Query Languages for XML

XPath
Xquery

Slides from the textbook webpage:
http://infolab.stanford.edu/~ullman/dscb.html
Storage of XML Data

XML data can be stored in

- Non-relational data stores
  - Flat files
    - Natural for storing XML
    - Limitations (no concurrency, no recovery, …)
  - XML database
    - Database built specifically for storing XML data, supporting DOM model and declarative querying
    - Currently no commercial-grade systems

- Relational databases
  - Data must be translated into relational form.
  - Advantage: mature database systems.
  - Disadvantages: overhead of translating data and queries.
Storage of XML in Relational Databases

Alternatives:
- String Representation
- Tree Representation
- Map to relations
There are two standard application program interfaces to XML data:

- **SAX** (Simple API for XML)
  - Based on parser model, user provides event handlers for parsing events.

- **DOM** (Document Object Model)
  - XML data is parsed into a tree representation.
  - Variety of functions provided for traversing the DOM tree.
  - E.g.: Java DOM API provides Node class with methods `getParentNode()`, `getFirstChild()`, `getNextSibling()`, `getAttribute()`, `getData()` (for text node) `getElementsByTagName(...)`, ...
Corresponding to the fundamental “relation” of the relational model is: \textit{sequence of items}.

An \textit{item} is either:

1. A primitive value, e.g., integer or string.
2. A \textit{node} (defined next).
Principal Kinds of Nodes


2. *Elements* are pieces of a document consisting of some opening tag, its matching closing tag (if any), and everything in between.

3. *Attributes* names that are given values inside opening tags.
Document Nodes

- Formed by `doc(URL)` or `document(URL)`.
- **Example:** `doc(/usr/class/cs475/bars.xml)`
- All XPath (and XQuery) queries refer to a `doc` node, either explicitly or implicitly.
  - **Example:** key definitions in XML Schema have XPath expressions that refer to the document described by the schema.
<!DOCTYPE BARS [ 
  <!ELEMENT BARS (BAR*, BEER*)> 
  <!ELEMENT BAR (PRICE+)> 
    <!ATTLIST BAR name ID #REQUIRED> 
  <!ELEMENT PRICE (#PCDATA)> 
    <!ATTLIST PRICE theBeer IDREF #REQUIRED> 
  <!ELEMENT BEER EMPTY> 
    <!ATTLIST BEER name ID #REQUIRED> 
    <!ATTLIST BEER soldBy IDREFS #IMPLIED> 
]>
An element node

<BARS>
  <BAR name = "JoesBar">
    <PRICE theBeer = "Bud">2.50</PRICE>
    <PRICE theBeer = "Miller">3.00</PRICE>
  </BAR>
</BARS>

An attribute node

<BEER name = "Bud" soldBy = "JoesBar SuesBar ... ">
  SuesBar ... 
</BEER>

Document node is all of this, plus the header ( <? xml version... > ).
Nodes as Semistructured Data

```xml
<bars.xml>
  <BARS>
    <BAR name="JoesBar">
      <PRICE theBeer="Bud" value="2.50"/>
    </BAR>
    <BEER name="Bud" SoldBy="...">
      <PRICE theBeer="Miller" value="3.00"/>
    </BEER>
  </BARS>
</bars.xml>
```
Paths in XML Documents

- XPath is a language for describing paths in XML documents.
- The result of the described path is a sequence of items.
Path Expressions

❖ Simple path expressions are sequences of slashes (/) and tags, starting with /.
  ❖ Example: /BARS/BAR/PRICE

❖ Construct the result by starting with just the doc node and processing each tag from the left.
Evaluating a Path Expression

- Assume the first tag is the root.
  - Processing the doc node by this tag results in a sequence consisting of only the root element.

- Suppose we have a sequence of items, and the next tag is $X$.
  - For each item that is an element node, replace the element by the subelements with tag $X$. 
Example: /BARS

<BARS>
  <BAR name = "JoesBar">
    <PRICE theBeer = "Bud">2.50</PRICE>
    <PRICE theBeer = "Miller">3.00</PRICE>
  </BAR> ...
  <BEER name = "Bud" soldBy = "JoesBar SuesBar ... "/>
  ...
</BARS>
Example: /BARS/BAR

<BARS>
  <BAR name = "JoesBar">
    <PRICE theBeer ="Bud">2.50</PRICE>
    <PRICE theBeer = "Miller">3.00</PRICE>
  </BAR> ...
  <BEER name = "Bud" soldBy = "JoesBar SuesBar ..."/>
  ...
</BARS>

This BAR element followed by all the other BAR elements
Example: /BARS/BAR/PRICE

<BARS>
  <BAR name = "JoesBar">
    <PRICE theBeer ="Bud">2.50</PRICE>
    <PRICE theBeer = "Miller">3.00</PRICE>
  </BAR> ...
  <BEER name = "Bud" soldBy = "JoesBar SuesBar ..."/>
</BARS>

These PRICE elements followed by the PRICE elements of all the other bars.
Attributes in Paths

◆ Instead of going to subelements with a given tag, you can go to an attribute of the elements you already have.

◆ An attribute is indicated by putting @ in front of its name.
Example:

/BARS/BAR/PRICE/@theBeer

<BARS>

<BAR name = "JoesBar">

<PRICE theBeer = "Bud">2.50</PRICE>
<PRICE theBeer = "Miller">3.00</PRICE>

</BAR> ...

<BEER name = "Bud" soldBy = "JoesBar SuesBar ..."/> ...

</BARS>

These attributes contribute "Bud" "Miller" to the result, followed by other theBeer values.
Remember: Item Sequences

- Until now, all item sequences have been sequences of elements.
- When a path expression ends in an attribute, the result is typically a sequence of values of primitive type, such as strings in the previous example.
Paths that Begin Anywhere

◆ If the path starts from the document node and begins with //X, then the first step can begin at the root or any subelement of the root, as long as the tag is X.
Example: //PRICE

<BARS>
  <BAR name = "JoesBar">
    <PRICE theBeer = "Bud">2.50</PRICE>
    <PRICE theBeer = "Miller">3.00</PRICE>
  </BAR> ...
  <BEER name = "Bud" soldBy = "JoesBar SuesBar ...">
    ... 
  </BEER>
</BARS>

These PRICE elements and any other PRICE elements in the entire document
Wild-Card *

◆ A star (*) in place of a tag represents any one tag.

◆ Example: /*/*/PRICE represents all price objects at the third level of nesting.
Example: /BARS/*

This BAR element, all other BAR elements, the BEER element, all other BEER elements

<BARS>
  <BAR name = "JoesBar">
    <PRICE theBeer = "Bud">2.50</PRICE>
    <PRICE theBeer = "Miller">3.00</PRICE>
  </BAR>
  ...
</BARS>

<BARS>
  <BEER name = "Bud" soldBy = "JoesBar SuesBar ... "> ...
</BARS>
Selection Conditions

◆ A condition inside [...] may follow a tag.
◆ If so, then only paths that have that tag and also satisfy the condition are included in the result of a path expression.
Example: Selection Condition

◆ /BARS/BAR/PRICE[< 2.75]

The condition that the PRICE be < $2.75 makes this price but not the Miller price part of the result.
Example: Attribute in Selection

◆ /BARS/BAR/PRICE[@theBeer = "Miller"]

<BARS>
  <BAR name = "JoesBar">
    <PRICE theBeer = "Bud">2.50</PRICE>
    <PRICE theBeer = "Miller">3.00</PRICE>
  </BAR> …
</BARS>

Now, this PRICE element is selected, along with any other prices for Miller.
Axes

- In general, path expressions allow us to start at the root and execute steps to find a sequence of nodes at each step.
- At each step, we may follow any one of several axes.
- The default axis is child:: --- go to all the children of the current set of nodes.
Example: Axes

◆ /BARS/BEER is really shorthand for /BARS/child::BEER.
◆ @ is really shorthand for the attribute:: axis.
  ▪ Thus, /BARS/BEER[@name = "Bud"] is shorthand for /BARS/BEER[attribute::name = "Bud"]
More Axes

◆ Some other useful axes are:

1. **parent::** = parent(s) of the current node(s).

2. **descendant-or-self::** = the current node(s) and all descendants.
   - Note: // is really shorthand for this axis.

3. **ancestor::**, **ancestor-or-self**, etc.

4. **self** (the dot).
## XPath Syntax

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>users</td>
<td>Selects all the child nodes of the <strong>users</strong> element</td>
</tr>
<tr>
<td>/users</td>
<td>Selects the root element <strong>users</strong></td>
</tr>
<tr>
<td>users/user</td>
<td>Selects all <strong>user</strong> elements that are children of <strong>users</strong></td>
</tr>
<tr>
<td>//users</td>
<td>Selects all <strong>users</strong> elements no matter where they are in the document</td>
</tr>
<tr>
<td>users//user</td>
<td>Selects all <strong>user</strong> elements that are descendant of the <strong>users</strong> element, no matter where they are under the <strong>users</strong> element</td>
</tr>
</tbody>
</table>
XPath Injection (1/2)

- Scenario: authentication system which performs XPath query

**VB:** Dim FindUserXPath as String
FindUserXPath = "//Users/user[username/text()='' & Request("Username") & '' And password/text()='' & Request("Password") & '']"

**C#:**
String FindUserXPath;
FindUserXPath = "//Users/user[username/text()='' + Request("Username") + '' And password/text()='' + Request("Password") + '']"

- This is a standard authentication query.

Username = user
Password = password
XPath query becomes: //users/user[username/text()=‘user’ and password/text()=‘password’]

Avoid the dangers of XPath injection
XPath Injection (2/2)

◆ In this case, injection is possible in the Username variable. The same attack logic of SQL injection can be applied for XPath.

Username = user' or '1' = '1
Password = password
XPath query becomes: //users/user[username/text()=‘user’or ‘1’ = ‘1’ and password/text()=‘password’]

◆ In this case, only the first part of the XPath needs to be true.
◆ The password part becomes irrelevant, and the UserName part will match ALL users because of the "1=1" condition.
◆ This injection will allow the attacker to bypass the authentication system.
◆ Note that the big difference between XML files and SQL databases is the lack of access control.
◆ XPath does not have any restrictions when querying the XML file. Therefore it is possible to retrieve data from the entire document.
Summary

- What is XPath?
- XPath Syntax
- XPath Injection
Exercise

We want to export this data into an XML file. Write a DTD describing the following structure for the XML file:
- there is one root element called stores
  - the stores element contains a sequence of store sub elements, one for each store in the database
  - each store element contains one name, and one phone subelement, and a sequence of product subelements, one for each product that the store sells. Also, it has an attribute sid of type ID.
  - each product element contains one name, one price, one description, and one markup element, plus an attribute pid of type ID.
<!DOCTYPE CommodityData [ 
<!ELEMENT stores (store*)> 
<!ELEMENT store (name, phone, product+)>
<!ELEMENT product (name, price, description, markup)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT price (#PCDATA)>
<!ELEMENT description (#PCDATA)>
<!ELEMENT phone (#PCDATA)>
<!ELEMENT markup (#PCDATA)>
<!ATTLIST product 
    pid ID #REQUIRED
]>
<!ATTLIST store 
    sid ID #REQUIRED
]>
]>
2. Write the XML document obtained by exporting the database Commodity into the DTD.

```xml
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<!DOCTYPE CommodityData SYSTEM "stores.dtd">
<stores>
    <store sid="s282">
        <name>Wiz</name>
        <phone>555-1234</phone>
        <product pid="233">
            <name>gizmo plus</name>
            <price>99.99</price>
            <description>more features</description>
            <markup>25%</markup>
        </product>
    </store>
    <store sid="s521">
        <name>Econo-Wiz</name>
        <phone>555-6543</phone>
        <product pid="323">
            <name>gizmo</name>
            <price>22.99</price>
            <description>great</description>
            <markup>10%</markup>
        </product>
        <product pid="233">
            <name>gizmo plus</name>
            <price>99.99</price>
            <description>more features</description>
            <markup>15%</markup>
        </product>
    </store>
</stores>
```

3. XPath Queries
   1) /stores/store
   2) /stores/store/@sid
   3) /stores/store [@sid = "s282"]
   4) /stores/store/name
   5) /stores/store/product
   6) /stores/store/product/@pid
   7) /stores/store/product [@pid = "323"]
   8) /stores/store/product [@pid = "233"]
   9) //product

4. Which stores sell some products with a price higher than 50? List their IDs.

```
/stores/store[./product/price>50]/@sid
```
Review

- What is XPath?
- XPath Syntax
- XPath Injection
XQuery Motivation

- XPath expressivity insufficient
  - no join queries
  - no changes to the XML structure possible
  - no quantifiers
  - no aggregation and functions
XQuery

- XQuery extends XPath to a query language that has power similar to SQL.
- Uses the same sequence-of-items data model.
- XQuery is an expression language.
  - Like relational algebra --- any XQuery expression can be an argument of any other XQuery expression.
More About Item Sequences

- XQuery will sometimes form sequences of sequences.
- All sequences are flattened.
- **Example**: $(1 \ 2 \ (3 \ 4)) = (1 \ 2 \ 3 \ 4)$. 

Empty sequence
FLWR ("Flower") Expressions

XQuery uses XPath to express more complex queries.

FOR ...
LET...
WHERE...
RETURN...
FLWR Expressions

1. One or more for and/or let clauses.
2. Then an optional where clause.
3. A return clause.
Semantics of FLWR Expressions

- Each `for` creates a loop.
  - `let` produces only a local definition.

- At each iteration of the nested loops, if any, evaluate the `where` clause.

- If the `where` clause returns TRUE, invoke the `return` clause, and append its value to the output.
FOR Clauses

for <variable> in <expression>, . . .

- Variables begin with $.
- A for-variable takes on each item in the sequence denoted by the expression, in turn.
- Whatever follows this for is executed once for each value of the variable.
Example: FOR

for $beer in document("bars.xml")/BARS/BEER/@name
return

<BEERNAME>{$beer}</BEERNAME>

$beer ranges over the name attributes of all beers in our example document.

Result is a sequence of BEERNAME elements:

<BEERNAME>Bud</BEERNAME>
<BEERNAME>Miller</BEERNAME> . . .
Use of Braces

When a variable name like $x$, or an expression, could be text, we need to surround it by braces to avoid having it interpreted literally.

- **Example**: `<A>$x</A>` is an A-element with value ""$x"", just like `<A.foo</A>` is an A-element with "foo" as value.
Use of Braces --- (2)

- But `return $x` is unambiguous.
- You cannot return an untagged string without quoting it, as `return "$x"`. 
LET Clauses

let <variable> := <expression>, . . .

◆ Value of the variable becomes the sequence of items defined by the expression.

◆ Note let does not cause iteration; for does.
Example: LET

let $d := document("bars.xml")
let $beers := $d/BARS/BEER/@name
return

  <BEERNames> {$beers} </BEERNames>

✦ Returns one element with all the names of the beers, like:

  <BEERNames>Bud Miller ...</BEERNames>
Order-By Clauses

◆ FLWR is really FLWOR: an order-by clause can precede the return.
◆ Form: order by <expression>
  - With optional ascending or descending.
◆ The expression is evaluated for each assignment to variables.
◆ Determines placement in output sequence.
Example: Order-By

❖ List all prices for Bud, lowest first.

let $d := \text{document(“bars.xml”)}$

for $p$ in $d$/BARS/BAR/PRICE[@theBeer=“Bud”]

order by $p$

return $p$

Each binding is evaluated for the output. The result is a sequence of PRICE elements.

Order those bindings by the values inside the elements.

Generates bindings for $p$ to PRICE elements.
Predicates

- Normally, conditions imply existential quantification.
- Example: /BARS/BAR[@name] means “all the bars that have a name.”
- Example: /BARS/BEER[@soldAt = "JoesBar"] gives the set of beers that are sold at Joe’s Bar.
Example: Comparisons

◆ Let us produce the PRICE elements (from all bars) for the beers that are sold by Joe’s Bar.

◆ The output will be BBP elements with the names of the bar and beer as attributes and the price element as a subelement.
Strategy

1. Create a triple for-loop, with variables ranging over all BEER elements, all BAR elements, and all PRICE elements within those BAR elements.

2. Check that the beer is sold at Joe’s Bar and that the name of the beer and theBeer in the PRICE element match.

3. Construct the output element.
The Query

let $bars = doc("bars.xml")/BARS
for $beer in $bars/BEER
for $bar in $bars/BAR
for $price in $bar/PRICE
where $beer/@soldAt = "JoesBar" and $price/@theBeer = $beer/@name
return <BBP bar = {$bar/@name} beer = {$beer/@name}>{$price}</BBP>
Strict Comparisons

To require that the things being compared are sequences of only one element, use the Fortran comparison operators:

- eq, ne, lt, le, gt, ge.

**Example:** $\text{beer/}@\text{soldAt eq "JoesBar"}$ is true only if Joe’s is the only bar selling the beer.
Comparison of Elements and Values

◆ When an element is compared to a primitive value, the element is treated as its value, if that value is atomic.

◆ Example:

```
/BARS/BAR[@name="JoesBar"]/PRICE[@theBeer="Bud"] eq "2.50"
```

is true if Joe charges $2.50 for Bud.
Comparison of Two Elements

◆ It is insufficient that two elements look alike.
◆ Example:

/BARS/BAR[@name="JoesBar"]/
PRICE[@theBeer="Bud"] eq
/BARS/BAR[@name="SuesBar"]/
PRICE[@theBeer="Bud"]

is false, even if Joe and Sue charge the same for Bud.
Comparison of Elements – (2)

◆ For elements to be equal, they must be the same, physically, in the implied document.

◆ **Subtlety**: elements are really pointers to sections of particular documents, not the text strings appearing in the section.
Getting Data From Elements

- Suppose we want to compare the values of elements, rather than their location in documents.
- To extract just the value (e.g., the price itself) from an element $E$, use $\text{data}(E)$. 
Example: data()

Suppose we want to modify the return for “find the prices of beers at bars that sell a beer Joe sells” to produce an empty BBP element with price as one of its attributes.
Previous Query

let $bars = doc("bars.xml")/BARS
for $beer in $bars/BEER
for $bar in $bars/BAR
for $price in $bar/PRICE
where $beer/@soldAt = "JoesBar" and
   $price/@theBeer = $beer/@name
return <BBP bar = {$bar/@name} beer = {$beer/@name}>{$price}</BBP>
Modified Query

let $bars = doc("bars.xml")/BARS
for $beer in $bars/BEER
for $bar in $bars/BAR
for $price in $bar/PRICE
where $beer/@soldAt = "JoesBar" and
    $price/@theBeer = $beer/@name
return <BBP bar = {$bar/@name} beer =
    {$beer/@name} price = {data($price)} />
Eliminating Duplicates

◆ Use function `distinct-values` applied to a sequence.

◆ Subtlety: this function strips tags away from elements and compares the string values.
  ♦ But it doesn’t restore the tags in the result.
Example: All the Distinct Prices

```xquery
return distinct-values(
  let $bars = doc("bars.xml")
  return $bars/BARS/BAR/PRICE
)
```

Remember: XQuery is an expression language. A query can appear any place a value can.
Exercise

We want to export this data into an XML file. Write a DTD describing the following structure for the XML file:
- there is one root element called **stores**
  - the **stores** element contains a sequence of **store** sub elements, one for each store in the database
    - each **store** element contains one **name**, and one **phone** subelement, and a sequence of **product** subelements, one for each product that the store sells. Also, it has an attribute **sid** of type ID.
    - each **product** element contains one **name**, one **price**, one **description**, and one **markup** element, plus an attribute **pid** of type ID.
<!DOCTYPE CommodityData [
  <!ELEMENT stores (store*)>
  <!ELEMENT store (name, phone, product+)>
  <!ELEMENT product (name, price, description, markup)>
  <!ELEMENT name (#PCDATA)>
  <!ELEMENT price (#PCDATA)>
  <!ELEMENT description (#PCDATA)>
  <!ELEMENT phone (#PCDATA)>
  <!ELEMENT markup (#PCDATA)>
  <!ATTLIST product pid ID #REQUIRED>
  >
  <!ATTLIST store sid ID #REQUIRED>
  >
]>
1. Which stores sell some products with a price higher than 50? List their IDs.

2. Which stores (except “Wiz”) sell the same products as store “Wiz”? List their names.
Solutions

1. Let \( d = \text{document(“stores.xml”)} \)
   
   FOR \( x \) IN \( d//\text{store[./product/price>50]}/@sid \)
   
   RETURN \{\$x\}

2. FOR \( x \) IN \( \text{document(“stores.xml”)//store[./name = “Wiz”]}/\text{product} \)
   
   FOR \( y \) IN \( \text{document(“stores.xml”)//store[./name<>”Wiz”]} \)
   
   WHERE \( x = y/\text{product} \)
   
   RETURN \{\$y/name\}
XQuery Motivation

- XPath expressivity insufficient
  - no join queries
  - no changes to the XML structure possible
  - no quantifiers
  - no aggregation and functions
FLWR ("Flower") Expressions

◆ XQuery uses XPath to express more complex queries.

FOR ...
LET...
WHERE...
RETURN...
XQuery Variables

- **FOR $x in expr**  -- binds $x to each value in the list expr

- **LET $x := expr**  -- binds $x to the entire list expr
  - Useful for common subexpressions and for aggregations
Sample Data for Queries

```xml
<bib>
  <book price="75">
    <publisher> Addison-Wesley </publisher>
    <author> Serge Abiteboul </author>
    <author> Rick Hull </author>
    <author> Victor Vianu </author>
    <title> Foundations of Databases </title>
    <year> 1995 </year>
  </book>
  <book price="95">
    <publisher> Freeman </publisher>
    <author> Jeffrey D. Ullman </author>
    <title> Principles of Database and Knowledge Base Systems </title>
    <year> 1998 </year>
  </book>
</bib>
```
Basic FLWR

Find all book titles published after 1995:

FOR $x$ IN document("bib.xml")/bib/book
WHERE $x/year > 1995
RETURN $x/title

Result:
<title> Principles of Database and Knowledge Base Systems </title>
Result Structuring

Find all book titles and the year when they were published:

```
FOR $x IN document("bib.xml")/ bib/book
RETURN <answer>
    {$x/title}
    {$x/year}
</answer>
```
Result Structuring

- Notice the use of "{" and "}" 
- What is the result without them?

```
FOR $x$ IN document("bib.xml")/bib/book
RETURN <answer>
  $x/title
  $x/year
</answer>
```
FOR v.s. LET

FOR $x$ IN
document("bib.xml")[bib/book]
RETURN <result> {$x}$ </result>

LET $x$:=
document("bib.xml")[bib/book]
RETURN <result> {$x}$ </result>

Returns:
<result> <book>...</book></result>
<result> <book>...</book></result>
<result> <book>...</book></result>
...

Returns:
<result> <book>...</book></result>
<book>...</book></result>
<book>...</book></result>
...
</result>
Aggregates

Find all books with more than 3 authors:

```
FOR $x$ IN document("bib.xml")/bib/book
WHERE count($x/author)>3
RETURN $x$
```

count = a function that counts
avg = computes the average
sum = computes the sum
distinct-values = eliminates duplicates
LET

Find all publishers that published more than 100 books:

```
FOR $p$ IN distinct-values(//publisher)
LET $b$ := /db/book[./publisher = $p$]
WHERE count($b$) > 100
RETURN <publisher> {$p$} </publisher>
```

$b$ is a collection of elements, not a single element
Branching Expressions

◆ if \((E_1)\) then \(E_2\) else \(E_3\) is evaluated by:
  - Compute the **effective boolean value** of \(E_1\).
  - If true, the result is \(E_2\); else the result is \(E_3\).

◆ **Example**: the PRICE subelements of $bar, provided that bar is Joe’s.

if($bar/@name eq "JoesBar")
then $bar/PRICE else ()
Effective Boolean Values

◆ The *effective boolean value* (EBV) of an expression is:

1. The actual value if the expression is of type boolean.
2. FALSE if the expression evaluates to 0, """" [the empty string], or () [the empty sequence].
3. TRUE otherwise.
EBV Examples

1. @name="JoesBar" has EBV TRUE or FALSE, depending on whether the name attribute is "JoesBar".

2. /BARS/BAR[@name="GoldenRail"] has EBV TRUE if some bar is named the Golden Rail, and FALSE if there is no such bar.
Boolean Operators

- $E_1$ and $E_2$, $E_1$ or $E_2$, not($E$), apply to any expressions.
- Take EBV’s of the expressions first.
- **Example**: not(3 eq 5 or 0) has value TRUE.
- Also: true() and false() are functions that return values TRUE and FALSE.
Quantifier Expressions

some $x$ in $E_1$ satisfies $E_2$

1. Evaluate the sequence $E_1$.
2. Let $x$ (any variable) be each item in the sequence, and evaluate $E_2$.
3. Return TRUE if $E_2$ has EBV TRUE for at least one $x$.

◆ Analogously:

every $x$ in $E_1$ satisfies $E_2$
Example: Some

The bars that sell at least one beer for less than $2.

```xml
for $bar in
doc("bars.xml")/BARS/BAR
where some $p in $bar/PRICE satisfies $p < 2.00
return $bar/@name
```
Example: Every

The bars that sell no beer for more than $5.

for $bar in doc("bars.xml")/BARS/BAR
where every $p in $bar/PRICE satisfies $p <= 5.00
return $bar/@name
Document Order

◆ Comparison by document order: << and >>.

◆ Example: $d/BARS/BEER[@name=“Bud”] << $d/BARS/BEER[@name=“Miller”] is true iff the Bud element appears before the Miller element in the document $d.$
Set Operators

- `union`, `intersect`, `except` operate on sequences of nodes.
  - Meanings analogous to SQL.
  - Result eliminates duplicates.
  - Result appears in document order.
XQuery Injection

- XQuery Injection is a variant of the classic SQL injection attack against the XML XQuery Language.

- XQuery injection can be used to enumerate elements on the victim's environment, inject commands to the local host, or execute queries to remote files and data sources.
<?xml version="1.0" encoding="ISO-8859-1"?>

<userlist>
  <user category="group1">  
    <uname>jpublic</uname> 
    <fname>john</fname> 
    <lname>public</lname> 
    <status>good</status>  </user>
  <user category="admin">  
    <uname>jdoe</uname> 
    <fname>john</fname> 
    <lname>doe</lname> 
    <status>good</status>  </user>
  <user category="group2">  
    <uname>mjane</uname> 
    <fname>mary</fname> 
    <lname>jane</lname> 
    <status>good</status>  </user>
  <user category="group1">  
    <uname>anormal</uname> 
    <fname>abby</fname> 
    <lname>normal</lname> 
    <status>revoked</status>  </user>
</userlist>

doc("users.xml")/userlist/user[uname = "something" or ""=""]
Summary

◆ Xquery
◆ Assignment 5 is posted.
◆ Next Topic: OLAP
1. Which stores sell some products with a price higher than 50? List their IDs.

2. Which stores (except “Wiz”) sell the same products as store “Wiz”? List their names.

3. Write an XQuery query that returns the names and prices of products that are sold in all stores with a markup no lower than 15%.
3. FOR $p$ IN distinct(document("stores.xml")//product) WHERE EVERY $m$ IN (document("stores.xml")//product[./name = $p/name]/markup) SATISFIES $m >= 15\%$ RETURN <result>{$p/name} {$p/price}</result>