessed using “@”.
  - E.g. /bank-2/account[@balance > 400]/@account-number
    - returns the account numbers of those accounts with balance > 400
  - IDREF attributes are not dereferenced automatically (more on this later).
Functions in XPath

- 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[customer/count() > 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.

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.
- “//" is used to skip multiple levels of nodes.
  - E.g. /bank-2/name
  - finds any name element anywhere under the /bank-2 element, regardless of the element in which it is contained.

XQuery

- XQuery is a general purpose query language for XML data.
  - A proposed World Wide Web Consortium (W3C) standard.
  - The current description is based on a March 2001 draft of the standard. The final version may differ, but major features likely to stay unchanged.
- Alpha version of XQuery engine available free from Microsoft.
- XQuery is derived from the Quilt query language, which itself borrows from SQL, XQL and XML-QL.

XQuery Syntax

- XQuery uses a syntax:
  - for ... let ... where .. result ...
- for SQL from
- where SQL where
- result SQL select
- let allows temporary variables, and has no equivalent in SQL.
- For clause uses XPath expressions, and variable in for clause ranges over values in the set returned by XPath.
An XQuery Example

- 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>
```

An XQuery Example (cont.)

- The let-clause is not really needed in this query, and selection can be done using XPath.

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

- Path expressions can be used in other clauses
  - E.g. used in let clause, to bind variables to results of path expressions

Functions

- The function distinct() can be used to removed duplicates in path expression results
- The function document(name) returns root of named document
  - E.g. document("bank-2.xml")/bank-2/account
- Aggregate functions such as sum() and count() can be applied to path expression results
- XQuery does not support groupby, but the same effect can be obtained by nesting FLWR expressions within a result clause
  - More on nested queries later

Joins

- Joins are specified in a manner very similar to SQL

```
for $b in /bank/account, $c in /bank/customer, $d in /bank/depositor
where $b/account-number = $d/account-number
and $c/customer-name = $d/customer-name
return <cust-acct> $c $b </cust-acct>
```

Changing Nesting Structure

- 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>
```
Changing Nesting Structure

- $c/*$ denotes all the children of the node to which $c$ is bound, without the enclosing top-level tag.
- Exercise: write a nested query to find sum of account balances, grouped by branch.

XQuery Path Expressions

- $c/text()$ gives text content of an element without any subelements/tags.
- XQuery path expressions support the “=>” operator for dereferencing IDREFs.
  - Equivalent to the id() function of XPath, but simpler to use.
  - Can be applied to a set of IDREFs to get a set of results.

Sorting in XQuery

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

Other XQuery Features

- User defined functions with the type system of XML Schema.

Storage of XML Data

- 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 Storage of XML Data
- Data must be translated into relational form
  - Advantage: mature database systems
  - Disadvantages: overhead of translating data and queries
- There are many storage options
  - Store strings and indexes
  - Store tree representation
  - Store data content

Store Strings and Indexes
- Store each top level element in a tuple as a string
  - Use a single relation to store all elements, or
  - Use a separate relation for each top-level element type
    - E.g., account, customer, depositor
- Build indexes
  - Store values of subelements/attributes to be indexed, such as customer-name and account-number as extra fields of the relation, and build indices
  - Oracle 9 supports function indices which use the result of a function as the key value. Here, the function should return the value of the required subelement/attribute

Store Strings and Indexes
- 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, allowing faster access to individual elements.
- Drawback:
  - Need to parse strings to access values inside the elements
  - Parsing is slow.

Store Tree Representation
- Represent both the data values and the structural information using relations
- May need to shred XML trees into pieces
- Many options
  - Node & edge tables based on unique id
  - Node & value tables based on positions
  - Node & path tables

Node & Edge Tables
- Relations:
  - nodes(id, type, label, value)
  - edges(parent, child, order)
  - Each node has a unique id
  - Type is either “element” or “attribute”
  - Node table can be split into several tables
    - Tags, Elements, Attributes, Values
- Use joins to evaluate path queries

Node & Value Tables
- Relations:
  - nodes(tag, docId, startPos, endPos, level)
  - value(word, docId, position)
  - Position is the word displacement within the document
  - Use [startPos, endPos] as an interval to determine parent-child and ancestor-descendant relationship
  - Special types of joins are needed to evaluate path queries
**Node & Path Tables**
- Relations:
  - nodes(docId, pathId, tag, order, position, type, value)
  - paths(pathId, path)
- Each node has a unique path from the root of the document
- Need to have index on path expressions
- Positions are ranges, to be used to find subtrees

**Store Data Content**
- Map XML data to relations based on DTD
  - Leaf elements and attributes are mapped to attributes of relations
  - A relation is created for each element type
    - An id attribute to store a unique id for each element
    - All element attributes become relation attributes

**Store Data Content**
- All subelements that occur only once become attributes
  - For text-valued subelements, store the text as attribute value
  - For complex subelements, store the id of the subelement
- Subelements that can occur multiple times are mapped into separate tables
  - Similar to handling of multivalued attributes when converting ER diagrams to tables

**More XML Database Issues**
- Tree-based algebra
- Query optimization issues
- Indexing especially path indexes
- Updates
- Cost estimation
- ...