XPath and XQuery

Introduction to Databases
CompSci 316 Fall 2016

Announcements (Tue. Oct. 25)
• Homework #3 due in two weeks
• Project milestone #1 feedback emailed
  • Milestone #2 due in 2½ weeks

Query languages for XML
• XPath
  • Path expressions with conditions
    • Building block of other standards (XQuery, XSLT, XLink, XPointer, etc.)
• XQuery
  • XPath + full-fledged SQL-like query language
• XSLT
  • XPath + transformation templates
Example DTD and XML

```xml
<?xml version="1.0"?>
<!DOCTYPE bibliography [
<!ELEMENT bibliography (book+)>
<!ELEMENT book (title, author*, publisher?, year?, section*)>
<!ATTLIST book ISBN CDATA #REQUIRED>
<!ATTLIST book price CDATA #IMPLIED>
<!ELEMENT title (#PCDATA)>
<!ELEMENT author (#PCDATA)>
<!ELEMENT publisher (#PCDATA)>
<!ELEMENT year (#PCDATA)>
<!ELEMENT section (#PCDATA)> ]>
</DOCTYPE>

<bibliography>
  <book ISBN="ISBN-10" price="80.00">
    <title>Foundations of Databases</title>
    <author>Abiteboul</author>
    <author>Hull</author>
    <author>Vianu</author>
    <publisher>Addison Wesley</publisher>
    <year>1995</year>
    <section>…</section>…
  </book>
…
</bibliography>
```

XPath

- XPath specifies path expressions that match XML data by navigating down (and occasionally up and across) the tree
- Example
  - Query: /bibliography/book/author
    - Like a file system path, except there can be multiple “subdirectories” with the same name
  - Result: all author elements reachable from root via the path /bibliography/book/author

Basic XPath constructs

- `//` separator between steps in a path
- `@name` matches any attribute
- `@*` matches any attribute
- `/@name` matches the attribute with this name
- `@*` matches any attribute
- `@` matches any attribute
- `@name` matches the attribute with this name
- `name` matches any child element with this tag name
- `*` matches any child element
- `/.` matches the current element
- `..` matches the parent element
Simple XPath examples

- All book titles
  `/bibliography/book/title`
- All book ISBN numbers
  `/bibliography/book/@ISBN`
- All title elements, anywhere in the document
  `//@title`
- All section titles, anywhere in the document
  `//@section/title`
- Authors of bibliographical entries (suppose there are articles, reports, etc. in addition to books)
  `/bibliography/*/author`

Predicates in path expressions

[condition] matches the “current” element if condition evaluates to true on the current element

- Books with price lower than $50
  `/bibliography/book[@price<50]`
  - XPath will automatically convert the price string to a numeric value for comparison
- Books with author “Abiteboul”
  `/bibliography/book[author='Abiteboul']`
- Books with a publisher child element
  `/bibliography/book[publisher]`
- Prices of books authored by “Abiteboul”
  `/bibliography/book[author='Abiteboul']/@price`

More complex predicates

Predicates can use and, or, and not

- Books with price between $40 and $50
  `/bibliography/book[40<=@price and @price<=50]`
- Books authored by “Abiteboul” or those with price no lower than $50
  `/bibliography/book[author='Abiteboul' or @price>=50]`
  - Any difference between these two queries?
Predicates involving node-sets

/bibliography/book[author='Abiteboul']
• There may be multiple authors, so author in general returns a node-set (in XPath terminology)
• The predicate evaluates to true as long as it evaluates true for at least one node in the node-set, i.e., at least one author is “Abiteboul”
• Tricky query
  /bibliography/book[author='Abiteboul' and author!='Abiteboul']
  • Will it return any books?

XPath operators and functions

Frequently used in conditions:
 x + y, x − y, x * y, x div y, x mod y
contains(x, y) true if string x contains string y
count(node-set) counts the number nodes in node-set
position() returns the “context position” (roughly, the position of the current node in the node-set containing it)
last() returns the “context size” (roughly, the size of the node-set containing the current node)
name() returns the tag name of the current element

More XPath examples

• All elements whose tag names contain “section” (e.g., “subsection”)
  //*[contains(name(), 'section')]
• Title of the first section in each book
  /bibliography/book/section[position()==1]/title
  • A shorthand: /bibliography/book/section[1]/title
• Title of the last section in each book
  /bibliography/book/section[position()==last()]/title
• Books with fewer than 10 sections
  /bibliography/book[count(section)<10]
• All elements whose parent’s tag name is not “book”
  //*[name()='book']/*
A tricky example

• Suppose for a moment that price is a child element of book, and there may be multiple prices per book.
  • Wrong answer:
    `/bibliography/book [price >= 20 and price <= 50]`
  • Correct answer:
    `/bibliography/book [price[. >= 20 and . <= 50]]`

De-referencing IDREF’s

`id(identifier)` returns the element with identifier

• Suppose that books can reference other books
  `<section><title>Introduction</title><p>XML is a hot topic these days; see <bookref ISBN='ISBN-10'/> for more details...</p></section>`
  • Find all references to books written by “Abiteboul” in the book with “ISBN-10”
  Or simply:
    `/bookref[id(@ISBN)/author='Abiteboul']`

General XPath location steps

• Technically, each XPath query consists of a series of location steps separated by `/`
• Each location step consists of
  • An axis: one of self, attribute, parent, child, ancestor, ancestor-or-self, descendant, descendant-or-self, following, following-sibling, preceding, preceding-sibling, and namespace
  • A node-test: either a name test (e.g., book, section, #) or a type test (e.g., text(), node(), comment()), separated from the axis by `::`
  • Zero or more predicates (or conditions) enclosed in square brackets

¹These reverse axes produce result node sets in reverse document order; others (forward axes) produce node sets in document order.
Example of verbose syntax

Verbose \((\text{axis}, \text{node test}, \text{predicate})\):
/child::bibliography
/descendant-or-self::node()
/child::title

Abbreviated:
• child is the default axis
• // stands for //descendant-or-self::node() /

Some technical details on evaluation

Given a context node, evaluate a location path as follows:
1. Start with node-set \(N = \{\text{context node}\}\)
2. For each location step, from left to right:
   • \(U \leftarrow \emptyset\)
   • For each node \(n\) in \(N\):
     • Using \(n\) as the context node, compute a node-set \(N'\) from the axis
       and the node test
     • Each predicate in turn filters \(N'\), in order
       • For each node \(n'\) in \(N'\), evaluate predicate with the following context:
         • Context node is \(n'\)
         • Context size is the number of nodes in \(N'\)
         • Context position is the position of \(n'\) within \(N'\)
     • \(U \leftarrow U \cup N'\)
     • \(N \leftarrow U\)
3. Return \(N\)

One more example

• Which of the following queries correctly find the third author in the entire input document?
  • //author[position()=3]
    • Same as //descendant-or-self::node()/author[position()=3]
  • /descendant-or-self::node()
    [name()='author' and position()=3]
  • /descendant-or-self::node() [name()='author'] [position()=3]
XQuery
• XPath + full-fledged SQL-like query language
• XQuery expressions can be
  • XPath expressions
  • FLWOR expressions
  • Quantified expressions
  • Aggregation, sorting, and more...
• An XQuery expression in general can return a new result XML document
  • Compare with an XPath expression, which always returns a sequence of nodes from the input document or atomic values (boolean, number, string, etc.)

A simple XQuery based on XPath
Find all books with price lower than $50
<result>{
  doc("bib.xml")/bibliography/book[@price<50]
}</result>
• Things outside {}’s are copied to output verbatim
• Things inside {}’s are evaluated and replaced by the results
  • doc("bib.xml") specifies the document to query
    • Can be omitted if there is a default context document
  • The XPath expression returns a sequence of book elements
  • These elements (including all their descendants) are copied to output

FLWR expressions
• Retrieve the titles of books published before 2000, together with their publisher
<result>{
  for $b in doc("bib.xml")/bibliography/book
  let $p := $b/publisher
  where $b/year < 2000
  return <book>
    { $b/title }
    { $p }
  </book>
}</result>
• for: loop
  • $b ranges over the result sequence, getting one item at a time
  • let: "assignment"
    • $p gets the entire result of $b/publisher (possibly many nodes)
  • where: filtering by condition
  • return: result structuring
    • Invoked in the "innermost loop," i.e., once for each successful binding of all query variables that satisfies where
An equivalent formulation

• Retrieve the titles of books published before 2000, together with their publisher

```xml
<result>
  for $b in doc("bib.xml")/bibliography/book[year<2000]
    return
      <book>
        { $b/title }
        { $b/publisher }
      </book>
</result>
```

Another formulation

• Retrieve the titles of books published before 2000, together with their publisher

```xml
<result>
  for $b in doc("bib.xml")/bibliography/book
    for $p in $b/publisher
      where $b/year < 2000
    return
      <book>
        { $b/title }
        { $p }
      </book>
</result>
```

Yet another formulation

• Retrieve the titles of books published before 2000, together with their publisher

```xml
<result>
  let $b := doc("bib.xml")/bibliography/book
    where $b/year < 2000
  return
    <book>
      { $b/title }
      { $b/publisher }
    </book>
</result>
```
Subqueries in return

- Extract book titles and their authors; make title an attribute and rename author to writer

```xml
<bibliography>{
  for $b in doc("bib.xml")/bibliography/book
  return
    <book title="{normalize-space($b/title)}">
      for $a in $b/author
      return <writer>{string($a)}</writer>
    </book>
}</bibliography>
```

- `normalize-space(string)` removes leading and trailing spaces from string, and replaces all internal sequences of white spaces with one white space

An explicit join

- Find pairs of books that have common author(s)

```xml
<result>{
  for $b1 in doc("bib.xml")//book
  for $b2 in doc("bib.xml")//book
  where $b1/author = $b2/author
  and $b1/title > $b2/title
  return
    <pair>
      {$b1/title}
      {$b2/title}
    </pair>
}</result>
```

Existentially quantified expressions

- Can be used in `where` as a condition
- Find titles of books in which XML is mentioned in some section

```xml
<result>{
  for $b in doc("bib.xml")//book
  where (some $section in $b//section satisfies contains(string($section), "XML"))
  return $b/title
}</result>
```
Universally quantified expressions

(every $var in collection satisfies condition)
- Can be used in where as a condition
- Find titles of books in which XML is mentioned in every section

```xml
<result>{
  for $b in doc("bib.xml")//book
  where (every $section in $b//section satisfies contains(string($section), "XML"))
  return $b/title
}</result>
```

Aggregation

- List each publisher and the average prices of all its books

```xml
<result>{
  for $pub in distinct-values(doc("bib.xml")//publisher)
  let $price := avg(doc("bib.xml")//book[publisher=$pub]/@price)
  return <publisherpricing>
      <publisher>{$pub}</publisher>
      <avgprice>{$price}</avgprice>
  </publisherpricing>
}</result>
```

- distinct-values(collection) removes duplicates by value
  - If the collection consists of elements (with no explicitly declared types),
    they are first converted to strings representing their "normalized contents"
  - `avg(collection)` computes the average of collection (assuming each item in collection can be converted to a numeric value)

Conditional expression

- List each publisher and, only if applicable, the average prices of all its books

```xml
<result>{
  for $pub in distinct-values(doc("bib.xml")//publisher)
  let $price := avg(doc("bib.xml")//book[publisher=$pub]/@price)
  return <publisherpricing>
      <publisher>{$pub}</publisher>
      <avgprice>({if ($price) then <avgprice>{$price}</avgprice> else ()})</avgprice>
  </publisherpricing>
}</result>
```

- Use anywhere you’d expect a value, e.g.:
  - let $foo := if (...) then else ...
  - return <bar blah="{ if (...) then else else "}"/>
Sorting (a brief history)

- A path expression in XPath returns a sequence of nodes according to original document order.
- For loop will respect the ordering in the sequence.
- August 2002 (http://www.w3.org/TR/2002/WD-xquery-20020816/)
  - Introduce an operator `sort by (sort-by-expression-list)` to output results in a user-specified order.
  - Example: list all books with price higher than $100, in order by first author; for books with the same first author, order by title.
    ```xml
    <result>{
      doc("bib.xml")//book[@price>100]
      sort by (author[1], title)
    }</result>
    ```

Tricky semantics

- List titles of all books, sorted by their ISBN
  ```xml
  <result>{
    (doc("bib.xml")//book sort by (@ISBN))/title
  }</result>
  ```
  - What is wrong?
  - Correct versions
    ```xml
    <result>{
      for $b in doc("bib.xml")//book sort by (@ISBN)
      return $b/title
    }</result>
    ```
    ```xml
    <result>{
    }</result>
    ```

Current version of sorting

Since June 2006
- `sort by` has been ditched.
- A new `order by` clause is added to FLWR.
  - Which now becomes FLWOR.
- Example: list all books in order by price from high to low; for books with the same price, sort by first author and then title.
  ```xml
  <result>{
    for $b in doc("bib.xml")//book[@price>100]
    stable order by
      number($b/price) descending,
      $b/author[1],
      $b/title empty least
    return $b
  }</result>
  ```
Summary

- Many, many more features not covered in class
- XPath is very mature, stable, and widely used
  - Has good implementations in many systems
  - Is used in many other standards
- XQuery is also fairly popular
  - Has become the SQL for XML
  - Has good implementations in some systems

XQuery vs. SQL

- Where did the join go?
- Is navigational query going to destroy physical data independence?
- Strong ordering constraint
  - Can be overridden by `unordered { for... }
  - Why does that matter?