Chapter 10: XML
XML: Extensible Markup Language
Defined by the WWW Consortium (W3C)
Originally intended as a document markup language not a database language

- Documents have tags giving extra information about sections of the document
  - E.g. `<title> XML </title> <slide> Introduction … </slide>`
- Derived from SGML (Standard Generalized Markup Language), but simpler to use than SGML
- **Extensible**, unlike HTML
  - Users can add new tags, and *separately* specify how the tag should be handled for display
- Goal was (is?) to replace HTML as the language for publishing documents on the Web
The ability to specify new tags, and to create nested tag structures made 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.

```xml
<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>
```
XML: Motivation

- Data interchange is critical in today’s networked world
  - Examples:
    - Banking: funds transfer
    - Order processing (especially inter-company orders)
    - Scientific data
      - Chemistry: ChemML, …
      - Genetics: BSML (Bio-Sequence Markup Language), …
  - Paper flow of information between organizations is being replaced by electronic flow of information

- Each application area has its own set of standards for representing information

- XML has become the basis for all new generation data interchange formats
XML Motivation (Cont.)

- Earlier generation formats were based on plain text with line headers indicating the meaning of fields
  - Similar in concept to email headers
  - Does not allow for nested structures, no standard “type” language
  - Tied too closely to low level document structure (lines, spaces, etc)
- Each XML based standard defines what are valid elements, using
  - XML type specification languages to specify the syntax
    - DTD (Document Type Descriptors)
    - XML Schema
  - Plus textual descriptions of the semantics
- XML allows new tags to be defined as required
  - However, this may be constrained by DTDs
- A wide variety of tools is available for parsing, browsing and querying XML documents/data
Structure of XML Data

- **Tag**: label for a section of data
- **Element**: section of data beginning with `<tagname>` and ending with matching `</tagname>`
- Elements must be properly **nested**
  - Proper nesting
    - `<account> … <balance> …. </balance> </account>`
  - Improper nesting
    - `<account> … <balance> …. </account> </balance>`
  - Formally: every start tag must have a unique matching end tag, that is in the context of the same parent element.
- Every document must have a single top-level element
Example of Nested Elements

<bank-1>
  <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>
  </customer>
  ...
  <account>
  ...
  </account>
  ...
  ...
</bank-1>
Motivation for Nesting

- Nesting of data is useful in data transfer
  - Example: elements representing customer-id, customer name, and address nested within an order element

- Nesting is not supported, or discouraged, in relational databases
  - With multiple orders, customer name and address are stored redundantly
  - Normalization replaces nested structures in each order by foreign key into table storing customer name and address information
  - Nesting is supported in object-relational databases

- But nesting is appropriate when transferring data
  - External application does not have direct access to data referenced by a foreign key
Mixture of text with sub-elements is legal in XML.

Example:

```xml
<account>
  This account is seldom used any more.
  <account-number> A-102 </account-number>
  <branch-name> Perryridge </branch-name>
  <balance>400 </balance>
</account>
```

Useful for document markup, but discouraged for data representation.
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

- An element may have several attributes, but each attribute name can only occur once

```
<account acct-type = “checking” monthly-fee=“5”>
```
Attributes Vs. Subelements

Distinction between subelement and attribute

- In the context of documents, attributes are part of markup, while subelement contents are part of the basic document contents.
- In the context of data representation, the difference is unclear and may be confusing.

Same information can be represented in two ways:

- `<account account-number = “A-101”> …. </account>`
- `<account>`
  `<account-number>A-101</account-number>` …
  `</account>`

Suggestion: use attributes for identifiers of elements, and use subelements for contents.
More on XML Syntax

- Elements without subelements 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
XML data has to be exchanged between organizations

Same tag name may have different meaning in different organizations, causing confusion on exchanged documents

Specifying a unique string as an element name avoids confusion

Better solution: use `unique-name:element-name`

Avoid using long unique names all over document by using XML Namespaces

```xml
<bank xmlns:FB='http://www.FirstBank.com'>
  ...
  <FB:branch>
    <FB:branchname>Downtown</FB:branchname>
    <FB:branchcity>Brooklyn</FB:branchcity>
  </FB:branch>
  ...
</bank>
```
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, increasing use
The type of an XML document can be specified using a DTD

- DTD constraints 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

  ```xml
  <! ELEMENT depositor (customer-name  account-number)>
  <! ELEMENT customer-name (#PCDATA)>
  <! ELEMENT account-number (#PCDATA)>
  ```

- Subelement specification may have regular expressions

  ```xml
  <!ELEMENT bank ( ( account | customer | depositor)+)> 
  ```

  - Notation:
    - “|” - alternatives
    - “+” - 1 or more occurrences
    - “*” - 0 or more occurrences
<!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.

```xml
<!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
]>```

<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
  - $(A \mid B)^*$ 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

- 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 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 ..

- BUT: significantly more complicated than DTDs, not yet widely used.
XML Schema Version of Bank DTD

```xml
<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>

  .... definitions of customer and depositor ....

  <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>
```
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
- Wide variety of other languages have been proposed, and some served as basis for the Xquery standard
  - XML-QL, Quilt, XQL, …
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 (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.

We use the terminology of nodes, children, parent, siblings, ancestor, descendant, etc., which should be interpreted in the above tree model of XML data.
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 those accounts with balance > 400
  - IDREF attributes are not dereferenced automatically (more on this later)
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[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.

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.
  - We omit further details,
A **stylesheet** stores formatting options for a document, usually separately from document.

- E.g. 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
  
  ```xml
  <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

Which one depends on a complex priority scheme/user-defined priorities

We assume only one template matches any element
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.

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

Example output:

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

      results in output of the form
      ```xml
      <customer customer-id="...."> ....
      ```

- `xsl:element` is used to create output elements with computed names
Action of a template can be to recursively apply templates to the contents of a matched element.

E.g.

```xml
<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:

```xml
<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: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>`
Using an `xsl:sort` directive inside a template causes all elements matching the template to be sorted.

- Sorting is done before applying other templates.

E.g.

```xml
<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>
  <xsl:template match="*"/>
```
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 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 uses a

```
for ... let ... where .. result ...
```
syntax
```
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

Simple FLWR 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>
```

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>
```
Path Expressions and Functions

- Path expressions are used to bind variables in the for clause, but can also be used in other places
  - E.g. path expressions can be used in let clause, to bind variables to results of path expressions
- 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 group by, but the same effect can be got by nested queries, with nested FLWR expressions within a `result` clause
  - More on nested queries later
Joins

- Joins are specified in a manner very similar to SQL

```plaintext
for $a \text{ in } /\text{bank/account},
   $c \text{ in } /\text{bank/customer},
   $d \text{ in } /\text{bank/depositor}
where  $a/\text{account-number} = $d/\text{account-number}
   and $c/\text{customer-name} = $d/\text{customer-name}
return <\text{cust-acct}> \ $c \ \$a \ </\text{cust-acct}>
```

- The same query can be expressed with the selections specified as XPath selections:

```plaintext
for $a \text{ in } /\text{bank/account}
   $c \text{ in } /\text{bank/customer}
   $d \text{ in } /\text{bank/depositor}[
      account-number = $a/\text{account-number} \ \text{and}
      customer-name = $c/\text{customer-name}]
return <\text{cust-acct}> \ $c \ \$a</\text{cust-acct}>
```
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.
- Exercise for reader: 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
  - June 2001 version of standard has changed “→” to “=>”
Sorting in XQuery

- **Sortby** clause can be used at the end of any expression. E.g. to return customers sorted by name
  ```xquery
define $c as /bank/customer
for $c in $c
return <customer> $c/* </customer> sortby(name)
```  
- Can sort at multiple levels of nesting (sort by customer-name, and by account-number within each customer)
  ```xquery
<bank-1>
for $c in /bank/customer
return <customer>
  $c/*
  for $d in /bank/depositor[customer-name=$c]
  for $a in /bank/account[account-number=$d]
  return <account> $a/* </account>
  sortby(account-number)
</customer>
sortby(customer-name)
</bank-1>
```
Functions and Other XQuery Features

- User defined functions with the type system of XMLSchema
  ```xml
  function balances(xsd:string $c) returns list(xsd:numeric) {
    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

- Universal and existential quantification in where clause predicates
  - ```xml
    Φ some $e in path satisfies P
    Φ every $e in path satisfies P
  ```

- XQuery also supports If-then-else clauses
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
  - Oracle 9 supports **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

- Map to relations
  - If DTD of document is known, can map data to relations
  - A relation is created for each element type
    - Elements (of type #PCDATA), and attributes are mapped to attributes of relations
    - More details on next slide …

- Benefits:
  - Efficient storage
  - Can translate XML queries into SQL, execute efficiently, and then translate SQL results back to XML

- Drawbacks: need to know DTD, translation overheads still present
Mapping XML Data to Relations (Cont.)

- Relation created for each element type contains
  - 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
E.g. For bank-1 DTD with account elements nested within customer elements, create relations

.customer(id, parent-id, customer-name, customer-stret, customer-city)

- parent-id can be dropped here since parent is the sole root element
- All other attributes were subelements of type #PCDATA, and occur only once

.account (id, parent-id, account-number, branch-name, balance)

- parent-id keeps track of which customer an account occurs under
- Same account may be represented many times with different parents