XML Query Languages
Outline

- Semistructured data
- Introduction to XML
- Introduction to DTDs
- XPath – core query language for XML
- XQuery – full-featured query language for XML
Semistructured Data

- An HTML document to be displayed on the Web:

```html
<dt>Name: John Doe
  <dd>Id: s111111111
  <dd>Address:  
    <ul>
      <li>Number: 123</li>
      <li>Street: Main</li>
    </ul>
  </dd>
</dt>

<dt>Name: Joe Public
  <dd>Id: s222222222
      … … … …
</dt>
```

*HTML does not distinguish between attributes and values*
Why study XML?

- Huge demands for data exchange
  - across platforms
  - across enterprises

- Huge demands for data integration
  - heterogeneous data sources
  - data sources distributed across different locations

- XML (eXtensible Markup Language) has become the prime standard for data exchange on the Web and a uniform data model for integrated data.
Why not HTML? An Example

- Amazon publishes a catalog for books on sale
  - Data source: a relational database
  - Publishing: HTML pages generated from the relational database

- Customers want to query the catalog data:
  - They can only access the published Web pages (and hence need a parser)
  - They are only interested in information about books on Databases -- in SQL:
    
    ```sql
    select B
    from book B
    where B.title LIKE “Database%”
    ```
What is wrong with HTML?

<h3>Books</h3> “Databases”

<ol>
  <li><b>Database Design for Mere Mortals</b> Michael J. Hernandez<br>
      Mar 13 2003</li>
  <li><b>Beginning Database Design: From Novice to Professional</b> Clare Churcher</li>
</ol>
What is wrong with HTML?

- A minor format change to the HTML document may break the parser – and yield wrong answer to the query

  - Why? HTML tags are
    - predefined and fixed
    - describing display format rather than structure of data

- HTML is good for presentation (human friendly), but does not help automatic data extraction by means of programs (queries)
An XML solution

<books>

<book>
    <title>Database Design for Mere Mortals</title>
    <author>Michael J. Hernandez</author>
    <date>13/03/2003</date>
</book>

<book id="B2">
    <title>Beginning Database Design: From Novice to Professional</title>
    <author>Clare Churcher</author>
</book>

</books>
To make the previous student list suitable for machine consumption on the Web, it should have these characteristics:

- Be **object-like**
- Be **schemaless** (not guaranteed to conform exactly to any schema, but different objects have some commonality among themselves)
- Be **self-describing** (some schema-like information, like attribute names, is part of data itself)

Data with these characteristics are referred to as **semistructured**.
What is Self-describing Data?

- Non-self-describing (relational, object-oriented):
  - **Data part:**
    (#123, [“Students”, {[“John”, s111111111, [123,”Main St”]],
       [“Joe”, s222222222, [321, “Pine St”]] }]
  - **Schema part:**
    PersonList[ ListName: String,
                   Contents: [ Name: String,
                               Id: String,
                               Address: [Number: Integer, Street: String] ] ]
What is Self-Describing Data? cont’d

Self-describing:

- Attribute names embedded in the data itself, *but are distinguished* from values
- Doesn’t need schema to figure out what is what (but schema might be useful nonetheless)

(#12345, [ListName: “Students”,
  Contents: { [ Name: “John Doe”,
    Id: “s111111111”,
    Address: [Number: 123, Street: “Main St.”] ],
  [Name: “Joe Public”,
    Id: “s222222222”,
    Address: [Number: 321, Street: “Pine St.”] } })
Overview of XML

- Like HTML, but any number of different tags can be used (up to the document author) – extensible
- Unlike HTML, no semantics behind the tags
  - For instance, HTML’s `<table>…</table>` means: render contents as a table; in XML: doesn’t mean anything special
  - Some semantics can be specified using XML Schema (types); some using stylesheets (browser rendering)
- Unlike HTML, is intolerant to bugs
  - Browsers will render buggy HTML pages
  - XML processors are not supposed to process buggy XML documents
Example

```xml
<?xml version="1.0" ?>
<PersonList Type="Student" Date="2002-02-02">
    <Title Value="Student List" />
    <Person> 
        ... 
    </Person>
    <Person> 
        ... 
    </Person>
</PersonList>
```

- Elements are nested
- Root element contains all others
More Terminology

John is a nice fellow

```
<Person Name = "John" Id = "s111111111"

John is a nice fellow
</Person>

<Address>
    <Number>21</Number>
    <Street>Main St.</Street>
</Address>

“standalone” text, not very useful as data, non-uniform

Child of Address, Descendant of Person

Opening tag

Closing tag: What is open must be closed
Well-formed XML Documents

- Must have a **root element**
- Every **opening tag** must have matching **closing tag**
- Elements must be **properly nested**
  - `<foo><bar></foo></bar>` is a no-no
- An **attribute** name can occur *at most once* in an opening tag. If it occurs,
  - It *must have an explicitly specified value* (Boolean attrs, like in HTML, are not allowed)
  - The value *must be quoted* (with “ or ‘)
- **XML processors are not supposed to try and fix ill-formed documents** (unlike HTML browsers)
<bookstore>
    <book category="COOKING">
        <title lang="en">Everyday Italian</title>
        <author>Giada De Laurentiis</author>
        <year>2005</year>
        <price>30.00</price>
    </book>
    <book category="CHILDREN">
        <title lang="en">Harry Potter</title>
        <author>J K. Rowling</author>
        <year>2005</year>
        <price>29.99</price>
    </book>
    <book category="WEB">
        <title lang="en">Learning XML</title>
        <author>Erik T. Ray</author>
        <year>2003</year>
        <price>39.95</price>
    </book>
</bookstore>
Identifying and Referencing with Attributes

An attribute can be declared to have type:

- **ID** – unique identifier of an element
  - If attr1 & attr2 are both of type ID, then it is illegal to have
    `<something attr1="abc"> … <somethingelse attr2="abc">`
    within the same document

- **IDREF** – references a unique element with matching ID attribute
  - If attr1 has type ID and attr2 has type IDREF then we **can** have:
    `<something attr1="abc"> … <somethingelse attr2="abc">`

- **IDREFS** – a list of references, if attr1 is ID and attr2 is IDREFS, then we **can** have
  - `<something attr1="abc">…<somethingelse attr1="cde">…<someotherthing attr2="abc cde">`
Document Type Definition (DTD)

- A **DTD** is a grammar specification for an XML document.
- DTDs are optional – don’t need to be specified.
  - If specified,
    - DTD can be part of the document (at the top), or
    - it can be given as a URL.
- A document that conforms (i.e., parses) w.r.t. its DTD is said to be **valid**.
  - XML processors are **not required to check validity** even if DTD is specified.
  - But they are required to test well-formedness.
DTDs (cont’d)

- DTD specified as part of a document:
  ```xml
  <?xml version="1.0" ?>
  <!DOCTYPE Book [ … … … ]>
  <Book> … … … </Book>
  ```

- DTD specified as a standalone thing
  ```xml
  <?xml version="1.0" ?>
  <Book> … … … </Book>
  ```
DTD Components

- `<!ELEMENT elt-name (...)contents...)/>EMPTY/ANY >`
  - Element’s contents

- `<!ATTLIST elt-name attr-name CDATA/ID/IDREF/IDREFS #IMPLIED/#REQUIRED >`
  - An attr for elt
  - Type of attribute
  - Optional/mandatory

- Can define other things, like macros (called *entities* in the XML jargon)
<!DOCTYPE Report [ 
<!ELEMENT Report (Students, Classes, Courses)> 
<!ELEMENT Students (Student*)> 
<!ELEMENT Classes (Class*)> 
<!ELEMENT Courses (Course*)> 
<!ELEMENT Student (Name, Status, CrsTaken*)> 
<!ELEMENT Name (First, Last)> 
<!ELEMENT First (#PCDATA)> 
... ... ... 
<!ELEMENT CrsTaken EMPTY> 
<!ELEMENT Class (CrsCode, Semester, ClassRoster)> 
<!ELEMENT Course (CrsName)> 
... ... ... 
<!ATTLIST Report Date CDATA #IMPLIED> 
<!ATTLIST Student StudId ID #REQUIRED> 
<!ATTLIST Course CrsCode ID #REQUIRED> 
<!ATTLIST CrsTaken CrsCode IDREF #REQUIRED 
Semester CDATA #REQUIRED > 
<!ATTLIST ClassRoster Members IDREFS #IMPLIED>
<?xml version="1.0" ?>
<Report Date="2002-12-12">
  <Students>
    <Student StudId="s1111111111">
      <Name>
        <First>John</First>
        <Last>Doe</Last>
      </Name>
      <Status>U2</Status>
      <CrsTaken CrsCode="CS308" Semester="F1997" />
      <CrsTaken CrsCode="MAT123" Semester="F1997" />
    </Student>
    <Student StudId="s6666666666">
      <Name>
        <First>Joe</First>
        <Last>Public</Last>
      </Name>
      <Status>U3</Status>
      <CrsTaken CrsCode="CS308" Semester="F1994" />
      <CrsTaken CrsCode="MAT123" Semester="F1997" />
    </Student>
    <Student StudId="s987654321">
      <Name>
        <First>Bart</First>
        <Last>Simpson</Last>
      </Name>
      <Status>U4</Status>
      <CrsTaken CrsCode="CS308" Semester="F1994" />
    </Student>
  </Students>
</Report>

…… Continued … …
<Classes>
  <Class>
    <CrsCode>CS308</CrsCode> <Semester>F1994</Semester>
    <ClassRoster Members="s666666666 987654321" />
  </Class>
  <Class>
    <CrsCode>CS308</CrsCode> <Semester>F1997</Semester>
    <ClassRoster Members="s111111111" />
  </Class>
  <Class>
    <CrsCode>MAT123</CrsCode> <Semester>F1997</Semester>
    <ClassRoster Members="s111111111 s666666666" />
  </Class>
</Classes>

...... continued ....
<Courses>
    <Course CrsCode = “CS308” >
        <CrsName>Market Analysis</CrsName>
    </Course>
    <Course CrsCode = “MAT123” >
        <CrsName>Market Analysis</CrsName>
    </Course>
</Courses>
</Report>
Limitations of DTDs

- Don’t understand namespaces
- Very limited assortment of data types (just strings)
- Very weak w.r.t. consistency constraints (ID/IDREF/IDREFS only)
- Can’t express unordered contents conveniently
- All element names are global: can’t have one Name type for people and another for companies:
  
  ```
  <!ELEMENT Name (Last, First)>
  <!ELEMENT Name (#PCDATA)>
  ```

  both can’t be in the same DTD
XML Schema

- Came to rectify some of the problems with DTDs
- Advantages:
  - Integrated with namespaces
  - Many built-in types
  - User-defined types
  - Has local element names
  - Powerful key and referential constraints
- Disadvantages:
  - Unwieldy – much more complex than DTDs
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, very complicated

- **XQuery** – W3C standard.
  - Very powerful, fairly intuitive, SQL-style

- **SQL/XML** – attempt to marry SQL and XML, part of SQL:2003
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 new 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, …)
A fragment of the report document from earlier

```xml
<?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*.
- **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*:
  - / Students/Student
  - /Student
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.
  - `/` – 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).
XPath Basics cont’d

- **Current context** (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
  - ./foo/bar
  - ../abc/cde

- Expressions that don’t start with / are relative (to the current node)
XPath Basics cont’d

- **Current context** (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 children of abc e-children of the parent of the current node

- Expressions that don’t start with / are relative (to the current node)
Attributes, Text, etc.

- `/Students/Student/@StudentId`
- `/Students/Student/Name/Last/text()`
- XPath provides means to select other document components as well
Attributes, Text, etc.

- `/Students/Student/@StudentId` – returns all a-children of Student, which are e-children of Students, which are children of the root

- `/Students/Student/Name/Last/text()` – returns all t-children of Last e-children of ...

- XPaths provides means to select other document components as well
Overall Idea and Semantics

- An XPath expression is:
  \[\text{locationStep1/locationStep2/…locationStep1/locationStep2/…}\]

- **Location step:**
  \(\text{Axis::nodeSelector[predicate]}\)

- **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 = “CSC343”]

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

- The meaning of the expression $\text{locationStep1}/\text{locationStep2}/\ldots$ is the set of all document nodes obtained as follows:
  - Find all nodes reachable by $\text{locationStep1}$ from the current node
  - For each node $N$ in the result, find all nodes reachable from $N$ by $\text{locationStep2}$; take the union of all these nodes
  - For each node in the result, find all nodes reachable by $\text{locationStep3}$, 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
Overall Idea of the Semantics cont’d

- \( \text{locationStep1/\text{locationStep2}/...} \) means:
  - Find all nodes specified by \( \text{locationStep1} \)
  - For each such node N:
    - sFind all nodes specified by \( \text{locationStep2} \) using N as the current node
    - Take union
  - For each node returned by \( \text{locationStep2} \) do the same

- \( \text{locationStep} = \text{axis::node[predicate]} \)
  - Find all nodes specified by \( \text{axis::node} \)
  - Select only those that satisfy \( \text{predicate} \)
More on Navigation Primitives

- 2\textsuperscript{nd} \texttt{CrsTaken} child of 1\textsuperscript{st} \texttt{Student} child of \texttt{Students}:
  
  \texttt{/Students/Student[1]/CrsTaken[2]}

- All \texttt{last} \texttt{CourseTaken} elements within each \texttt{Student} element:
  
  \texttt{/Students/Student/CrsTaken[last( )]}
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

- The  *  wildcard:
  - *  – any element:  Student/*/text()
  - @*  – any attribute:  Students//@*
XPath Queries (selection predicates)

- Recall: Location step = `Axis::nodeSelector[predicate]`
- Predicate:
  - XPath expression = const | built-in function | XPath expression
  - XPath expression
  - built-in predicate
  - a Boolean combination thereof

- `Axis::nodeSelector[predicate] ⊆ 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.
XPath Queries – Examples

- Students who have taken CSC343:
  \[//\text{Student}[\text{CrsTaken/}@\text{CrsCode}="\text{CSC343}"]\]
  True if: “CSC343” \(\in\) //Student/CrsTaken/@CrsCode

- Complex example:
  \[//\text{Student}[\text{Status}="U3" \text{ and } \text{starts-with}(.//\text{Last}, "A") \text{ and } \text{contains}(.//@\text{CrsCode},"ESE") \text{ and } \text{not}(.//\text{Last} = .//\text{First})]\]

- Aggregation: \(\text{sum}( ), \text{count}( )\)
  \[//\text{Student}[\text{sum}(.//@\text{Grade}) \text{ div } \text{count}(.//@\text{Grade}) > 3.5]\]
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[Semeester=“F1990”]

  - union lets us define *heterogeneous* collections of nodes
XQuery – XML Query Language

- Integrates XPath with earlier proposed query languages: XQL, XML-QL
- SQL-style, not functional-style
- 2007: XQuery 1.0
XQuery Basics: FLOWR Expression

- General structure:

  FOR variable declarations
  LET variable declarations
  WHERE condition
  ORDER BY list
  RETURN document
XQuery Basics: FLOWR Expression

- Example:
  (: students who took MAT123 :)  
  FOR $t$ IN doc("transcript.xml")//Transcript  
  WHERE $t$/CrsTaken/@CrsCode = "MAT123"  
  RETURN $t$/Student

- Result:
  <Student StudId="111111111" Name="John Doe" />  
  <Student StudId="123454321" Name="Joe Blow" />
<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 … …
<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>
XQuery Basics (cont’d)

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

```xml
<StudentList>
{
  FOR $t IN doc("transcript.xml")//Transcript
  WHERE $t/CrsTaken/@CrsCode = "MAT123"
  RETURN $t/Student
}
</StudentList>
```
Reconstruct lists of students taking each class using the Transcript records:

```xquery
FOR $c$ IN doc("transcript.xml")//CrsTaken
ORDER BY $c/@CrsCode
RETURN
<ClassRoster>
{
    FOR $t$ IN doc("transcript.xml")//Transcript
    WHERE $t/CrsTaken[@CrsCode = $c/@CrsCode and @Semester = $c/@Semester]
    ORDER BY $t/student/@studId
    RETURN $t/Student
}
</ClassRoster>
```

Query inside

`RETURN`
Output elements have the form:

```xml
<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
  ```xml
  <CrsTaken  CrsCode="CS305"  Semester="F1995"  Grade="A" />
  ```
- and once when `$c$` is bound to
  ```xml
  <CrsTaken  CrsCode="CS305"  Semester="F1995"  Grade="C" />
  ```

**Note:** grades are different – distinct-values() won’t eliminate transcript records that refer to same class!
Solution: instead of
for $c$ in doc("transcript.xml")//CrsTaken
  use
for $c$ in doc("classes.xml")//Class

- 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
Document Restructuring (cont’d)

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

```
FOR $c IN doc("classes.xml")//Class
WHERE doc("transcripts.xml")//CrsTaken[@CrsCode = $c/@CrsCode
  and @Semester = $c/@Semester]
ORDER BY $c/@CrsCode
RETURN

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

</ClassRoster>
```

Test that classes aren’t empty.
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
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
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 $B$:
  - $A/v$, $B/x$
  - $A/v$, $B/y$
  - $A/v$, $B/z$
  - $A/w$, $B/x$
  - $A/w$, $B/y$
  - $A/w$, $B/z$
XQuery Semantics cont’d

- **Step 2**: filter the bindings via the WHERE clause
  - Use each tuple binding to substitute its components 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
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
Grouping and Aggregation

- Does not use separate grouping operator
- Uses built-in aggregate functions count, avg, sum, etc. (some borrowed from XPath)
Produce a list of students along with the number of courses each student took:

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

The grouping effect is achieved because $c$ is bound to a new set of nodes for each binding of $t$
Quantification in XQuery

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

**Example:**

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

"Almost" equivalent to:

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

Not equivalent, if students can take same course twice!
Implicit Quantification

- Note: in SQL, variables that occur in FROM, but not SELECT are implicitly quantified with $\exists$
- 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
  - 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*
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 fn:doc(classes.xml)//Class
  LET $g := \{\text{ Transcript records that correspond to class }$c$\}
  FOR $t$ IN fn:doc(“transcript.xml”)//Transcript
  WHERE $t$/CrsTaken/@Semester = $c$/@Semester
  AND $t$/CrsTaken/@CrsCode = $c$/@CrsCode
  RETURN $t$

  WHERE EVERY $\text{str} \text{ IN } \text{g} \text{ SATISFINES}
  NOT fn:empty($\text{str}[\text{CrsTaken/@CrsCode=“MAT123”}]$)

  RETURN $c$ ORDER BY $c$/@CrsCode
Count the number of e-children recursively:

```
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/*) }
```
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, …