Chapter 9

XQuery

Motivation

- Now that we have XPath, what do we need XQuery for?
- XPath was designed for addressing parts of existing XML documents
- XPath cannot
  - create new XML nodes
  - perform joins between parts of a document (or many documents)
  - re-order the output it produces
  - ...
- Furthermore, XPath
  - has a very simple type system
  - can be hard to read and understand (due to its conciseness)

Data Model

- XQuery closely follows the XML Schema data model
- The most general data type is an *item*
- An item is either a (single) node or an atomic value

Data Model (2)

- XQuery works on *sequences*, which are series of items
- In XQuery every value is a sequence
  - There is no distinction between a single item and a sequence of length one
- Sequences can only contain items; they cannot contain other sequences

Document Representation

- Every document is represented as a tree of nodes
- Every node has a unique node identity that distinguishes it from other nodes (independent of any ID attributes)
- The first node in any document is the document node (which contains the whole document)
- The order in which the nodes occur in an XML document is called the *document order*

Document Representation (2)

- Attributes are not considered children of an element
  - They occur after their element and before its first child
  - The relative order within the attributes of an element is implementation-dependent

Query Language

- We are now going to look at the query language itself
  - Basics
  - Creating nodes/documents
  - FLWOR expressions
  - Advanced topics

Comments

- XQuery uses “smileys” to begin and end comments:
  `(: This is a comment :)`
- These are comments found in a query (to comment the query)
  - Not to be confused with comments in XML documents

Literals

- XQuery supports numeric and string literals
- There are three kinds of numeric literals
  - Integers (e.g. 3)
  - Decimals (e.g. -1.23)
  - Doubles (e.g. 1.2e5)
- String literals are delimited by quotation marks or apostrophes
  - “a string”
  - ’a string’
  - ’This is a “string”’

Input Functions

- XQuery uses input functions to identify the data to be queried
- There are two different input functions, each taking a single argument
  - `doc()`
    - ★ Returns an entire document (i.e. the document node)
    - ★ Document is identified by a Universal Resource Identifier (URI)
  - `collection()`
    - ★ Returns any sequence of nodes that is associated with a URI
    - ★ How the sequence is identified is implementation-dependant
    - ★ For example, eXist allows a database administrator to define collections, each containing a number of documents

Sample Data

- In order to illustrate XQuery queries, we use a sample data file `books.xml` which is based on bibliography data

```
<bib>

  <book year='1994'>
    <title>TCP/IP Illustrated</title>
    <author>
      <last>Stevens</last>
      <first>W.</first>
    </author>
    <publisher>Addison Wesley</publisher>
    <price>65.95</price>
  </book>
```

Sample Data (cont'd)

```
<book year='1992'>
  <title>
    Advanced Programming in the UNIX environment
  </title>
  <author>
    <last>Stevens</last>
    <first>W.</first>
  </author>
  <publisher>Addison Wesley</publisher>
  <price>65.95</price>
</book>
```

Sample Data (cont'd)

```
<book year='2000'>
  <title>Data on the Web</title>
  <author>
    <last>Abiteboul</last> <first>Serge</first>
  </author>
  <author>
    <last>Buneman</last> <first>Peter</first>
  </author>
  <author>
    <last>Suciu</last> <first>Dan</first>
  </author>
  <publisher>Morgan Kaufmann</publisher>
  <price>39.95</price>
</book>
```

Sample Data (cont'd)

```
  <book year='1999'>
    <title>
      The Economics of Technology and Content for Digital TV
    </title>
    <editor>
      <last>Gerbarg</last>
      <first>Darcy</first>
      <affiliation>CITI</affiliation>
    </editor>
    <publisher>Kluwer Academic</publisher>
    <price>129.95</price>
  </book>

</bib>
```

Input Functions (2)

- `doc("books.xml")` returns the entire document
- A run-time error is raised if the `doc` function is unable to locate the document

Input Functions (3)

- XQuery uses XPath to locate nodes in XML data
- An XPath expression can be appended to a `doc` (or `collection`) function to select specific nodes
- For example, `doc("books.xml")//book` returns all book nodes of `books.xml`

Creating Nodes

- So far, XQuery does not look much more powerful than XPath
- We only located nodes in XML documents
- Now we take a look at how to create nodes
- Note that this creates nodes in the *output* of a query; it does *not* update the document being queried

Creating Nodes (2)

- Elements, attributes, text nodes, processing instructions, and comment nodes can all be created using the same syntax as XML
- The following element constructor creates a book element:

```
<book year='1977'>
  <title>Harold and the Purple Crayon</title>
  <author>
    <last>Johnson</last>
    <first>Crockett</first>
  </author>
  <publisher>
    Harper Collins Juvenile Books
  </publisher>
  <price>14.95</price>
</book>
```

Creating Nodes (3)

- Document nodes do not have an explicit syntax in XML
- XQuery provides a special document node constructor
- The query

  ```
  document {}
  ```

  creates an empty document node

Creating Nodes (4)

- Document node constructor can be combined with other constructors to create entire documents

```
document {
  <?xml-stylesheet type='text/xsl' href='trans.xslt'?>
  <!-- I love this book -->
  <book year='1977'>
    <title>Harold and the Purple Crayon</title>
    <author>
      <last>Johnson</last>
      <first>Crockett</first>
    </author>
    <publisher>
      Harper Collins Juvenile Books
    </publisher>
    <price>14.95</price>
  </book>
}
```

Creating Nodes (5)

- Constructors can be combined with other XQuery expressions to generate content dynamically
- In element constructors, curly braces { } delimit enclosed expressions which are evaluated to create content
- Enclosed expressions may occur in the content of an element or the value of an attribute

Creating Nodes (6)

- This query creates a list of book titles from `books.xml`

```
<titles count =
 '{ count(doc("books.xml")//title) }'>
 {
  doc("books.xml")//title
 }
</titles>
```

- The result is:

```
<titles count="4">
    <title>TCP/IP Illustrated</title>
    <title>Advanced Programming ...</title>
    <title>Data on the Web</title>
    <title>The Economics of ...</title>
</titles>
```

Whitespace

- Implementations may discard boundary whitespace (whitespace between tags with no intervening non-whitespace)
- This whitespace can be preserved by an `xmlspace` declaration in the *prolog* of a query
- The prolog of a query is an optional section setting up the compile-time context for the rest of the query

Whitespace (2)

- The following query declares that all whitespace in element constructors must be preserved (which will output the element in exactly the same format)

```
declare xmlspace preserve;

<author>
  <last>Stevens</last>
  <first>W.</first>
</author>
```

- Omitting this declaration (or setting the mode to `strip`) will give:

```
<author><last>Stevens</last><first>W.</first></author>
```

Combining and Restructuring

- The expressiveness of XQuery goes beyond just creating nodes
- Information from one or more sources can be combined and restructured to create new results
- We are going to have a look at the most important expressions and functions

FLWOR

- FLWOR expressions (pronounced “flower”) are one of the most powerful and common expressions in XQuery
- Syntactically, they show similarity to the select-from-where statements in SQL
- However, FLWOR expressions do not operate on tables, rows, and columns

FLWOR (2)

- The name FLWOR is an acronym standing for the first letter of the clauses that may appear
  - For
  - Let
  - Where
  - Order by
  - Return

FLWOR (3)

- The acronym FLWOR roughly follows the order in which the clauses occur
- A FLWOR expression
  - starts with one or more `for` or `let` clauses (in any order)
  - followed by an optional `where` clause,
  - an optional `order by` clause,
  - and a required `return` clause

For and Let Clauses

- Every clause in a FLWOR expression is defined in terms of tuples
- The `for` and `let` clauses create these tuples
- Therefore, every FLWOR expression must have at least one `for` or `let` clause
- We will start with artificial-looking queries to illustrate the inner workings of `for` and `let` clauses

For and Let Clauses (2)

- The following query creates an element named `tuple` in its return clause

```
for $i in (1, 2, 3)
return
  <tuple><i> { $i } </i></tuple>
```

- We bind the variable `$i` to the expression `(1, 2, 3)`, which constructs a sequence of integers
- The above query results in:

```
<tuple><i>1</i></tuple>
<tuple><i>2</i></tuple>
<tuple><i>3</i></tuple>
```

(a `for` clause preserves order when it creates tuples)

For and Let Clauses (3)

- A `let` clause binds a variable to the entire result of an expression
- If there are no `for` clauses, then a single tuple is created

```
let $i := (1, 2, 3)
return
  <tuple><i> { $i } </i></tuple>
```

results in:

```
<tuple><i>1 2 3</i></tuple>
```

For and Let Clauses (4)

- Variable bindings of `let` clauses are added to the tuples generated by `for` clauses

```
for $i in (1, 2, 3)
let $j := ('a', 'b', 'c')
return
  <tuple><i>{ $i }</i><j>{ $j }</j></tuple>
```

results in:

```
<tuple><i>1</i><j>abc</j></tuple>
<tuple><i>2</i><j>abc</j></tuple>
<tuple><i>3</i><j>abc</j></tuple>
```

For and Let Clauses (5)

- `for` and `let` clauses can be bound to any XQuery expression
- Let us do a more realistic example
- List the title of each book in `books.xml` together with the numbers of authors:

```
for $b in doc("books.xml")//book
let $a := $b/author
return
  <book> { $b/title,
    <count> { count($a) } </count> }
  </book>
```

For and Let Clauses (6)

- This results in:

```
<book>
  <title>TCP/IP Illustrated</title>
  <count>1</count>
</book>
<book>
  <title>Advanced Programming ...</title>
  <count>1</count>
</book>
<book>
  <title>Data on the Web</title>
  <count>3</count>
</book>
<book>
  <title>The Economics of Technology ...</title>
  <count>0</count>
</book>
```

Where Clauses

- A `where` clause eliminates tuples that do not satisfy a particular condition
- A return clause is only evaluated for tuples that “survive” the where clause
- The following query returns only books whose prices are less than 50.00:

```
for $b in doc("books.xml")//book
where $b/price < 50.00
return $b/title
```

returns

```
<title>Data on the Web</title>
```

Order By Clauses

- An `order by` clause sorts the tuples before the return clause is evaluated
- If there is `no order by` clause, then the results are returned in document order
- The following example lists the titles of books in alphabetical order:

```
for $t in doc("books.xml")//title
order by $t
return $t
```

- An order spec may also specify whether to sort in ascending or descending order (using `ascending` or `descending`)

Return Clauses

- Any XQuery expression may occur in a `return` clause
- Element constructors are very common in `return` clauses
- The following query represents an author's name as a string in a single element

```
for $a in doc("books.xml")//author
return
  <author> { string($a/first), " ",
             string($a/last) } </author>
```

results in

```
<author>W. Stevens</author>
<author>W. Stevens</author>
<author>Serge Abiteboul</author>
<author>Peter Buneman</author>
<author>Dan Suciu</author>
```

Return Clauses (2)

- The following query adds another level to the hierarchy:

```
for $a in doc("books.xml")//author
return
  <author>
    <name> { $a/first, $a/last } </name>
  </author>
```

results in

```
<author>
    <name>
        <first>W.</first>
        <last>Stevens</last>
    </name>
</author>
...
```

Operators

- The operators shown in the queries so far have not been covered yet
- XQuery has three different kinds of operators
  - Arithmetic operators
  - Comparison operators
  - Sequence operators

Arithmetic Operators

- XQuery supports the arithmetic operators `+`, `-`, `*`, `div`, `idiv`, and `mod`
- The `idiv` and `mod` operators require integer arguments, returning the quotient and the remainder, respectively
- If an operand is a node, atomization is applied (casting the content to an atomic type)
- If an operand is an empty sequence, the result is an empty sequence
- If an operand is untyped, it is cast to a double (raising an error if the cast fails)

Comparison Operators

- XQuery has different sets of comparison operators: value comparisons, general comparisons, node comparisons, and order comparisons
- Value comparison operators compare atomic values:

eq	equals
ne	not equals
lt	less than
le	less than or equal to
gt	greater than
ge	greater than or equal to

General Comparisons

- The following query raises an error

```
for $b in doc("books.xml")//book
where $b/author/last eq 'Stevens'
return $b/title
```

  because we try to compare several author names to `'Stevens'` (books may have more than one author)
- We need a general comparison operator for this to work
- A general comparison returns true if **any** value in a sequence of atomic values matches

General Comparisons (2)

- The following table shows the corresponding general comparison operator for each value comparison operator

value comparison	general comparison
eq	=
ne	!=
lt	<
le	<=
gt	>
ge	>=

Built-in Functions

- XQuery also offers a set of built-in functions and operators
- We focus only on the most common ones here
- SQL users will be familiar with the `min()`, `max()`, `count()`, `sum()`, and `avg()` functions
- Other familiar functions include
  - Numeric functions like `round()`, `floor()`, and `ceiling()`
  - String functions like `concat()`, `string-length()`, `substring()`, `upper-case()`, `lower-case()`
  - Cast functions for the various atomic types

User-Defined Functions

- When a query becomes large and complex, it becomes easier to understand if it is split up into functions
- For example, if the titles of books written by a given author are needed in different places of a query, a function could be defined (in the prolog):

```
define function books-by-author($last, $first)
  as element()*
{
  for $b in doc("books.xml")//book
  for $a in $b/author
  where $a/first = $first
  and   $a/last = $last
  return $b/title
}
```

Library Modules

- Functions can be put into library modules, which can be imported by any query
- Every module in XQuery is either a main module (which contains a query body) or a library module (which has no query body)
- A library module begins with a module declaration which provides a URI for identification:

```
module "http://example.com/xq/book"

define function ...

define function ...
```

Library Modules (2)

- Any module can import another module using a `import module` declaration
- This declaration has to specify a URI and may specify a location where the module can be found

```
import module "http://example.com/xq/book"
       at "file:///home/xquery/..."
```

Positional Variables

- The `for` clause supports positional variables
- This identifies the position of a given item in the sequence generated by an expression
- The following query returns the titles of books with an attribute that numbers the books:

```
for $t at $i in doc("books.xml")//title
return
  <title pos=' { $i } '>
    { string($t) }
  </title>
```

Positional Variables (2)

- The output of this query looks like this:

```
<title pos="1">
  TCP/IP Illustrated
</title>
<title pos="2">
  Advanced Programming in ...
</title>
<title pos="3">
  Data on the Web
</title>
<title pos="4">
  The Economics of Technology ...
</title>
```

Eliminating Duplicates

- Data (or intermediate query results) often contain duplicate values
- The following query returns one of the authors twice

```
doc("books.xml")//author/last
```

which outputs

```
<last>Stevens</last>
<last>Stevens</last>
<last>Abiteboul</last>
<last>Buneman</last>
<last>Suciu</last>
```

Eliminating Duplicates (2)

- The `distinct-values()` function is used to remove duplicate values
- It extracts values of a sequence of nodes and creates a sequence of unique values
- Example:

```
distinct-values(doc("books.xml")//author/last)
```

which outputs

```
Stevens Abiteboul Buneman Suciu
```

Combining Data Sources

- A query may bind multiple variables in a `for` clause to combine data from different expressions
- Suppose we have a file named `reviews.xml` that contains book reviews:

```
<reviews>
  <entry>
    <title>Data on the Web</title>
    <price>34.95</price>
    <review>
      A very good discussion of
      semi-structured database ...
    </review>
  </entry>
    ...
```

Combining Data Sources (2)

- A FLWOR expression can bind one variable to the bibliography data and another to the review data
- In the following query we join data from the two files:

```
for $t in doc("books.xml")//title,
    $e in doc("reviews.xml")//entry
where $t = $e/title
return
  <review>
    { $t, $e/review }
  </review>
```

Combining Data Sources (3)

- This returns the following answer:

```
<review>
  <title>TCP/IP Illustrated</title>
  <review>
    One of the best books on TCP/IP.
  </review>
</review>
<review>
  <title>Advanced Programming in the ...</title>
  <review>
    A clear and detailed discussion of ...
  </review>
</review>
...
```

Inverting Hierarchies

- XQuery can be used to do general transformations
- In the example file, books are sorted by title
- If we want to group books by publisher, we have to “pull up” the publisher element (i.e., invert the hierarchy of the document)
- The next slide shows a query to do this

Inverting Hierarchies (2)

```
<listings> {
  for $p in
    distinct-values(doc("books.xml")//publisher)
  order by $p
  return
    <result>
      { $p }
      { for $b in doc("books.xml")//book
        where $b/publisher = $p
        order by $b/title
        return $b/title
      }
    </result>
  }
</listings>
```

Inverting Hierarchies (3)

Result:

```
<listings>
  <result>Addison-Wesley
    <title>Advanced Programming ...</title>
    <title>TCP/IP Illustrated</title>
  </result>
  <result>Kluwer Academic Publishers
    <title>The Economics of ...</title>
  </result>
  <result>Morgan Kaufmann Publishers
    <title>Data on the Web</title>
  </result>
</listings>
```

Quantifiers

- Some queries need to determine whether
  - at least one item in a sequence satisfies a condition
  - every item in sequence satisfies a condition
- This is done using quantifiers:
  - `some` is an existential quantifier
  - `every` is a universal quantifier

Quantifiers (2)

- The following query shows an existential quantifier
- We are looking for a book where *at least one* of the authors has the last name ‘Buneman’:

```
for $b in doc("books.xml")//book
where some $a in $b/author
      satisfies ($a/last = 'Buneman')
return $b/title
```

which returns:

```
<title>Data on the Web</title>
```

Quantifiers (3)

- The following query shows a universal quantifier
- We are looking for a book where *all* of the authors have the last name 'Stevens':

```
for $b in doc("books.xml")//book
where every $a in $b/author
      satisfies ($a/last = 'Stevens')
return $b/title
```

which returns:

```
<title>TCP/IP Illustrated</title>
<title>Advanced Programming ...</title>
<title>The Economics of Technology ...</title>
```

Quantifiers (4)

- A universal quantifier applied to an empty sequence always yields true (there is no item violating the condition)
- An existential quantifier applied to an empty sequence always yields false (there is no item satisfying the condition)

Conditional Expressions

- XQuery's conditional expressions (`if - then - else`) are used in the same way as in other languages
- In XQuery, both the `then` and the `else` clause are required
- The empty sequence `()` can be used to specify that a clause should return nothing
- The following query returns all authors for books with up to two authors and "et al." for any remaining authors

Conditional Expressions (2)

```
for $b in doc("books.xml")//book
return
  <book> { $b/title } {
    for $a at $i in $b/author
    where $i <= 2
    return <author> { string($a/last), ", ",
                      string($a/first) }
           </author>
    }
    { if (count($b/author) > 2)
      then <author> et al. </author>
      else ()
    }
  </book>
```

Conditional Expressions (3)

Result:

```
<book>
    <title>TCP/IP Illustrated</title>
    <author>Stevens, W.</author>
</book>
 <book>
    <title>Advanced Programming in ...</title>
    <author>Stevens, W.</author>
</book>
 <book>
    <title>Data on the Web</title>
    <author>Abiteboul, Serge</author>
    <author>Buneman, Peter</author>
    <author>et al. </author>
</book>
 <book>
    <title>The Economics of Technology ...</title>
</book>
```

Summary

- XQuery was designed to be compact and compositional
- It is well-suited to XML-processing tasks like data integration and data transformation