Semistructured Data and XML

Harvard Extension School

Cody Doucette, Ph.D.

Lecture designed by David G. Sullivan

Structured Data

- We've covered two logical data models thus far:
  - ER diagrams
  - relational schemas
- Both use a *schema* to define the structure of the data.
- 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 fields/attributes

Semistructured Data

- In semistructured data:
  - there may or may not be a *separate* schema
  - the schema is *not* rigid
    - example: capturing people's addresses
      - some records may have 4 separate fields:
        - `street`, `city`, `state`, `zip`
      - other records may use a single `address` field
- Semistructured data is *self-documenting*.
  - information describing the data is embedded with the data

```
<course>
   <name>CS 460</name>
   <begin>1:25</begin>
   …
</course>
```

Semistructured Data (cont.)

- Its features 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

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>
```

HTML begin tag for a list item

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>CS 460</name>
```

- An element can include other nested *child elements.*

```
<course>
   <name>CS 460</name>
   <begin>1:25</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 also include *attributes* that describe it.
- Specified within the element's begin tag.
  - syntax: *name*="*value*"
- Example:

```
<course catalog_number="12345" exam_group="16">
   <name>CS 460</name>
   <begin>1:25</begin>
   …
</course>
```

Attributes vs. Child Elements

	attribute	***child element***
number of occurrences	at most once in a given element	an arbitrary number of times
value	always a string	can have its own children

- 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
    - may optionally be preceded by an XML declaration (more on this in a moment)
  - each child element is completely nested within its parent
    - this would *not* be allowed:

```
<course><name>CS 460</name>
   <time>
      <begin>1:25</begin>
      <end>2:15</end>
</course>
</time>
```

- The elements need not correspond to any predefined standard.
  - a separate schema is not required

Example of an XML Document

```
<?xml version="1.0" standalone="yes"?>
<university-data>
   <course>
        <name>CS 111</name>
        <start>10:10</start>
        <end>11:00</end>
    </course>
    <room>
        <bldg>CAS</bldg>
        <num>B12</num>
    </room>
    <course>
        <name>CS 460</name>
        <time>
            <begin>1:25</begin>
            <end>2:15</end>
        </time>
    </course>
    ...
</university-data>
```

optional declaration

single root element

Specifying a Separate Schema

- XML doesn't require a separate schema.
- However, we still need one if we want programs to:
  - easily process XML documents
  - validate the contents of a given document
- The resulting schema can still be semistructured.
  - for example, can include optional components
  - more flexible than ER models and relational schema

Special Types of Attributes

- `ID` an identifier that must be unique within the document (among *all* `ID` attributes – not just this attribute)
- `IDREF` a single value that is the value of an `ID` attribute elsewhere in the document
- `IDREFS` a *list* of `ID` values from elsewhere in the document

Capturing Relationships in XML

- Two options:

1. store references from one element to other elements using `ID`, `IDREF` and `IDREFS` attributes:

```
<course cid="C20119" teacher="P123456">
  <cname>CS 111</cname>
  …
</course>

<course cid="C20268" teacher="P123456">
  <cname>CS 460</cname>
  …
</course>

<person pid="P123456" teaches="C20119 C20268">
  <pname>
     <last>Sullivan</last>
     <first>David</first>
  </pname>
</person>
```

   - where have we seen something similar?

Capturing Relationships in XML (cont.)

2. use child elements:

```
<course cid="C20119">
  <cname>CS 111</cname>
  <teacher id="P123456">David Sullivan</teacher>
</course>

 …
<person pid="P123456">
  <pname>
     <last>Sullivan</last>
     <first>David</first>
  </pname>
  <courses-taught>
     <course-taught>CS 111</course-taught>
     <course-taught>CS 460</course-taught>
  </courses-taught>
</person>
```

- There are pluses and minuses to each approach.
  - we'll revisit this design issue later in the course

Summary: Features of an XML Document

```
<?xml version="1.0" standalone="yes"?>
<university-data>
  <course cid="C20268" teacher="P123456">
      <name>CS 460</name>
      <start>1:25</start>
      <end>2:15</end>
  </course>
  <course cid="C20119" teacher="P123456" room="CAS 522">
      <name>CS 111</name><start>10:10</start><end>11:00</end>
  </course>
  <person pid="P123456"
          teaches="C20119 C20268">
    <name>
        <last>Sullivan</last>
        <first>David</first>
    </name>
  </person>
  <holiday date="04/15/2019" />
   ...
</university-data>
```

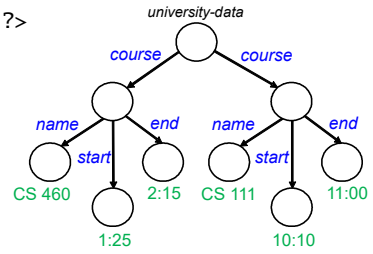
optional declaration

single root element

- *Elements* can have other *child elements* nested inside them.
- *Attributes* are found in the start tag of an element.
- *Simple elements* have no children or attributes.
- *Empty elements* only have a start tag (and possibly attributes)
  - use a / at end of start tag

XML Documents as Trees

```
<?xml version="1.0" standalone="yes"?>
<university-data>
    <course><name>CS 460</name>
      <start>1:25</start>
      <end>2:15</end>
    </course>
      …
    <course><name>CS 111</name>
      <start>10:10</start>
      <end>11:00</end>
    </course>
      …
</university-data>
```

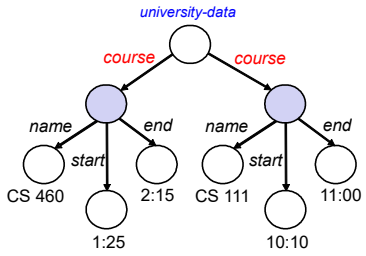


- Elements correspond to nodes in the tree.
  - root element == root node of the entire tree
  - child element == child of a node
  - leaf nodes == empty elements or ones without child elements
- Start tags are edge labels.
- Attributes and text values are data stored in the node.

XPath Expressions

- Used to specify one or more elements or attributes by providing a path to the relevant nodes in the document tree.
  - like a pathname in a hierarchical filesystem
- Expressions that begin with `/` specify a path that begins at the root of the document.

  `/university-data/course`
  - selects all `course` elements that are children of the `university-data` root element

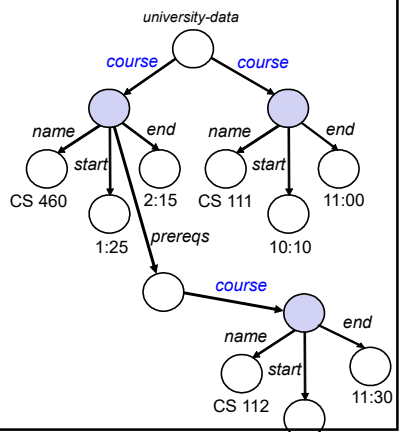


XPath Expressions

- Used to specify one or more elements or attributes by providing a path to the relevant nodes in the document tree.
  - like a pathname in a hierarchical filesystem
- Expressions that begin with `/` specify a path that begins at the root of the document.

  `/university-data/course`
  - selects all `course` elements that are children of the `university-data` root element
- Expressions that begin with `//` select elements from *anywhere* in the document.

  `//course`
  - selects *all* `course` 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*

  - example:

    `document("university.xml")//course/start`

Predicates in XPath Expressions

```
<course cid="C20119" teacher="P123456" room="CAS 522">
   <name>CS 111</name><start>10:10</start><end>11:00</end>
</course>
<course cid="C20268" teacher="P123456">
   <name>CS 460</name><start>1:25</start><end>2:15</end>
</course>
<course cid="C20757" teacher="P778787" room="COM 101">
   <name>CS 112</name><start>11:30</start><end>12:45</end>
</course>
```

- Example:

  `//course[@teacher="P123456"]`

  - selects all `course` elements with a `teacher` **attribute** of "P123456"
- In general, predicates are:
  - surrounded by square brackets
  - applied to elements selected by the preceding path expression

Predicates in XPath Expressions (cont.)

```
<course cid="C20119" teacher="P123456" room="CAS 522">
   <name>CS 111</name><start>10:10</start><end>11:00</end>
</course>
<course cid="C20268" teacher="P123456">
   <name>CS 460</name><start>1:25</start><end>2:15</end>
</course>
<course cid="C20757" teacher="P778787" room="COM 101">
   <name>CS 112</name><start>11:30</start><end>12:45</end>
</course>
```

```
//course[name="CS 460"]
```

- selects all course elements with a name child element whose value is "CS 460"

➔
```
<course cid="C20268" teacher="P123456">
   <name>CS 460</name><start>1:25</start><end>2:15</end>
</course>
```

```
//course[start="1:25"]/name
```

Predicates in XPath Expressions (cont.)

```
<course cid="C20119" teacher="P123456" room="CAS 522">
   <name>CS 111</name><start>10:10</start><end>11:00</end>
</course>
<course cid="C20268" teacher="P123456">
   <name>CS 460</name><start>1:25</start><end>2:15</end>
</course>
<course cid="C20757" teacher="P778787" room="COM 101">
   <name>CS 112</name><start>11:30</start><end>12:45</end>
</course>
```

```
//course[name="CS 112"]/@room
```

Predicates in XPath Expressions (cont.)

```
<course cid="C20119" teacher="P123456" room="CAS 522">
   <name>CS 111</name><start>10:10</start><end>11:00</end>
</course>
<course cid="C20268" teacher="P123456">
   <name>CS 460</name><start>1:25</start><end>2:15</end>
</course>
<course cid="C20757" teacher="P778787" room="COM 101">
   <name>CS 112</name><start>11:30</start><end>12:45</end>
</course>
```

- We can test for the presence of an element or attribute:
  - example: `//course[@room]`
    - selects all `course` elements that have a specified `room` attribute
- We can use the `contains()` function for substring matching:
  - example: `//course[contains(name, "CS")]`

Predicates in XPath Expressions (cont.)

```
<room>
   <building>CAS</building><room_num>212</room_num>
</room>
<room>
   <building>CAS</building><room_num>100</room_num>
</room>
<room>
   <building>KCB</building><room_num>101</room_num>
</room>
<room>
   <building>PSY</building><room_num>228D</room_num>
</room>
```

- Use `.` to represent nodes selected by the preceding path.

  `//room/room_num[. < 200]`
  - selects all `room_num` elements with values < 200

  `//room[room_num < 200]`
  - selects all `room` elements with `room_num` child values < 200

Predicates in XPath Expressions (cont.)

```
<room>
   <building>CAS</building><room_num>212</room_num>
</room>
<room>
   <building>CAS</building><room_num>100</room_num>
</room>
<room>
   <building>KCB</building><room_num>101</room_num>
</room>
<room>
   <building>PSY</building><room_num>228D</room_num>
</room>
```

- Use `..` to represent the *parents* of the nodes selected by the preceding path.

  `//room_num[../building="CAS"]` ➔
  ```
  <room_num>212</room_num>
  <room_num>100</room_num>
  ```

  - selects all `room_num` elements for parent elements that also have a `building` child whose value is "CAS"
  - this is similar: `//room[building="CAS"]/room_num`

Predicates in XPath Expressions (cont.)

```
<room>
   <building>CAS</building><room_num>212</room_num>
</room>
<office>
   <building>CAS</building><room_num>100</room_num>
</office>
<room>
   <building>KCB</building><room_num>101</room_num>
</room>
<office>
   <building>PSY</building><room_num>228D</room_num>
</office>
```

- If there are *other* elements that also have nested `room_num` and `building` elements (like `office` elements above)
  - `//room_num[../building="CAS"]` will get `room_num` children from *all* such elements with a `building` child = "CAS"
  - `//room[building="CAS"]/room_num` will only get `room_num` children from *room* elements with a `building` child = "CAS"

What would this expression select?

```
<course cid="C20119" teacher="P123456" room="CAS 522">
   <name>CS 111</name><start>10:10</start><end>11:00</end>
</course>
<course cid="C20268" teacher="P123456">
   <name>CS 460</name><start>1:25</start><end>2:15</end>
</course>
<course cid="C20757" teacher="P778787" room="COM 101">
   <name>CS 112</name><start>11:30</start><end>12:45</end>
</course>
```

```
//end[../@teacher="P778787"]
```

A.
```
<course cid="C20757" teacher="P778787" room="COM 101">
   <name>CS 112</name><start>11:30</start><end>12:45</end>
</course>
```

B. `<course teacher="P778787"><end>12:45</end></course>`

C. `<end>12:45</end>`

D. none of these

Which of these would select the highlighted element?

```
<course id="C20119" teacher="P123456" room="011">
   <name>CS 111</name><start>10:10</start><end>11:00</end>
</course>
<course id="C20268" teacher="P123456">
   <name>CS 460</name><start>13:25</start><end>14:15</end>
</course>
<course id="C20757" teacher="P778787" room="789">
   <name>CS 112</name><start>11:30</start><end>12:45</end>
</course>
```

A. `//course[start = "10:10"]`

B. `//course/start[. = "10:10"]`

C. `/course/start[. = "10:10"]`

D. `/course[start = "10:10"]`

E. `//start[../end = "11:00"]`

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_f1,
    $fvar2 in xpath_f2,…
let $lvar1 := xpath_l1, …
where condition
order by xpath_o1, …
return result-format
```

FLWOR Expressions

```
for $r in //room[contains(name, "CAS")],
    $c in //course
let $e := //person[contains(@enrolled, $c/@id)]
where $c/@room = $r/@id and count($e) > 20
order by $r/name
return ($r/name, $c/name)
```

- The `for` clause is like the `FROM` clause in SQL.
  - the query iterates over all combinations of values from its XPath expressions (like Cartesian product!)
  - query above looks at combos of CAS `rooms` and `courses`
- The `let` clause is applied to each combo. from the `for` clause.
  - each variable gets the *full set* produced by its XPath expr.
  - unlike a `for` clause, which assigns the results of the XPath expression one value at a time

FLWOR Expressions (cont.)

```
for $r in //room[contains(name, "CAS")],
    $c in //course
let $e := //person[contains(@enrolled, $c/@id)]
where $c/@room = $r/@id and count($e) > 20
order by $r/name
return ($r/name, $c/name)
```

- The `where` clause is applied to the results of `for` and `let`.
- If the `where` clause is true, the `return` clause is applied.
- The `order by` clause can be used to sort the results.

Note: The Location of Predicates

```
for $r in //room[contains(name, "CAS")],
    $c in //course
let $e := //person[contains(@enrolled, $c/@id)]
where $c/@room = $r/@id and count($e) > 20
order by $r/name
return ($r/name, $c/name)
```

- It's sometimes possible to move components of the `where` clause up into the `for` clause as predicates.
- In the above query, we could move the first condition up:

```
for $r in //room[contains(name, "CAS")],
    $c in //course[@room = $r/@id]
let $e := //person[contains(@enrolled, $c/@id)]
where count($e) > 20
order by $r/name
return ($r/name, $c/name)
```

return Clause

```
$c = <course cid="C20119" teacher="P123456" room="CAS 522">
       <name>CS 111</name><start>10:10</start><end>11:00</end>
     </course>
     <course cid="C20268" teacher="P123456">
       <name>CS 460</name><start>13:25</start><end>14:15</end>
     </course>
     <course cid="C20757" teacher="P778787" room="COM 101">
       <name>CS 112</name><start>11:30</start><end>12:45</end>
     </course>
```

- Like the SELECT clause in SQL.
- Can be used to perform something like a projection.

```
for $c in //course
where $c/start > "11:00"
return $c/name
```

➔
```
<name>CS 460</name>
<name>CS 112</name>
```

return Clause (cont.)

- Another example:

```
for $c in //course
where $c/start > "11:00"
return ($c/name, $c/start, " ")
```

- To return multiple elements/attributes for each item:
  - separate them using a comma
  - surround them with parentheses, because the comma operator has higher precedence and would end the FLWOR
  - you can also include string literals
    - above, we specify a blank line after the start time
    - full elements already appear on separate lines, so we don't need spaces for that

Reshaping the Output

- We can reshape the output by constructing new elements:

```
for $c in //course
where $c/start > "11:00"
return <after11-course>
       {$c/name/text(), " - ", $c/start/text()}
       </after11-course>
```

  - the `text()` function gives just the value of a simple element
    - without its start and end tags
  - when constructing a new element, need curly braces around expressions that should be evaluated
    - otherwise, they'll be treated as literal text that is the value of the new element
  - here again, use commas to separate items
    - because we're using `text()`, there are no newlines after the name and start time
    - we use a string literal to put something between them

Reshaping the Output (cont.)

```
<course id="C20119" teacher="P123456" room="011">
  <name>CS 111</name><start>10:10</start><end>11:00</end>
</course>
<course id="C20268" teacher="P123456">
  <name>CS 460</name><start>13:25</start><end>14:15</end>
</course>
<course id="C20757" teacher="P778787" room="789">
  <name>CS 112</name><start>11:30</start><end>12:45</end>
</course>
```

```
for $c in //course
where $c/start > "11:00"
return <after11-course>
       {$c/name/text(), " - ", $c/start/text()}
       </after11-course>
```

- The result will look something like this:

```
<after11-course>CS 460 - 13:25</after11-course>
<after11-course>CS 112 - 11:30</after11-course>
```

for vs. let

- Here's an example that illustrates how they differ:

```
for $d in document("depts.xml")/depts/dept/deptno
let $e := document("emps.xml")/emps/emp[deptno = $d]
where count($e) >= 10
return <big-dept>
       {
           $d,
           <headcount>{count($e)}</headcount>,
           <avgsal>{avg($e/salary)}</avgsal>
       }
       </big-dept>
```

  - the `for` clause assigns to `$d` one `deptno` element at a time
  - for each value of `$d`, the `let` clause assigns to `$e` the *full set* of `emp` elements from that department
  - the `where` clause limits us to depts with >= 10 employees
  - we create a new element for each such dept.
  - we use functions on the set `$e` and on values derived from it

Nested Queries

- We can nest FLWOR expressions:
  - example: group together each instructor's person info. with the courses taught by him/her

```
for $p in //person[@teaches]
return <instructor-courses>
    { $p,
      for $c in //course
      where contains($p/@teaches, $c/@id)
      return $c
    }
    </instructor-courses>
```

  - result:

```
<instructor-courses>
   <person id="P123456" teaches="C20119 C20268">
      <name><last>Sullivan</last>…</name>
   </person>
   <course id="C20119" teacher="P123456">
      <name>CS 111</name> …
   </course>
   …
</instructor-courses>
...
```

Reformatting the Results of the Previous Query

```
for $p in //person[@teaches]
return
  <instructor>
  {<name>{$p/pname/first/text(), " ", $p/pname/last/text()}
   </name>,
   for $c in //course
   where contains($p/@teaches, $c/@id)
   return <course>{$c/name/text()}</course>
  }
  </instructor>
```

- result:

```
<instructor>
    <name>David Sullivan</name>
    <course>CS 111</course>
    <course>CS 460</course>
    …
</instructor>
…
```

Implementing an XML DBMS

- Two possible approaches:
  1) build it on top of a DBMS that uses another model
     - use a logical-to-***logical*** mapping that can accommodate any XML document
     - example: define an XML-to-relational mapping

       XML document → one or more tuples
  2) build it directly on top of a storage engine (or file system!)
     - use an appropriate logical-to-***physical*** mapping
     - similar to what you did in PS 2, Part II!

Approach 1: Logical-to-*Logical* Mappings

- Possible XML-to-relational mappings:
  1) use a schema that stores an entire XML document as the value of a single attribute:
     document(id, contents)
     - useful if you need to preserve the exact bytes of the original document (ex: for legal purposes)
     - may also be useful if you have small documents that are typically retrieved in their entirety
  2) use a schema that encodes the tree structure of the document
     - example: a table for elements that looks something like
       element(id, parent_id, name, value)

Approach 2: 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