Structured Data

- So far, we’ve seen several different data models:
  - the entity-relationship (ER) model
  - the relational model
  - ODL
    - the object-relational model
- All of these models use some type of schema to define the structure of data.
  - ER diagrams
  - relational or object-relational schemas
  - ODL class declarations
- The schema in these models is:
  - *separate* from the data itself
  - *rigid*: all data items of a particular type must have the same set of attributes
Semistructured Data

• In semistructured data:
  • there may or may not be a separate schema
  • the schema is not rigid
    • example: addresses
      • some records may have 3 fields: city, state, and zip
      • other records may group all this info. into a single field

• Semistructured data is self-documenting.
  • information describing the data is embedded with the data

  <course>
  <name>CSCI E-268</name>
  <begin>19:35</begin>
  ...
  </course>

Semistructured Data (cont.)

• Its characteristics facilitate:
  • the integration of information from different sources
  • the exchange of information between applications

• Example: company A receives data from company B
  • A only cares about certain fields in certain types of records
  • B’s data includes:
    • other types of records
    • other fields within the records that company A cares about
  • with semistructured data, A can easily recognize and ignore unexpected elements
  • the exchange is more complicated with structured data
The Semistructured-Data Model

- In the semistructured-data model, a database is a graph.
  - There is a single root node that represents the entire database.
    - there is at least one path from it to every other node
    - may be more than one path, so the graph may not be a tree
  - A leaf node represents a value with an atomic type (int, real, etc.).
  - A non-root interior node represents an entity, object, or struct.
    - example: the shaded node represents the course CSCI E-119.

The directed edges in the model:
- connect the root node to all top-level entities
- connect an entity to attributes of that entity
  - ex: CSCI E-119 is connected to its name and start/end times
- specify relationships between two entities.
  - ex: CSCI E-119 is connected to its room
- The label on an edge specifies the nature of the connection.
As mentioned earlier, the structure used for a given type of information can vary within a single database.

- CSCI E-268's `begin` and `end` times are grouped together into a `time` struct that is an attribute of the course.
- CSCI E-119's `start` and `end` times are specified as two separate attributes.
- This type of heterogeneity can occur when records from two or more databases are merged into a single database.

**XML (Extensible Markup Language)**

- One way of representing semistructured data.
- Like HTML, XML is a *markup language*.
  - it annotates ("marks up") documents with tags
  - tags generally come in pairs:
    - begin tag: `<tagname>`
    - end tag: `</tagname>`
  - example:
    `<li>Like HTML, XML is a markup language.</li>`
    
    `<li>` HTML begin tag for a list item  
    `</li>` HTML end tag for a list item
- Unlike HTML, XML is *extensible*.
  - the set of possible tags – and their meaning – is not fixed
XML Elements

- An XML element is:
  - a begin tag
  - an end tag (in some cases, this is merged into the begin tag)
  - all info. between them.
  - example:
    ```
    <name>CSCI E-268</name>
    ```

- An element can include other nested child elements.
  ```
  <course>
  <name>CSCI E-268</name>
  <begin>19:35</begin>
  ...
  </course>
  ```

- Related XML elements are grouped together into documents.
  - may or may not be stored as an actual text document

XML Attributes

- An element may include attributes that describe it.

- Specified within the element’s begin tag.
  - syntax: name="value"

- Example:
  ```
  <course catalog_number="12345" exam_group="16">
  ```
Attributes vs. Child Elements

<table>
<thead>
<tr>
<th></th>
<th>attribute</th>
<th>child element</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>number of occurrences</strong></td>
<td>at most once in a given element</td>
<td>an arbitrary number of times</td>
</tr>
<tr>
<td><strong>value</strong></td>
<td>always a string</td>
<td>can have its own children</td>
</tr>
</tbody>
</table>

- The string values used for attributes can serve special purposes (more on this later)

Well-Formed XML

- In a *well-formed* XML document:
  - there is a single root element that contains all other elements
    - the root element may optionally be preceded by an XML declaration that looks something like this:
      ```xml
      <?xml version="1.0" standalone="yes"?>
      ```
  - each child element is completely nested within its parent element
    - this would *not* be allowed:
      ```xml
      <course><name>CSCI E-268</name>
      <time>
        <begin>19:35</begin>
        <end>21:35</end>
      </time>
      </course>
      ```
  - The elements need not correspond to any predefined standard.
    - a separate schema is not required
Example of Well-Formed XML Document

```xml
<?xml version="1.0" standalone="yes"?>
<university-data>
  <course>
    <name>CSCI E-119</name>
    <start>19:35</start>
    <end>21:35</end>
  </course>
  <room>
    <bldg>Sever</bldg>
    <num>203</num>
  </room>
  <course>
    <name>CSCI E-268</name>
    <time>
      <begin>19:35</begin>
      <end>21:35</end>
    </time>
  </course>
  ...
</university-data>
```

XML and Semistructured Data

- XML captures the same structure as semistructured data graphs.

```xml
<?xml version="1.0" standalone="yes"?>
<university-data>
  <course>
    <name>CSCI E-119</name>
    <start>19:35</start>
    <end>21:35</end>
  </course>
  ...
  <course>
    <name>CSCI E-268</name>
    <time>
      <begin>19:35</begin>
      <end>21:35</end>
    </time>
  </course>
</university-data>
```

- An element corresponds to a node in the graph.
- A child element corresponds to the child of a node.
- The root element is like the root node in the graph.
- Tags play the same role as edge labels.
XML and Semistructured Data (cont.)

- Nesting elements allows us to capture a semistructured graph that is a tree.
  - at most one path from the root node to a given element
- We'll see later how to capture non-tree graphs.

XML Namespaces

- Because XML is used to combine info. from different sources, a given document may contain different types of elements with the same name.
  - A namespace allows us to disambiguate element names.
    - associate each namespace with a URI (universal resource identifier) like a Web address, so that it's globally unique
    - specify that a given prefix refers to that namespace
    - prepend the prefix to element names from that namespace
- Example:

```xml
<?xml version='1.0' standalone='yes'?>
<university-data>
  <course><name>CSCI E-119</name>
    <start>19:35</start>
    <end>21:35</end>
  </course>
  <course><name>CS 165</name>
    <time><begin>12:00</begin>
      <end>13:00</end>
    </time>
  </course>
</university-data>
```

```xml
<!-- Example: -->
<university-data xmlns:crs='http://www.harvard.edu/course'
  xmlns:per='http://www.harvard.edu/person'>
  <crs:course><crs:name>CSCI E-268</crs:name>
    ...
  </crs:course>
  <per:person>
    <per:name><per:last>Sullivan</per:last>...
  </per:person>
</university-data>
```
Specifying a Separate Schema

- XML doesn’t require a separate schema.

- However, we still need one if we want computer programs to be able to easily process and validate XML documents!

- We'll look at two different ways of doing so:
  - a DTD (document type definition)
  - an XMLSchema document

- The resulting schema:
  - are less flexible than semistructured-data models
  - can still be more flexible than schema from the other models that we’ve seen
    - for example, can include optional components

XML DTDs

- A DTD (document type definition) is one way to specify a schema for an XML document.
  - basic structure:
    ```xml
    <!DOCTYPE root-element-name [
    element and attribute-list declarations
    ]>
    ```

- A DTD can be defined:
  - at the start of a XML document
  - in a separate file whose location is specified in the XML document
DTD Element Declarations

• Syntax 1:
  ```xml
  <!ELEMENT element-name (content-description)>
  ```

• `content-description` is one of the following:
  1. `#PCDATA`: "parsed character data"; text with no child elements
  2. the name of a child element
  3. a regular expression built using instances of 1 and/or 2

• Regular expression syntax:
  - `comp*` the component `comp` appears 0 or more times
  - `comp+` the component `comp` appears 1 or more times
  - `comp?` the component `comp` appears 0 or 1 time
  - `comp1, comp2` `comp1` followed by `comp2`
  - `comp1|comp2` either `comp1` or `comp2`

DTD Element Declarations (cont.)

• Examples:
  ```xml
  <!ELEMENT university-data ((course|room|person)*)>
  ```
  - A `university-data` element contains 0 or more child elements
  - The child elements are either `course`, `room`, or `person` elements, and they may appear in any order

  ```xml
  <!ELEMENT university-data (course+, room*, person*)>
  ```
  - A `university-data` element contains at least one nested `course` element, followed by 0 or more `room` elements, followed by 0 or more `person` elements

  ```xml
  <!ELEMENT course (name, start, end?)>
  ```
  - Meaning:
DTD Element Declarations (cont.)

- Examples (cont.):
  
  ```xml
  <!ELEMENT person_name (#PCDATA|last, first)>
  ```

  - meaning:

  DTD Element Declarations (cont.)

  - Syntax 2:
    
    ```xml
    <!ELEMENT element-name content-category>
    ```

    - `content-category` is one or the following:
      
      - **EMPTY**: the element is an *empty element* that will use a single tag of the form `<name />`
      
      - **ANY**: the element can contain any data (no restrictions)
DTD Attribute-List Declarations

- Syntax:
  ```xml
  <!ATTLIST elem-name att1-name att1-type att1-default 
  att2-name att2-type att2-default 
  ...
  >
  ```

- Attribute types include the following:
  - `CDATA` character data
  - `(val1 | val2 | ...)` an enumeration of possible values
  - `ID` an identifier that must be unique within the document (among all ID attributes – not just this attribute)
  - `IDREF` has a value that is the value of an ID attribute elsewhere in the document
  - `IDREFS` a list of ID values from elsewhere in the document

DTD Attribute-List Declarations (cont.)

- Syntax:
  ```xml
  <!ATTLIST elem-name att1-name att1-type att1-default 
  att2-name att2-type att2-default 
  ...
  >
  ```

- Attribute-default specifications include the following:
  - `#REQUIRED` the attribute must always be specified
  - `#IMPLIED` the attribute is optional and has no default value (omit it if it is not specified)
  - `"default-val"` the attribute is optional; if it isn't specified, use this default value
Capturing Connections Between Entities

- **ID** and **IDREF** attributes can be used to connect elements:

```xml
<course cid="20119" teacher="123456">
  <name>CSCI E-119</name>
</course>

<course cid="20268" teacher="123456">
  <name>CSCI E-268</name>
</course>

<person id="123456" teaches="20119 20268">
  <name><last>Sullivan</last><first>David</first></name>
</person>
```

- Allow XML to capture a semistructured graph that isn't a tree.

---

**DTD for the University-Data Domain**

- ```xml
<!DOCTYPE university-data [ 
  <!ELEMENT university-data ((course|room|person)*)> 
  <!ELEMENT course (cname, start, end)> 
  <!ATTLIST course 
    cid ID #REQUIRED 
    teacher IDREF #REQUIRED 
    room IDREF #IMPLIED> 
  <!ELEMENT cname (#PCDATA)> 
  <!ELEMENT start (#PCDATA)> 
  <!ELEMENT end (#PCDATA)> 
  <!ELEMENT room (building, room_num?)> 
  <!ATTLIST room 
    rid ID #REQUIRED> 
  <!ELEMENT building (#PCDATA)> 
  <!ELEMENT room_num (#PCDATA)> 
  <!ELEMENT person (pname, department*)> 
  <!ATTLIST person 
    pid ID #REQUIRED 
    teaches IDREFS #IMPLIED> 
  <!ELEMENT pname (#PCDATA)> 
  <!ELEMENT last (#PCDATA)> 
  <!ELEMENT first (#PCDATA)> 
  <!ELEMENT department (#PCDATA)> ]>
```
Limitations of DTDs

- ID and IDREF attributes cannot be restricted to identifiers for a specific type of element.
  - why is this problematic?
  -
  -

- More generally, DTDs have an extremely limited type system.
  - examples:
    - can’t specify that an attribute should be an integer
    - can’t specify that all person IDs should have 6 chars

- They don’t allow you to specify namespaces.
  - as a result, element/attribute names in a DTD must be unique
    - example: you can’t have two different name elements – one for people, and one for courses

Limitations of DTDs (cont.)

- They have a limited ability to specify unordered child elements.
  - the | operator allows us to specify some types of unordered collections
    
```xml
<ELEMENT university-data ((course|room|person)*)>
```
  - it doesn't allow us to specify collections in which:
    - all elements must appear at least once
    - we don't care about the order
    - example: "a course must have a name, a start time, and an end time, and their order doesn't matter"

- They are written in a format that is different from XML.
  - we need separate tools/algorithms to parse them
XMLSchema

• An alternative approach to schema specification that addresses the limitations of DTDs.

• Uses one XML document (a schema document) to specify the schema that other XML documents (instance documents of that schema) should follow.

• Schema documents are specified using a schema vocabulary.

• The namespace for the XMLSchema vocabulary is associated with the following URL: http://www.w3.org/2001/XMLSchema.
  - any prefix can be associated with this namespace, but typically it is either xsd or xs is used (we'll use xs)

• We specify the location of the schema document in the root element of the instance document.

Data Types in XMLSchema

• Simple types: used to specify the type of a single value.
  - examples: string, integer, float, double, boolean, date, time, ID, IDREF, IDREFS
  - used for:
    • attributes
    • elements with no attributes and no child elements
    • how would these be specified in a DTD?

• Complex types: used to specify the type of an element that has attributes and/or child elements.

• User-defined types that are derived from the built-in types.
Specifying a Simple-Type Element or Attribute

- Syntax for specifying a simple-type element:
  <xs:element name="element-name" type="xs:element-type" />
  - examples:
    <xs:element name="building" type="xs:string" />
    <xs:element name="room_num" type="xs:integer" />

- Syntax for specifying an attribute:
  <xs:attribute name="attr-name" type="xs:attr-type"
    default="default-val" use="use-specification" />
  - the default and use attributes are optional
  - examples:
    <xs:attribute name="rid" type="xs:ID" use="required" />
    <xs:attribute name="color" type="xs:string"
      default="black" />

Specifying Elements with Child Elements

- Syntax for an element with child elements:
  <xs:element name="element-name">
    <xs:complexType>
      <xs:compositor>
        child element declarations
      </xs:compositor>
      attribute declarations
    </xs:complexType>
  </xs:element>
  - Example:
    <xs:element name="room">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="building" type="xs:string" />
          <xs:element name="room_num" type="xs:integer" />
        </xs:sequence>
        <xs:attribute name="rid" type="xs:ID"
          use="required" />
      </xs:complexType>
    </xs:element>
Specifying Elements with Child Elements (cont.)

- Compositors include the following:
  - **sequence**: the elements must appear in the specified order
  - **choice**: one of the specified elements must appear
  - **all**: the specified elements must each appear once, but in any order
    - see the next slide for how to change the required number of occurrences
- **example**:
  ```xml
  <xs:element name="person">
    <xs:complexType>
      <xs:all>
        <xs:element name="last" type="xs:string"/>
        <xs:element name="first" type="xs:string"/>
      </xs:all>
    </xs:complexType>
  </xs:element>
  ```

Specifying Elements with Child Elements (cont.)

- You can also specify the number of times that a given element (or a compositor applied to a set of elements) should appear.
  - **syntax**: `<xs:element ... minOccurs="min" maxOccurs="max">`
  - if `min` is 0, the element is optional
  - for an unlimited # of occurrences, use `maxOccurs="unbounded"`
Named Types

- You can give a name to a type and then use that type elsewhere in the schema document.
- example:

  ```xml
  <xs:complexType name="personName">
    <xs:all>
      <xs:element name="last" type="xs:string" />
      <xs:element name="first" type="xs:string" />
    </xs:all>
  </xs:complexType>
  ...
  <xs:element name="name" type="personName" />
  ```

- why isn’t the type of the name element "xs:personName"?

Restricted Types

- You can also derive a new type that is based on an existing type.
- One way to do so is to apply a restriction to the values that an existing simple type can take on. Syntax:

  ```xml
  <xs:simpleType name="type-name">
    <xs:restriction base="existing-type">
      restriction declarations
    </xs:restriction>
  </xs:simpleType>
  ```

- Example: specifying that an ID must have a certain length.

  ```xml
  <xs:simpleType name="courseID">
    <xs:restriction base="xs:ID">
      <xs:length value="6" />
    </xs:restriction>
  </xs:simpleType>
  ```
Restricted Types (cont.)

- Other types of restrictions include:
  - `minLength/maxLength`: specify the min/max # of characters
  - `minInclusive`: specify that a number is >= some value
  - `minExclusive`: specify that a number is > some value
  - `maxInclusive`: specify that a number is <= some value
  - `maxExclusive`: specify that a number is < some value
  - `enumeration`: specify possible values for this type
  - `pattern`: specify that a string should match a regular expression

- They all follow this syntax:
  ```xml
  <xs:restriction-type value="value" />
  ```

List Types

- You can also derive a new type that is list of values of an existing type.
  - syntax:
    ```xml
    <xs:simpleType name="list-type-name">
    <xs:list itemType="existing-type-name" />
    </xs:simpleType>
    ```

- example: the definition of the `xs:IDREFS` type
  ```xml
  <xs:simpleType name="IDREFS">
    <xs:list itemType="IDREF" />
  </xs:simpleType>
  ```
Using User-Defined Types for Keys

- Recall one of the limitations of DTDs: ID and IDREF attributes cannot be restricted to identifiers for a specific type of element.

- We can employ user-defined types to get around this problem.
  - restrict the values of the ID type
    - ex: make person IDs begin with the letter P followed by 6 digits
      ```
      <xs:simpleType name="personID">
        <xs:restriction base="xs:ID">
          <xs:pattern value="P[0-9]{6}" />
        </xs:restriction>
      </xs:simpleType>
      ```
  - similarly restrict the values of the corresponding IDREF type
    ```
    <xs:simpleType name="personIDREF">
        <xs:restriction base="xs:IDREF">
          <xs:pattern value="P[0-9]{6}" />
        </xs:restriction>
      </xs:simpleType>
    ```
  
  (note: xs:ID values must start with a letter)

Using User-Defined Types for Keys (cont.)

- Continuing the example from the previous slide:
  - if you need a list of IDREFs, create a list type:
    ```
    <xs:simpleType name="personIDREFS">
      <xs:list itemType="personIDREF" />
    </xs:simpleType>
    ```
XPath Expressions

- Used to specify part of an XML document.
  - treat an XML document as a tree
  - specify a path to the relevant nodes in the tree
    - like a pathname in a hierarchical filesystem

- Can be used when performing queries on an XML database.

- **Absolute** path expressions – those that begin with a slash (/) – specify a path that begins at the root of the document.
  - example: `/university-data/course`
    selects all `course` elements that are children of the `university-data` root element

- Expressions that begin with a double slash (`//`) select elements from anywhere in the document.
  - example: `//name`
    selects all `name` elements, regardless of where they appear

XPath Expressions (cont.)

- Attribute names are preceded by an `@` symbol:
  - example: `//person/@pid`
    selects all `pid` attributes of all `person` elements

- We can specify a particular document as follows:
  
  ```
  document("doc-name") path-expression
  ```
Predicates in XPath Expressions

- Used to select elements that meet a specified condition.
  - surrounded by square brackets
  - applied to the elements selected by the path expression that precedes them.

- Examples:
  - `//course[@teacher="123456"]`
    selects all `course` elements taught by the teacher with ID 123456, where `teacher` is an attribute of `course`
  - `//course[start="19:35"]/name`
    selects the `name` elements of all `course` elements that have a start time of 19:35, where `start` is a child element of `course`
  - `//course[@room]`
    selects all `course` elements with a specified `room` attribute

Predicates in XPath Expressions (cont.)

- Use: `. ` to represent a node selected by the preceding path
  `.. ` to represent the parent of a selected node

- Examples:
  - `//room/room_num[. > 200]`
    selects all `room_num` elements with values > 200
  - `//room[room_num > 200]`
    selects all `room` elements with `room_num` values > 200
  - `//room_num[../building="Sever"]]`
    selects all `room_num` elements for rooms located in Sever
  - how could these same elements be selected w/o using .. ?
  - note: the original expression would produce extra results if there were other types of elements (other than `room`) with `building` and `room_num` child elements
XQuery and FLWOR Expressions

• XQuery is to XML documents what SQL is to relational tables.

• XPath is a subset of XQuery.
  • every XPath expression is a valid XQuery query

• In addition, XQuery provides FLWOR expressions.
  • similar to SQL SELECT commands
  • syntax: FOR $fvar1 IN XPath-expression_f1, $fvar2 IN XPath-expression_f2, ...
  LET $lvar1 := XPath-expression_l1, ...
  WHERE condition
  ORDER BY XPath-expression_o1, ...
  RETURN result-format

• simple example:
  FOR $c in //course
  WHERE $c/start = "19:35"
  RETURN $c/name

FOR Clause

• Like the FROM clause in SQL.

• The query iterates over all combinations of values from the component XPath expressions
  FOR $fvar1 IN XPath-expression_f1, $fvar2 IN XPath-expression_f2, ...
  • $fvar1, $fvar2, etc. are assigned one value at a time

• The WHERE and RETURN clauses are applied to each combination.
**RETURN Clause**

- Like the SELECT clause in SQL.

- Can be used to perform something like a projection of existing elements and/or attributes:

  ```
  FOR $c$ in //course
  WHERE $c/start = '19:35'
  RETURN $c/name
  ```

- To return a concatenated sequence of items for each value produced by the FOR clause:

  - separate them using a comma
  - surround them with parentheses, because the comma operator has higher precedence

  *example:*  
  ```
  FOR $c$ in //course
  WHERE $c/start = '19:35'
  RETURN ($c/name, ' ', $c/start)
  ```

  - you need to explicitly specify any whitespace between items

**RETURN Clause (cont.)**

- Can also reshape the output by constructing new elements:

  ```
  for $d$ in document("depts.xml")/depts/deptno
  let $e := document("emps.xml")/emps/emp[deptno = $d]
  ...
  return <big-dept>
  {
    $d,
    <headcount>{count($e)}</headcount>,
    <avgsal>{avg($e/salary)}</avgsal>
  }
  </big-dept>
  ```

- When a new element is constructed, we use braces to surround expressions that we want to be evaluated.
  - otherwise, they'll be treated as text that is the value of the new element

- Here again, we use commas to separate items in a concatenated sequence of items.
**LET** Clause

- Applied to each combination of values produced by the FOR clause.
- Assigns to a variable the *complete set of values* produced by the corresponding XPath expression.
  - unlike a FOR clause, which assigns the results of the XPath expression one value at a time

**FOR vs. LET**

- Here’s an example that illustrates how they differ:
  ```xml
  for $d in document("depts.xml")/depts/deptno
  let $e := document("emps.xml")/emps/emp[deptno = $d]
  where count($e) >= 10
  order by avg($e/salary) descending
  return <big-dept>
  { $d,
    <headcount>{count($e)}</headcount>,
    <avgsal>{avg($e/salary)}</avgsal>
  }
  </big-dept>
  ```
Nested Queries

- We can nest FLWOR expressions:
  - example: create XML elements that group together the person info. for a given instructor with the courses taught by him/her.
    
    ```xquery
    FOR $p in /university-data/person[@teaches]
    RETURN <instructor-courses>
      {$p,
       FOR $c in /university-data/course
       WHERE $c/@cid = $p/@teaches
       RETURN $c
      }
    </instructor-courses>
    ```

    - result:
      ```xml
      <instructor-courses>
      <person id="123456" teaches="20119 20268">
        <name><last>Sullivan</last>...</name>
      </person>
      <course cid="20119" teacher="123456">
        <name>CSCI E-119</name> ...
      </course>
      ...
      </instructor-courses>
      ```

Using XQuery for Data Exchange

- The `text()` function gives just the value of a simple element.

- This allows us to use an XQuery FLWOR expression to change the tags used for a given type of data:
  ```xquery
  FOR $p in /university-data/person[@teaches]
  RETURN <instructor>
      {$p/name/first/text(), " ", $p/name/last/text()}
      FOR $c in /university-data/course
      WHERE $c/@cid = $p/@teaches
      RETURN <course>{$c/name/text()}</course>
  </instructor>
  ```

- result:
  ```xml
  <instructor>
    <name>David Sullivan</name>
    <course>CSCI E-119</course>
    <course>CSCI E-268</course>...
  </instructor>
  ```
Using XQuery for Data Display

• A FLWOR expression can be used to generate web pages that display the contents of a database:

```html
<html>
<body>
<h2>Faculty and the Courses They Teach</h2>
<ul>
  FOR $p in /university-data/person[@teaches]
  RETURN <li>{{$p/name/first/text(), " ", $p/name/last/text()}}<b>,
  <ul>
    FOR $c in /university-data/course
    WHERE $c/@cid = $p/@teaches
    RETURN <li>{$c/name/text()}}</li>
  </ul>
</li>
</ul>
</body></html>
```

XML-Enabled DBMS

• A DBMS that is based on another model, but has the ability to export and import XML.
  • the XML is not necessarily "visible" in the stored data
  • publish: convert data from another model to XML
  • shred: convert XML data to another data model

• An appropriate logical-to-logical mapping is needed.
  • could derive it from a user-supplied XML schema
Native XML DBMS

- Stores XML documents in a form based on the XML data model.
  - the XML is "visible" in the stored data
    (although not necessarily as it would appear in a text file)
- Can store any XML document.
  - including those with no separate schema
- Two approaches to implementing one:
  1. build it on top of a DBMS that uses another model
     - use a fixed (non-user-defined) logical-to-logical mapping
       that can accommodate any XML document
  2. build it directly on top of a storage engine
     - define the appropriate logical-to-physical mapping

Native XML DBMS: Logical-to-Logical Mappings

- Possible XML-to-relational mappings:
  1) use a relational schema that stores an entire XML document
     as the value of a text attribute:
        \[ \text{document}(id, \text{docstring}) \]
     - useful if you need to preserve the exact bytes of the
       original document (ex: for legal purposes)
     - 100% round-tripping: what goes in is what comes out
     - may also be useful if you have small documents that are
       typically retrieved in their entirety
  2) use a relational schema that encodes the tree structure
     of the document:
        \[ \text{document}(id, \text{name}) \]
        \[ \text{element}(\text{doc_id}, \text{elem_id}, \text{parent_id}, \text{name}) \]
        \[ \text{attribute}(\text{doc_id}, \text{attr_id}, \text{parent_id}, \text{name}) \]
        \[ \text{text}(\text{doc_id}, \text{text_id}, \text{parent_id}, \text{value}) \]
Logical-to-Logical Mappings (cont.)

- Example of mapping #2 from the previous slide (omitting doc_id):
  ```xml
  <univ-data>
    <course>
      <name>CSCI E-268</name>
    </course>
    <person>
      <name><last>Sullivan</last><first>David</first></name>
    </person>
  </univ-data>
  ```

<table>
<thead>
<tr>
<th>elem_id</th>
<th>parent_id</th>
<th>name</th>
<th>text_id</th>
<th>parent_id</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>null</td>
<td>univ-data</td>
<td>500</td>
<td>102</td>
<td>CSCI E-268</td>
</tr>
<tr>
<td>101</td>
<td>100</td>
<td>course</td>
<td>501</td>
<td>110</td>
<td>Sullivan</td>
</tr>
<tr>
<td>102</td>
<td>101</td>
<td>name</td>
<td>502</td>
<td>111</td>
<td>David</td>
</tr>
<tr>
<td>108</td>
<td>100</td>
<td>person</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>109</td>
<td>108</td>
<td>name</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>109</td>
<td>last</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>109</td>
<td>first</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Native XML DBMS: Logical-to-Physical Mappings

- Option 1: Store each document in a flat file.
  - advantages:
    - the mapping is very simple!
    - there are many tools that allow you to manipulate XML that is stored in this way
    - it makes the data easily readable
  - disadvantages?
    - 
    - 

- Option 2: make direct use of a traditional storage engine
  - get the benefits of a DBMS (indexing, transactions, etc.) without the overhead of a logical-to-logical mapping
  - the logical-to-physical mapping is less straightforward
Berkeley DB XML

- A native XML database system built on top of Berkeley DB
- Includes support for queries using XPath/XQuery.
- As in most native XML databases, the fundamental unit of storage is a document.
  - a root element, and possibly some number of nested elements
  - treated like a row/tuple in a relational DB
  - what do you think this would this map to in BDB?

- Related documents are grouped into a container.
  - comparable to a table in a relational DB
  - what do you think this would this map to in BDB?

BDB XML (cont.)

- Example: you could have a container named contacts.dbxml that contains documents that look something like this:

  ```xml
  <contact>
    <name> <last>Codd</last> <first>Ted</first>
  </name>
  <phone type="home">123-456-6789</phone>
  </contact>
  ```

- Different documents in a given container can have different schema.
Logical-to-Physical Mapping in BDB XML

• Each container is stored in its own BDB file.

• A container’s BDB file includes multiple databases:
  • one that stores the contents of the documents
  • others that represent indices maintained on the documents
    • important for good query performance
  • dictionaries that map element and attribute names to numeric codes
    • these codes are used in the indices

Logical-to-Physical Mapping in BDB XML (cont.)

• Two types of containers:
  1) *wholedoc containers*: store entire, intact documents
     • like the first XML-to-relational mapping
     • key-data pairs:
       • key = doc_id
       • data = document as text

  2) *node containers*: store XML documents as collections of nodes – where each node is an element
     • key-data pairs:
       • key = doc_id
       • data = a marshalled element, or document metadata