XML Query Languages

- XPath – core query language. Very limited, a glorified selection operator. Very useful, though: used in XML Schema, XSLT, XQuery, many other XML standards
- XSLT – a functional style document transformation language. Very powerful, <u>very</u> complicated
- XQuery – W3C standard. Very powerful, fairly intuitive, SQL-style
- SQL/XML – extension of SQL for XML

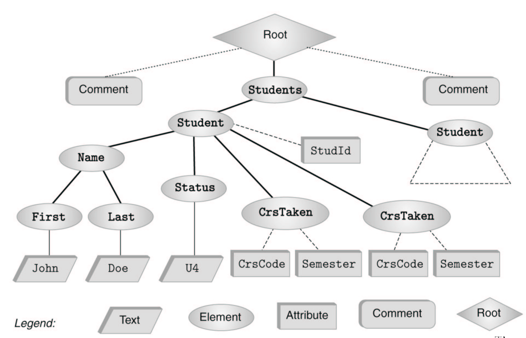
Why Query XML?

- Need to extract parts of XML documents
- Need to transform documents into different forms
- Need to relate – join – parts of the same or different documents

XPath

- Analogous to path expressions in object-oriented languages (e.g., OQL)
- Extends path expressions with query facility
- XPath views an XML document as a tree
  - Root of the tree is a *<u>new</u>* node, which doesn't correspond to anything in the document
  - Internal nodes are elements
  - Leaves are either
    - Attributes
    - Text nodes
    - Comments
    - Other things that we didn't discuss (processing instructions, …)

XPath Document Tree



Document Corresponding to the Tree

- A fragment of the report document that we used frequently

```
<?xml version="1.0" ?>
<!-- Some comment -->
<Students>
    <Student StudId="111111111" >
            <Name><First>John</First><Last>Doe</Last></Name>
            <Status>U2</Status>
            <CrsTaken CrsCode="CS308" Semester="F1997" />
            <CrsTaken CrsCode="MAT123" Semester="F1997" />
    </Student>
    <Student StudId="987654321" >
            <Name><First>Bart</First><Last>Simpson</Last></Name>
            <Status>U4</Status>
            <CrsTaken CrsCode="CS308" Semester="F1994" />
    </Student>
</Students>
<!-- Some other comment -->
```


Terminology

- *Parent*/*child* nodes, as usual
- Child nodes (that are of interest to us) are: of types *text*, *element*, *attribute*
  - We call them *t-children*, *e-children*, *a-children*
  - Also, *et-children* are child-nodes that are either elements or text, *ea-children* are child nodes that are either elements or attributes, etc.
- Ancestor/descendant nodes – as usual in trees

XPath Basics

- An XPath expression takes a document tree as input and returns a multi-set of nodes of the tree
- Expressions that *start* with **/** are *absolute path expressions*
  - Expression **/** – returns root node of XPath tree
  - /Students/Student – returns all Student-elements that are children of Students elements, which in turn must be children of the root
  - /Student – returns empty set (no such children at root)

75

XPath Basics (cont'd)

- *Current* (or *context* node) – exists during the evaluation of XPath expressions (and in other XML query languages)
- **.** – denotes the current node; **..** – denotes the parent
  - foo/bar – returns all bar-elements that are children of foo nodes, which in turn are children of the current node
  - **.**/foo/bar – same
  - **..**/abc/cde – all cde e-children of abc e-children of the parent of the current node
- Expressions that don't start with **/** are *relative* (to the current node)

76

Attributes, Text, etc.

Denotes an attribute

- /Students/Student/@StudentId – returns all StudentId a-children of Student, which are e-children of Students, which are under root
- /Students/Student/Name/Last/text( ) – returns all t-children of Last e-children of …
- /comment( ) – returns comment nodes under root
- XPath provides means to select other document components as well

77

Overall Idea and Semantics

This is called *full* syntax.
We used *abbreviated* syntax before.
Full syntax is better for describing meaning. Abbreviated syntax is better for programming.

- An XPath expression is:
  locationStep1/locationStep2/…
- *Location step*:
  Axis::nodeSelector[predicate]
- Navigation *axis*:
  - *child, parent* – have seen
  - *ancestor, descendant, ancestor-or-self, descendant-or-self* – will see later
  - some other
- *Node selector*: node name or wildcard; e.g.,
  - ./child::Student (we used ./Student, which is an abbreviation)
  - ./child::* – any e-child (abbreviation: ./*)
- *Predicate*: a selection condition; e.g.,
  Students/Student[CourseTaken/@CrsCode = "CS532"]

78

XPath Semantics

- The meaning of the expression locationStep1/locationStep2/… is the set of all document nodes obtained as follows:
  - Find all nodes reachable by locationStep1 from the current node
  - For each node *N* in the result, find all nodes reachable from *N* by locationStep2; take the union of all these nodes
  - For each node in the result, find all nodes reachable by locationStep3, etc.
  - The value of the path expression on a document is the set of all document nodes found after processing the last location step in the expression

79

Overall Idea of the Semantics (Cont'd)

- locationStep1/locationStep2/… means:
  - Find all nodes specified by locationStep1
  - For each such node N:
    - Find all nodes specified by locationStep2 using N as the current node
    - Take union
  - For each node returned by locationStep2 do the same
- locationStep = axis::node[predicate]
  - Find all nodes specified by axis::node
  - Select only those that satisfy predicate

80

More on Navigation Primitives

- 2nd CrsTaken child of 1st Student child of Students:

  /Students/Student[1]/CrsTaken[2]
- All <u>last</u> CourseTaken elements within each Student element:

  /Students/Student/CrsTaken[last( )]

81

Wildcards

- Wildcards are useful when the exact structure of document is not known
- *Descendant-or-self* axis, **//** : allows to descend down any number of levels (including 0)
  - //CrsTaken – all CrsTaken nodes under the root
  - Students//@Name – all Name attribute nodes under the elements Students, who are children of the current node
  - *Note*:
    - ./Last and Last are same
    - .//Last and //Last are <u>different</u>
- The * wildcard:
  - * – any element: Student/*/text()
  - @* – any attribute: Students//@*

82

XPath Queries (selection predicates)

- Recall: Location step = Axis::nodeSelector[<u>predicate</u>]
- Predicate:
  - XPath expression = const | built-in function | XPath expression
  - XPath expression
  - built-in predicate
  - a Boolean combination thereof
- Axis::nodeSelector[<u>predicate</u>] $\subseteq$ Axis::nodeSelector but contains only the nodes that satisfy predicate
- Built-in predicate: special predicates for string matching, set manipulation, etc.
- Built-in function: large assortment of functions for string manipulation, aggregation, etc.

83

XPath Queries – Examples

- Students who have taken CS532:

  //Student[CrsTaken/@CrsCode="CS532"]

  True if : "CS532" $\in$ //Student/CrsTaken/@CrsCode
- Complex example:

  //Student[Status="U3" and starts-with(.//Last, "A")
  and contains(concat(.//@CrsCode), "ESE")
  and not(.//Last = .//First)]
- Aggregation: sum( ), count( )

  //Student[sum(.//@Grade) div count(.//@Grade) > 3.5]

84

Xpath Queries (cont'd)

- Testing whether a subnode exists:
  - //Student[CrsTaken/@Grade] – students who have a grade (for some course)
  - //Student[Name/First or CrsTaken/@Semester
    or Status/text() = "U4"] – students who have either a first name or have taken a course in some semester or have status U4
- Union operator, | :

  //CrsTaken[@Semester="F2001"] | //Class[Semester="F1990"]

  - union lets us define *heterogeneous* collections of nodes

85

XPointer

- XPointer = URL + XPath
  - A URL on steroids
- Syntax:

  url # xpointer (XPathExpr1) xpointer (XPathExpr2) …
  - Follow *url*
  - Compute XPathExpr1
    - Result non-empty? – return result
    - Else: compute XPathExpr2; and so on
- Example: you might click on a link and run a query against your Registrar's database

  http://yours.edu/Report.xml#xpointer(
  //Student[CrsTaken/@CrsCode="CS532"
  and CrsTaken/@Semester="S2002"])

86

XSLT: XML Transformation Language

- Powerful programming language, uses *functional programming paradigm*
- Originally designed as a stylesheet language: this is what “S”, “L”, and “T” stand for
  - The idea was to use it to display XML documents by transforming them into HTML
  - For this reason, XSLT programs are often called *stylesheets*
  - Their use is not limited to stylesheets – can be used to query XML documents, transform documents, etc.
- In wide use, but semantics is very complicated

87

XSLT Basics

- One way to apply an XSLT program to an XML document is to specify the program as a stylesheet in the document *preamble* using a *processing instruction*:

Processing instruction

Preamble

```
<?xml version=“1.0” ?>
… … …
<?xml-stylesheet type=“text/xsl”
                 href=“http://xyz.edu/Report/report.xsl” ?>
… … …
<Report Date=“2002-11-11”>
… … …
</Report>
```

88

Simple Example

- *Extract the list of all students* from **this (hyperlinked) document**

Standard XSLT namespace

Result document skeleton

```
<?xml version=“1.0” ?>
<StudentList xmlns:xsl=“http://www.w3.org/1999/XSL/Transform”
             xsl:version=“1.0” >
   <xsl:copy-of select=“//Student/Name” />
</StudentList>
```

XSLT instruction – copies the result of path expression to stdout

- Result:

```
<StudentList>
   <Name><First>John</First><Last>Doe</Last></Name>
   <Name><First>Bart</First><Last>Simpson</Last></Name>
</StudentList>
```

- Quiz: *Can we use the XSLT namespace as the default namespace in a stylesheet*? What problem might arise?

89

More Complex (Still Simple) Stylesheet

```
<StudentList xmlns:xsl=“http://www.w3.org/1999/XSL/Transform”
             xsl:version=“1.0”>
   <xsl:for-each select=“//Student”>
      <xsl:if test=“count(CrsTaken) &gt; 1” >
         <FullName>
            <xsl:value-of select=“*/Last” /> ,
            <xsl:value-of select=“*/First” />
         </FullName>
      </xsl:if>
   </xsl:for-each>
</StudentList>
```

Extracts ***contents*** *of element, not the element itself (unlike copy-of)*

Result:

```
<StudentList>
   <FullName>
         Doe, John
   </FullName>
</StudentList>
```

90

XSLT Pattern-based Templates

- Where the real power lies
  - … and also where the peril lurks
- *Issue*: how to process XML documents by descending into their structure
- Previous syntax was just a shorthand for template syntax – next slide

91

Full Syntax vs. Simplified Syntax

- Simplified syntax:

```
<StudentList xmlns:xsl=“http://www.w3.org/1999/XSL/Transform”
             xsl:version=“1.0”>
   <xsl:for-each select=“//Student”>
      … … …
   </xsl:for-each>
</StudentList>
```

- Full syntax:

```
<xsl:stylesheet xmlns:xsl=“http://www.w3.org/1999/XSL/Transform”
                xsl:version=“1.0”>
   <xsl:template match=“/” >
      <StudentList>
         <xsl:for-each select=“//Student”>
            … … …
         </xsl:for-each>
      </StudentList>
   </xsl:template>
</xsl:stylesheet>
```

92

Recursive Stylesheets

- A bunch of templates of the form:

```
<xsl:template match="XPath-expression" >
        … tags, XSLT instructions …
</xsl:template>
```

- Template is applied to the node that is *current* in the evaluation process (will describe this process later)
- Template is used if its XPath expression is *matched*:
  - "Matched" means: *current node* ∈ *result set of XPath expression*
  - If several templates match: use the *best matching template* – template with the smallest (by inclusion) XPath expression result set
  - If several of those: other rules apply (see XSLT specs)
  - If *no* template matches, use the matching *default* template
    - There is one default template for *et*-children and one for *a*-children – later

Resursive Traversal of Document

- <xsl:apply-templates/> – XSLT instruction that drives the recursive process of descending into the document tree
- Constructs the list of *et*-children of the current node
- For each node in the list, applies the best matching template
- A typical initial template:

```
<xsl:template match="/" >
        <StudentList>
                <xsl:apply-templates />
        </StudentList>
</xsl:template>
```

Start with the root node – typically the first template to be used in a stylesheet

  - Outputs <StudentList> – </StudentList> tag pair
  - Applies templates to the *et*-children of the current node
  - Inserts whatever output is produced in-between <StudentList> and </StudentList>

Recursive Stylesheet Example

- As before: *list the names of students with* > 1 *courses*:

```
<?xml version="1.0" ?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xsl:version="1.0" >
   <xsl:template match="/" >
        <StudentList>
                <xsl:apply-templates/>
        </StudentList>
   </xsl:template >
   <xsl:template match="//Student" >
      <xsl:if test="count(CrsTaken) &gt; 1" >
        <FullName>
           <xsl:value-of select="*/Last" />
           <xsl:value-of select="*/First" />
        </FullName>
      </xsl if>
   </xsl:template>
   <xsl:template match="text()" >
   </xsl:template>
</xsl:stylesheet>
```

Initial template

The workhorse, does all the job

Empty template – no-op. Needed to block default template for text – later.

Example Dissected

- *Initial template*: starts off, applies templates to *et*-children. The only *et*-child is *Students* element
- Stylesheet has no matching template for *Students*!
- Use *default template*: For *e*-nodes *or* root (/) the default is to go down to the *et*-children:

```
<xsl:template match = " * | / " >
        <xsl:apply-templates />
</xsl:template>
```

- Children of *Students* node are two *Student* nodes – the "workhorse" template matches!
  - For each such (*Student*) node output:
    <FullName>*Last*, *First*</FullName>

Example (cont'd)

- Consider this *expanded* document :

```
<Report>
  <Students>
     <Student StudId="111111111" >
           … … …
     </Student>
     <Student StudId="987654321" >
           … … …
     </Student>
  </Students>
  <Courses>
     <Course CrsCode="CS308" >
        <CrsName>Software Engineering</CrsName>
     </Course>
     … … …
  </Courses>
</Report>
```

Old part

New part

- Then the previous stylesheet has another branch to explore

Example (cont'd)

- No stylesheet template applies to *Courses*-element, so use the default template
- No explicit template applies to children, *Course*-elements – use the default again
- Nothing applies to *CrsName* – use the default
- The child of *CrsName* is a text node. If we used the default here: For text/attribute nodes the XSLT default is

```
<xsl:template match="text( ) | @*" >
        <xsl:value-of select="." />
</xsl:template>
```

i.e., output the contents of text/attribute – we don't want this!

This is why we provided the empty template for text nodes – to suppress the application of the default template

XSLT Evaluation Algorithm

- Very involved
- Not even properly defined in the official XSLT specification!
- More formally described in a research paper by Wadler – can only hope that vendors read this
- Will describe simplified version – will omit the *for-each* statement

99

XSLT Evaluation Algorithm (cont'd)

- Create root node, *OutRoot*, for the output document
- Copy root of the input document, *InRoot*, to output document: $InRoot^R$. Make $InRoot^R$ a child of *OutRoot*
  - Set current node variable: *CN* := *InRoot*
  - Set current node list: *CNL* := <*InRoot*>
  - – *CN* : always the 1st node in *CNL*
  - – When a node *N* is placed on *CNL*, its copy, N^R, goes to the output document (becomes a child of some node – see later)
    - N^R is a marker for where subsequent actions apply in the output document
    - Might be deleted or replaced later
- Find the *best matching template* for *CN* (or default template, if nothing applies)
- Apply this template to *CN* – next slide

100

XSLT Evaluation Algorithm – Application of a Template

- Application of template can cause these changes:

Case A: CN^R is replaced by a subtree

Example: *CN* = *Students* node in **our document**. Assume **our stylesheet** has the following template instead of the initial template (it thus becomes best-matching):

```
<xsl:template  match="//Students" >
        <StudentList>
                <xsl:apply-templates />
        </StudentList>
</xsl:template>
```

Then:

- CN^R is replaced with *StudentList*
- Each child of *CN* (*Students* node) is copied over to the output tree as a child of *StudentList*

101

XSLT Evaluation Algorithm – Application of a Template (cont'd)

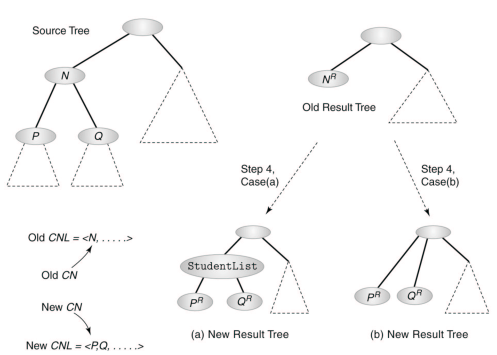
Case B: CN^R is deleted and its children become children of the parent of CN^R

Example: The default template, below, deletes CN^R when applied to any node:

```
<xsl:template  match="* | /" >
        <xsl:apply-templates />
</xsl:template>
```

102

The Effect of apply-templates on Document Tree



XSLT Evaluation Algorithm (cont'd)

- In both cases (A & B):
  - – If *CN* has no *et*-children, *CNL* becomes shorter
  - – If it does have children, *CNL* is longer or stays the same length
  - – The order in which *CN*'s children are placed on *CNL* is their order in the source tree
  - – The new 1st node in *CNL* becomes the new *CN*
- Algorithm terminates when *CNL* is empty
  - – Be careful – might not terminate (see next)

104

XSLT Evaluation Algorithm –Subtleties

- apply-templates instruction can have select attribute:
  - <xsl:apply-templates select="node()" /> – equivalent to the usual <xsl:apply-templates />
  - <xsl:apply-templates select="@* | text()" /> – instead of the *et*-children of *CN*, take *at*-children
  - <xsl:apply-templates select=".." /> – take the parent of *CN*
  - <xsl:apply-templates select="." /> – will cause an infinite loop!!
- Recipe to guarantee termination: make sure that *select* in apply-templates selects nodes only from a subtree of *CN*

Advanced Example

- *Example*: take any document and replace attributes with elements. So that

```
<Student StudId="111111111">
  <Name>John Doe</Name>
  <CrsTaken CrsCode="CS308" Semester="F1997" />
</Student>
```

would become:

```
<Student>
  <StudId>111111111</StudId>
  <Name>John Doe</Name>
  <CrsTaken>
    <CrsCode>CS308</CrsCode> <Semester>F1997</Semester>
  </CrsTaken>
</Student>
```


Advanced Example (cont'd)

- *Additional requirement*: don't rely on knowing the names of the attributes and elements in input document – should be completely general. Hence:
  1. Need to be able to output elements whose name is not known in advance (we don't know which nodes we might be visiting)
     - Accomplished with xsl:element instruction and Xpath functions current( ) and name( ):

```
<xsl:element name="name(current())" >
        Where am I?
</xsl:element>
```

If the current node is *foobar*, will output:

```
<foobar>
        Where am I?
</foobar>
```


Advanced Example (cont'd)

2. Need to be able to copy the current element over to the output document
   - The *copy-of* instruction won't do: it copies elements over with all their belongings. But remember: *we don't want attributes to remain attributes*
   - So, use the copy instruction
     - Copies the current node to the output document, but <u>without any of its children</u>

```
<xsl:copy>
   … XSLT instructions, which fill in the body
     of the element being copied over …
</xsl:copy>
```


Advanced Example (cont'd)

```
<xsl:stylesheet … … …>
  <xsl:template match="node()">                  ← Process elements/text
     <xsl:copy>
       <xsl:apply-templates select="@*" />       ← Process a-children of current element
       <xsl:apply-templates />                   ← Process et-children of current element
     </xsl:copy>
  <xsl:template>
   <xsl:template match="@*">                     ← Deal with attributes separately
     <xsl:element name="name(current( ))" >      } Convert attribute to element
          <xsl:value-of select="." />            }
     </xsl:element>
  <xsl:template>
</xsl:stylesheet>
```

<… Attr="foo" >
becomes
<Attr>foo</Attr>

Limitations of XSLT as a Query Language

- Programming style unfamiliar to people trained on SQL
- Most importantly: Hard to do joins, i.e., <u>*real*</u> queries
  - Requires the use of variables (we didn't discuss)
  - Even harder than a simple nested loop (which one would use in this case in a language like C or Java)

XQuery – XML Query Language

- Integrates XPath with earlier proposed query languages: XQL, XML-QL
- SQL-style, not functional-style
- Much easier to use as a query language than XSLT
- Can do pretty much the same things as XSLT, but typically easier
- 2003: XQuery 1.0 standard

111

XQuery Basics

- General structure:

```
FOR      variable declarations
WHERE    condition
RETURN   document
```

XQuery expression

- Example:

```
FOR $t IN document("http://xyz.edu/transcript.xml")//Transcript
WHERE  $t/CrsTaken/@CrsCode = "MAT123"
RETURN $t/Student
```

- Result:

```
<Student StudId="111111111" Name="John Doe" />
<Student StudId="123454321" Name="Joe Blow" />
```

This document on next slide

112

transcript.xml

```
<Transcripts>
  <Transcript>
    <Student StudId="111111111" Name="John Doe" />
    <CrsTaken CrsCode="CS308" Semester="F1997" Grade="B" />
    <CrsTaken CrsCode="MAT123" Semester="F1997" Grade="B" />
    <CrsTaken CrsCode="EE101" Semester="F1997" Grade="A" />
    <CrsTaken CrsCode="CS305" Semester="F1995" Grade="A" />
  </Transcript>
  <Transcript>
    <Student StudId="987654321" Name="Bart Simpson" />
    <CrsTaken CrsCode="CS305" Semester="F1995" Grade="C" />
    <CrsTaken CrsCode="CS308" Semester="F1994" Grade="B" />
  </Transcript>
```

… … *cont'd* … …

113

transcript.xml (cont'd)

```
  <Transcript>
    <Student StudId="123454321" Name="Joe Blow" />
    <CrsTaken CrsCode="CS315" Semester="S1997" Grade="A" />
    <CrsTaken CrsCode="CS305" Semester="S1996" Grade="A" />
    <CrsTaken CrsCode="MAT123" Semester="S1996" Grade="C" />
  </Transcript>
  <Transcript>
    <Student StudId="023456789" Name="Homer Simpson" />
    <CrsTaken CrsCode="EE101" Semester="F1995" Grade="B" />
    <CrsTaken CrsCode="CS305" Semester="S1996" Grade="A" />
  </Transcript>
</Transcripts>
```

114

XQuery Basics (cont'd)

- Previous query doesn't produce a well-formed XML document; the following does:

```
<StudentList>
{
   FOR $t IN document("transcript.xml")//Transcript
   WHERE $t/CrsTaken/@CrsCode = "MAT123"
   RETURN $t/Student
}
</StudentList>
```

Query inside XML

- FOR binds $t to Transcript elements one by one, filters using WHERE, then places Student-children as *e*-children of StudentList using RETURN

115

Document Restructuring with XQuery

- *Reconstruct lists of students taking each class using the* Transcript *records*:

```
FOR $c IN distinct-values(document("transcript.xml")//CrsTaken)
RETURN
  <ClassRoster CrsCode = {$c/@CrsCode}  Semester = {$c/@Semester}>
  {
    FOR $t IN document("transcript.xml")//Transcript
    WHERE $t/CrsTaken/[@CrsCode = $c/@CrsCode and
                       @Semester = $c/@Semester]
    RETURN $t/Student
           ORDER BY $t/Student/@StudId
  }
  </ClassRoster>
  ORDER BY $c/@CrsCode
```

Query inside RETURN – similar to query inside SELECT in OQL

116

Document Restructuring (cont'd)

- *Output elements have the form*:

```
<ClassRoster CrsCode="CS305" Semester="F1995" >
  <Student StudId="111111111" Name="John Doe" />
  <Student StudId="987654321" Name="Bart Simpson" />
</ClassRoster>
```

- *Problem*: the above element will be output *twice* – once when $c is bound to

```
<CrsTaken CrsCode="CS305" Semester="F1995" Grade="A" />
```

and once when it is bound to

(Bart Simpson's) (John Doe's)

```
<CrsTaken CrsCode="CS305" Semester="F1995" Grade="C" />
```

Note: grades are different – distinct-values( ) won't eliminate transcript records that refer to same class!

117

Document Restructuring (cont'd)

- *Solution*: instead of

```
FOR $c IN distinct-values(document("transcript.xml")//CrsTaken)
```

use

```
FOR $c IN document("classes.xml")//Class
```

(*Document on next slide*)

where classes.xml lists course offerings (course code/semester) *explicitly* (no need to extract them from transcript records).

Then $c is bound to each class exactly once, so each class roster will be output exactly once

118

http://xyz.edu/classes.xml

```
<Classes>
  <Class CrsCode="CS308" Semester="F1997" >
      <CrsName>SE</CrsName> <Instructor>Adrian Jones</Instructor>
  </Class>
  <Class CrsCode="EE101" Semester="F1995" >
      <CrsName>Circuits</CrsName> <Instructor>David Jones</Instructor>
  </Class>
  <Class CrsCode="CS305" Semester="F1995" >
      <CrsName>Databases</CrsName> <Instructor>Mary Doe</Instructor>
  </Class>
  <Class CrsCode="CS315" Semester="S1997" >
      <CrsName>TP</CrsName> <Instructor>John Smyth</Instructor>
  </Class>
  <Class CrsCode="MAR123" Semester="F1997" >
      <CrsName>Algebra</CrsName> <Instructor>Ann White</Instructor>
  </Class>
</Classes>
```

119

Document Restructuring (cont'd)

- *More problems*: the above query will list classes with no students. Reformulation that avoids this:

(*Test that classes aren't empty*)

```
FOR $c IN document("classes.xml")//Class
WHERE document("transcripts.xml")
        //CrsTaken[@CrsCode = $c/@CrsCode
                  and @Semester = $c/@Semester]
RETURN
  <ClassRoster CrsCode = {$c/@CrsCode} Semester = {$c/@Semester}>
  {
    FOR $t IN document("transcript.xml")//Transcript
    WHERE $t/CrsTaken[@CrsCode = $c/@CrsCode and
                      @Semester = $c/@Semester]
    RETURN $t/Student ORDER BY $t/Student/@StudId
  } </ClassRoster>
ORDER BY $c/@CrsCode
```

120

XQuery Semantics

- So far the discussion was informal
- XQuery *semantics* defines what the expected result of a query is
- Defined analogously to the semantics of SQL

121

XQuery Semantics (cont'd)

- *Step 1*: Produce a list of bindings for variables
  - The FOR clause binds each variable to a *list* of nodes specified by an XQuery expression.
    The expression can be:
    - An XPath expression
    - An XQuery query
    - A function that returns a list of nodes
  - End result of a FOR clause:
    - Ordered list of tuples of document nodes
    - Each tuple is a binding for the variables in the FOR clause

122

XQuery Semantics (cont'd)

Example (bindings):

- Let FOR declare $A and $B
- Bind $A to document nodes {v,w}; $B to {x,y,z}
- Then FOR clause produces the following list of bindings for $A and $B:
  - $A/v, $B/x
  - $A/v, $B/y
  - $A/v, $B/z
  - $A/w, $B/x
  - $A/w, $B/y
  - $A/w, $B/z

123

XQuery Semantics (cont'd)

- *Step 2*: filter the bindings via the WHERE clause
  - Use each tuple binding to substitute its conponents for variables; retain those bindings that make WHERE true
  - Example: WHERE $A/CrsTaken/@CrsCode = $B/Class/@CrsCode
    - Binding: $A/w, where w = <CrsTaken CrsCode="CS308" …/>
      $B/x, where x = <Class CrsCode="CS308" … />
    - Then w/CrsTaken/@CrsCode = x/Class/@CrsCode, so the WHERE condition is satisfied & binding retained

124

XQuery Semantics (cont'd)

- *Step 3*: Construct result
  - For each retained tuple of bindings, instantiate the RETURN clause
  - This creates a fragment of the output document
  - Do this for each retained tuple of bindings in sequence

125

User-defined Functions

- Can define functions, even recursive ones
- Functions can be called from within an XQuery expression
- Body of function is an XQuery expression
- Result of expression is returned
  - Result can be a primitive data type (integer, string), an element, a list of elements, a list of arbitrary document nodes, …

126

XQuery Functions: Example

- Count the number of *e*-children recursively:

Function signature

```
DECLARE FUNCTION countNodes($e AS element()) AS integer {
    RETURN
      IF empty($e/*) THEN 0
      ELSE
         sum(FOR $n IN $e/* RETURN countNodes($n))
          + count($e/*)
}
```

XQuery expression

Built-in functions sum, count, empty

127

Class Rosters (again) Using Functions

```
DECLARE FUNCTION extractClasses($e AS element()) AS element()* {
   FOR $c IN $e//CrsTaken
   RETURN <Class CrsCode={$c/@CrsCode} Semester={$c/@Semester} />
}
<Rosters>
   FOR $c IN
      distinct-values(FOR $d IN document("transcript.xml") RETURN extractClasses($d) )
   RETURN
       <ClassRoster CrsCode = {$c/@CrsCode} Semester = {$c/@Semester} >
       {
         LET $trs := document("transcript.xml")
         FOR $t IN $trs//Transcript[CrsTaken/@CrsCode=$c/@CrsCode and
                                    CrsTaken/@Semester=$c/@Semester]
         RETURN $t/Student
         ORDER BY $t/Student/@StudId
       }
      </ClassRoster>
</Rosters>
```

128

Converting Attributes to Elements with XQuery

- An XQuery reformulation of a **previous XSLT query** – much more straightforward (but ignores text nodes)

```
DECLARE FUNCTION convertAttributes($a AS attribute()) AS element() {
   RETURN  element {name($a)} {data($a)}
}
DECLARE FUNCTION convertElement($e AS node()) AS element() {
   RETURN   element {name($e)}
         {
           { FOR $a IN $e/@* RETURN convertAttribute ($a) } ,
           IF empty($e/*) THEN $e/text( )
           ELSE { FOR $n IN $e/* RETURN convertElement($n) }
         }
}
RETURN convertElement(document("my-document")/*)
```

Concatenate results

Computed element

The actual query: Just a RETURN statement!!

129

Integration with XML Schema and Namespaces

- Let type FOO be defined in http://types.r.us/types.xsd:

```
IMPORT SCHEMA "http://types.r.us  at
                                 http://types.r.us/types.xsd"
DECLARE NAMESPACE trs = "http://types.r.us"
DECLARE NAMESPACE xsd =
   "http://www.w3.org/2001/XMLSchema"
DECLARE FUNCTION  doSomething($x AS trs:FOO)
                                  AS xsd:string {
   … … …
}
```

Namespace

Location

Prefix for namespace

130

Grouping and Aggregation

- Does not use separate grouping operator
  - Recall that OQL does not need one either
  - Subqueries inside the RETURN clause obviate this need (like subqueries inside SELECT did so in OQL)
- Uses built-in aggregate functions count, avg, sum, etc. (some borrowed from XPath)

131

Aggregation Example

- *Produce a list of students along with the number of courses each student took*:

```
FOR $t IN document("transcripts.xml")//Transcript,
      $s IN $t/Student
LET  $c := $t/CrsTaken
RETURN
     <StudentSummary StudId = {$s/@StudId}  Name = {$s/@Name}
                          TotalCourses = {count(distinct-values($c))} />
ORDER BY StudentSummary/@TotalCourses
```

- The *grouping effect* is achieved because $c is bound to a *new* set of nodes for *each* binding of $t

132

Quantification in XQuery

- XQuery supports explicit quantification: SOME (∃) and EVERY (∀)
- *Example*:

```
FOR $t IN document("transcript.xml")//Transcript
WHERE SOME $ct IN $t/CrsTaken
                 SATISFIES $ct/@CrsCode = "MAT123"
RETURN $t/Student
```

"Almost" equivalent to:

```
FOR $t IN document("transcript.xml")//Transcript,
       $ct IN $t/CrsTaken
WHERE $ct/@CrsCode = "MAT123"
RETURN $t/Student
```

- *Not equivalent, if students can take same course twice*!

133

Implicit Quantification

- Note: in SQL, variables that occur in FROM, but not SELECT are implicitly quantified with ∃
- In XQuery, variables that occur in FOR, but not RETURN are similar to those in SQL. However:
  - In XQuery variables are bound to document nodes
    - Two nodes may look textually the same (e.g., two different instances of the same course element), but they are still different nodes and thus different variable bindings
    - Instantiations of the RETURN expression produced by binding variables to <u>different nodes</u> are output <u>even if these instantiations are textually identical</u>
  - In SQL a variable can be bound to the same value only once; identical tuples are not output twice (in theory)
  - *This is why the two queries in the previous slide are not equivalent*

134

Quantification (cont'd)

- Retrieve all classes (from classes.xml) where each student took MAT123
  - Hard to do in SQL (before SQL-99) because of the lack of explicit quantification

```
FOR $c IN document(classes.xml)//Class
LET $g := { -- Transctipt records that correspond to class $c
   FOR $t IN document("transcript.xml")//Transcript
   WHERE $t/CrsTaken/@Semester = $c/@Semester
            AND $t/CrsTaken/@CrsCode = $c/@CrsCode
   RETURN $t
         }
WHERE EVERY $tr IN $g SATISFIES
                  NOT empty($tr[CrsTaken/@CrsCode="MAT123])
RETURN $c ORDER BY $c/@CrsCode
```

135

SQL/XML – Extending SQL

- In the past, SQL was extended for OO:
  - added values for reference, tuple(row type), and collection(arrays), …
  - kind of took over ODL and OQL of ODMG
- Currently, SQL is extended for XML:
  - adding data types and functions for XML
  - will it take over XQuery?

136

Why SQL/XML

- Publish contents of SQL tables or entire DB as XML doc – need convention for translating primitive SQL data types
- Create XML doc out of SQL query results – need extension of SQL queries to create XML elements
- Store XML doc in relational DB and query them – need extension of SQL to use XPath for tree structures

137

Publishing Relations as XML Doc: Tables

- Current proposal: no built-in function from table to XML, but can create arbitrary XML using SELECT
- Encoding relational data in XML:
  - Entire relation: an element named after the relation
  - Each row: an elememt named 'row'
  - Each attribute: an element named after the attribute

138

Publishing Relations as XML Doc: Tables

Professor

Id	Name	DeptId
1024	Bob Smith	CS
3093	Amy Doe	EE
…		

```
<Professor>
   <row>
     <Id>1024</Id><Name>Bob Smith</Name><DeptId>CS<DeptId>
   </row>
   < row>
     <Id>3093</Id><Name>Amy Doe</Name><DeptId>EE<DeptId>
   </row >
   ...
</Professor>
```

139

Publishing Relations as XML Doc: Schema

Schema Id: INTEGER
Name: CHAR(50)
DeptId: CHAR(3)

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
      targeNamespace="http://xyz.edu/Admin">
   <element name="Professor">
    <complexType>
     <sequence>
      <element name="row" minOccurs="0" maxOccurs="unbounded">
        <complexType>
         <sequence>
           <element name="Id" type="integer"/>
           <element name="Name" type="CHAR_50"/>
           <element name="DeptId" type="CHAR_3"/>
         </sequence>
      …
</schema>
```

140

Publishing Relations as XML Doc: Schema

CHAR_len: standard conventions in SQL/XML for CHAR(len) in SQL, defined as

```
<simpleType>
  <restriction base="string">
    <length value="50">
  </restriction>
</simpleType>
```

A lot of the standard deals with such primitives, as well as user-defined types (defined using CREATE DOMAIN).

141

Storing XML in Relational DB: Data Type XML

Not stored as a string, but natively as a tree structure.

Support navigation via efficient storage and indexing.

```
CREATE  TABLE  StudentXML (
  Id          INTEGER,
  Details     XML )
```

where Details attribute contains

```
<Student>
  <Name><First>Amy</First><Last>Doe</Last></Name>
  <Status>U4</Status>
  <CrsTaken CrsCode="305" Semester="F2003"/>
  <CrsTaken CrsCode="336" Semester="F2003"/>
</Student>
```

142

Storing XML in Relational DB: Data Type XML

To validate

```
CREATE  TABLE  StudentXML (
  Id          INTEGER,
  Details     XML,
  CHECK(Details ISVALID INSTNACE OF 'http://xyz.edu/student.xsd') )
```

assuming schema is stored at http://xyz.edu/student.xsd

143

Creating XML from Queries: Functions XMLELEMENT, XMLATTRIBUTES

An SQL query does not return XML directly.

Can produce particular column as an XML document.

```
SELECT  P.Id,  XMLELEMENT (
        NAME  "Prof",                              --- element name
        XMLATTRIBUTES (P.DeptId AS "Dept"),        --- attributes
        P.Name                                     --- content
     ) AS Info
FROM Professor P
```

produce tuples

```
1024, <Prof Dept="CS">Bob Smith</Prof>
3093, <Prof Dept="EE">Amy Doe</Prof>
```

144

Creating XML from Queries: Functions XMLELEMENT, XMLATTRIBUTES

XMLELEMENT can be nested.

```
SELECT XMLELEMENT (NAME "Prof"
                        XMLELEMENT(NAME "Id", P.Id),
                        XMLELEMENT(NAME "Name", P.Name),
                        XMLELEMENT(NAME "DeptId", P.DeptId),
                       ) AS ProfElement
FROM Professor P
```

produce tuples

```
<Prof><Id>1024</Id><Name>Bob Smith</Name><DeptId>CS</DeptId>
</Prof>
<Prof><Id>3093</Id><Name>Amy Doe</Name><DeptId>EE</DeptId>
</Prof>
```

145

Creating XML from Queries: Function XMLGEN

```
SELECT XMLGEN ('<Prof>
                 <Id>{$I}</Id><Name>{$N}</Name><DeptId>{$D}</DeptId>
                </Prof>',             --- template with placeholder variables
                P.Id AS I,            --- values of exps subst for placehoders
                P.Name AS N,
                P.DeptId AS D,
               ) AS ProfElement
FROM Professor P
```

Placeholder can occur in position of XML elements and attibutes.

Expressions can be XML-generating exps or SELECT statements.

In example above, can replace

```
<Id>{$I}</Id>   with  {$I}
P.Id AS I       with  XMLELEMENT(NAME "Id", P.Id) AS I
```

146

Creating XML from Queries: Grouping and XMLAGG

In XQuery: group elements as children of another element by putting a subquery in RETURN clause of parent query.

In SQL/XML:

putting SELECT inside XML func in SELECT clause of parent.

Example: group the taken courses by student ids

```
SELECT  XMLELEMENT (
     NAME "Student",
     XMLATTRIBUTES(S.Id AS "Id"),
     (SELECT  XMLELEMENT(NAME "CrsTaken",
                              XMLATTRIBUTES(T.CrsCode AS "CrsCode",
                                                  T.Semester AS "Semester"))
      FROM Transcript T
      WHERE S.Id=T.StudId))
FROM Student S
```

Return set of 1-tuples, not list of elements. waiting for standard for conversion.

147

Creating XML from Queries: Grouping and XMLAGG

Same example: group the taken courses by student ids

```
SELECT  XMLELEMENT (
    NAME "Student",
    XMLATTRIBUTES(S.Id AS "Id"),
    XMLAGG(XMLELEMENT(Name "CrsTaken",
                              XMLATTRIBUTES(T.CrsCode AS "CrsCode",
                                                   T.Semester AS "Semester"))
           ORDER BY T.CrsCode))
FROM Student S, Transcript T
WHERE S.Id=T.StudId
GROUP BY S.Id
```

148

Querying XML Stored in Relations: XMLEXTRACT, XMLEXISTS

Use XPath expressions.

Can be in both SELECT and WHERE clauses.

Example: return Ids and names of students who have status U3 and have taken MAT123:

```
SELECT  S.Id,  XMLEXTRACT(S.Details, ’//Name’)
FROM  StudentXML S
WHERE XMLEXTRACT(S.Details, ’//Status/text()’)=’U3’ AND
          XMLEXTRACT(S.Details, ’//CrsTaken/@CrsCode’)=’MAT124’
```

149

Querying XML Stored in Relations: XMLEXTRACT, XMLEXISTS

Tell whether the set of nodes returned by XPath expression is empty.

Example: return Ids and names of students who have taken any course

```
SELECT  S.Id,  SMLEXTRACT(S.Details, ’//Name’)
FROM  StudentXML  S
WHERE XMLEXISTS(S.Details, ’//CrsTaken’)
```

150

Modifying Data in SQL/XML: XMLPARSE, XMLVALIDATE

XML stored as appropriately indexed tree structure, so need to parse.

```
INSERT  INTO  StudentXML(Id, Details)
VALUES(12343,
         XMLPARSE(
           ’<Student>
             <Name><First>Bob</First><Last>Smith</Last></Name>
             <Status>U4</Status>
             <CrsTake CrsCode="CS305" Semester="F2003"/>
             <CrsTake CrsCode="CS339" Semester="S2004"/>
           </Student>’))
```

151

Modifying Data in SQL/XML: XMLPARSE, XMLVALIDATE

To validate

```
INSERT  INTO  StudentXML(Id, Details)
VALUES(12343,
         XMLVALIDATE(XMLPARSE(
           ’<Student>
             <Name><First>Bob</First><Last>Smith</Last></Name>
             <Status>U4</Status>
             <CrsTake CrsCode="CS305" Semester="F2003"/>
             <CrsTake CrsCode="CS339" Semester="S2004"/>
           </Student>’)))
```

waiting for standard for option of specifying schema location

152

XMLSERIALIZE: Reverse of XMLPARSE

To store XML as string or

use by a host language that does not understand XML

Example: return Ids and names of students who have taken any course

```
EXEC SQL DECLARE GetEnrolled CURSOR FOR
SELECT  S.Id,  XMLSERIALIZE(XMLEXTRACT(S.Details,'//Name'))
FROM  StudentXML  S
WHERE XMLEXISTS(S.Details, '//CrsTaken')
```

return ids and strings, which can then be processed by

```
EXEC SQL GetEnrolled INTO :stuDId, :details
```

153