XML
(Extensible Markup Language)
扩展标记语言
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 （自描述）
  - E.g.
    ```xml
    <bank>
      <account>
        <account_number> A-101 </account_number>
        <branch_name> Downtown </branch_name>
        <balance> 500 </balance>
      </account>
    </bank>
    <depositor>
      <account_number> A-101 </account_number>
      <customer_name> Johnson </customer_name>
    </depositor>
    ```
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
Comparison with Relational Data

- Inefficient: tags, which in effect represent schema information, are repeated
- Better than relational tuples as a data-exchange format
  - Unlike relational tuples, XML data is self-documenting due to presence of tags
  - Non-rigid format: tags can be added
  - Allows nested structures
  - Wide acceptance, not only in database systems, but also in browsers, tools, and applications
Structure of XML Data

- **Tag** (标签) label for a section of data
- **Element** (元素) 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>
    <account>
      <account_number> 123 </account_number>
      <branch_name> Pleasantview </branch_name>
      <balance> 200 </balance>
    </account>
    <account>
      <account_number> 456 </account_number>
      <branch_name> Rosehill </branch_name>
      <balance> 300 </balance>
    </account>
  </customer>
  .
  .
</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
Structure of XML Data (Cont.)

- 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 (属性)

- 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
Namespaces（名字空间）

- 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>
```
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. CDATA stands for “character data”
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
Document Type Definition (DTD)

- 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
  - `<!DOCTYPE document [...]>`
  - `<!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
    - “?” – 0 or 1
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.

```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
]>```
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)
  - \((A \mid B)^*\) 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, decimal, boolean, date, 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 Scheme is integrated with namespaces
- **BUT:** XML Schema is significantly more complicated than DTDs.
<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:
  
  ```xml
  <xs:key name = “accountKey”>
  <xs:selector xpath = “[bank/account]”/>
  <xs:field xpath = “account_number”/>
  
  </xs:key>
  ```
- Foreign key constraint from depositor to account:
  
  ```xml
  <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 id numbers of 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 ...
  (FLWOR) 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
    ```xml
    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:
  ```xml
  for $x in /bank-2/account[balance>400]
  return <account_number> { $x/@account_number } </account_number>
  ```
Joins（连接）

- Joins are specified in a manner very similar to SQL

```xml
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:

```xml
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（嵌套查询）

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

```xml
<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 (排序) 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 <account> $a/* </account>
    </customer>
  } </bank-1>
  ```
Functions, Types 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
  ```xml
  <xsl:template match="/bank-2/customer">
    <xsl:value-of select="customer_name"/>
  </xsl:template>
  </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.

```xml
<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 （结构递归）

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

```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.
    ```xml
    <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
  ```xml
  <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>
  ```
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

```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="*"/>
```
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（字符表示法）

- 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 (映射到关系)

- 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 stored 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

```xml
<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

```sql
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 （简单对象访问协议）
  - 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 （通用描述、发现和集成标准）