Querying XML with XQuery

Jonathan Robie
DataDirect Technologies
jonathan.robie@datadirect.com
Basic Queries
XML Query History

- Early query facilities for SGML
- 1998: "Roll your own query language"
- Feb 1998: XQL proposal
  - [http://metalab.unc.edu/xql](http://metalab.unc.edu/xql)
- Aug 1998: XML-QL submission
  - [http://www.w3.org/TR/NOTE-xml-ql/](http://www.w3.org/TR/NOTE-xml-ql/)
- Dec 1998: W3C QL'98 Workshop
  - [http://www.w3.org/TandS/QL/QL98](http://www.w3.org/TandS/QL/QL98)
- Nov 1999: XPath Recommendation
  - [http://www.w3.org/TR/xpath](http://www.w3.org/TR/xpath)
XQuery Working Drafts

- XML Query Requirements
  http://www.w3.org/TR/xquery-requirements/
- XML Query Use Cases
  http://www.w3.org/TR/xquery-use-cases/
- XQuery 1.0: An XML Query Language
  http://www.w3.org/TR/xquery/
- XML Query 1.0 and XPath 2.0 Data Model
  http://www.w3.org/TR/xpath-datamodel/
- XSLT 2.0 and XQuery 1.0 Serialization
  http://www.w3.org/TR/xslt-xquery-serialization/
- XQuery 1.0 and XPath 2.0 Functions and Operators
  http://www.w3.org/TR/xpath-functions/
- XML Query Formal Semantics
  http://www.w3.org/TR/xquery-semantics/
XML Query Data Model

- Joint with XPath 2.0, XSL 2.0
- Ordered, labeled forest
- Based on XML Information Set, PSVI
  - but much leaner!
- Seven kinds of nodes
  - document, element, attribute, text, namespace, processing instruction, and comment
  - Nodes have identity
Document Node vs. Root Element

<?xml version="1.0"?>
<?format eyesight="20/200"?>

<!-- You can't see the document node in this file. -->

<!-- These children of the document node may be:
  1. comments
  2. processing instructions
  3. the root element -->

<rootElement>Hi, Mom!</rootElement>
Document Order

- "First character order"
  - Document node first
  - Element nodes before their children
  - Siblings in order of occurrence
  - Attributes before child elements – but in any order
  - Regions of uninterrupted text are treated as 'text nodes'

```xml
<book year="1994">
  <title>CP/IP Illustrated</title>
  <author>
    <last>S tevens</last>
    <first>W</first>
  </author>
</book>
```
Data Model – Types and Values

- Classes of documents
  - Well-formed documents
  - DTD-valid documents
  - W3C XML Schema-validated documents

- Two interpretations of a node
  - typed value – with XML Schema type
  - string value – lexical representation

- Untyped data is xs:anyType or xs:anySimpleType
  - Failed validation
  - No validation attempted
XQuery and the Data Model

- DOM
- SAX
- DBMS
- XML
- Java
- COBOL

W3C XML Query Data Model

XML Query

W3C XML Query Data Model

- DOM
- SAX
- DBMS
- XML
- Java
- COBOL
XML Query Formal Semantics

- Static Semantics
  - Type inference rules
  - Structural subsumption

- Dynamic Semantics
  - Value inference rules
  - Define the meaning of XQuery expressions in terms of the XML Query Data Model
The XQuery Language

- XQuery is a functional language
  - Evaluating expressions – not performing commands.
  - A query is an expression
  - Expressions can be nested with full generality.
  - A functional language - but syntax often does not look functional.

- Based on OQL, SQL, XQL, XML-QL, XPath
XQuery Expressions

- Element constructors
- Path expressions
- Restructuring
  - Sorting
  - FLWR expressions
  - Conditional expressions
  - Quantified expressions
- Operators and functions
- List constructors
- Expressions that test or modify datatypes
Basics
Literals and Data Types

- Comments
  - (: doesn't this look cheerful? :)

- W3C XML Schema simple types

- String
  - "a string"
  - 'a string'
  - "This is a string, isn't it?"
  - 'This is a "string"
  - "a "" or a ' delimits a string literal"
  - 'a " or a " delimits a string literal"
Character Entities and Strings

- Predeclared character entities

<table>
<thead>
<tr>
<th>&lt;</th>
<th>&lt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>&amp;</td>
<td>&amp;</td>
</tr>
<tr>
<td>&quot;</td>
<td>&quot;</td>
</tr>
<tr>
<td>'</td>
<td>'</td>
</tr>
</tbody>
</table>

- Using character entities in strings

'&lt;bold&gt;A sample element.&lt;/bold&gt;'
Literals and Data Types

- Numeric Literals
  - 1 (: An integer :)
  - -2 (: An integer :)
  - +2 (: An integer :)
  - 1.23 (: A decimal :)
  - -1.23 (: A decimal :)
  - 1.2e5 (: A double :)
  - -1.2E5 (: A double :)

Jonathan Robie — Querying XML with XQuery
Input functions

- `doc()` returns an entire document, identifying the document by a URI. To be more precise, it returns the document node.

- `collection()` returns a collection, which is any sequence of nodes that is associated with a URI. This is often used to identify a database to be used in a query.
Built-in Functions

- The following are used in this tutorial:
  - max(), min(), sum(), count(), avg()
  - distinct-values(), empty(), exists()
  - collection(), doc()
  - contains(), concat()
  - QName()
  - position(), last()

- Described more fully as encountered
- Many more in Functions & Operators spec.
Element Constructors
XML Element Constructors

(: These constructors look like the XML they construct :) 

<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>
Expressions in XML Constructors

- XQuery expressions are made to be combined
- Angle brackets escape to XML syntax
- Curly braces escape to XQuery expression syntax

(: the year contains two subtractions! :) 
<book year="\{2002-4-4\}"
  <title>TCP/IP Illustrated</title>
  <publisher>Addison-Wesley</publisher>
  <price>\{ 70-5-0.05 \}</price>
</book>
Computed Constructor Syntax

element book {
    (: Attributes must appear first! :) 
    attribute year { 1994 },
    element title { "TCP/IP Illustrated" },
    element author {
        element last { "Stevens" },
        element first { "W." }
    },
    element publisher { "Addison-Wesley" },
    element price { 65.95 }
}

Computing Names Dynamically

- Query

\[
\text{let } b := \text{concat("book", "001")} \\
\text{return} \\
\text{element } \{ b \} \{ b \}
\]

- Output:

\[
<\text{book001}>\text{book001}</\text{book001}>
\]
Which Constructor Syntax?

- XML constructor syntax looks like XML
  - Nothing new to learn
  - Easy to distinguish from rest of query
- Either syntax works for most construction
  - Matter of taste
  - Most people use XML constructor syntax
- Computed constructor syntax allows names to be computed at run time
Other Constructors

- `document { ... }`
  - Creates a document node
  - See next slide...

- `text { ... }`
  - Creates a text node
  - Used for untyped character data
  - Same as text in an element (when datatypes are not being used)
Creating a Document Node

document {
  <?format eyesight="20/200"?>

  <!-- You can't see the document node in this file. -->

  <!-- These children of the document node may be:
      1. comments
      2. processing instructions
      3. the root element -->

  <rootElement>Hi, Mom!</rootElement>
}

Practice: Element Constructors

- Ensure that XQuery is properly installed.
- Create the following element using XML syntax:

```xml
<product id="123">
  <name>Left-handed smoke shifter</name>
  <price>16.34</price>
  <description>Finally, a shifter for left-handers!</description>
</product>
```
Practice: Element Constructors

- Create the same element, computing the numeric values with arithmetic expressions.
- Create the same element using computed element constructor syntax.
- Create the same element using computed element constructor syntax, creating the element name "product" at run time.
Path Expressions
Path Expressions

- Start with an input function or a variable
  - In some implementations, you can start with a leading / - what it is bound to is implementation-defined
  - In some implementations, variables may be predeclared
- Navigate using step expressions
- Filter using predicates (part of a step expression)
Input Functions: `doc()`

- One parameter: a URI
  - `doc("books.xml")`
  - `doc("http://www.eg.com/books.xml")`

- URI identifies a document
  - Returns document node
  - Not same as document element!
Input Functions: collection()

- One parameter: a URI
  collection("jdbc:ddt:sql://host:1433;database='northwind'")

- Returns a sequence of nodes from a collection
- Collections are vendor-defined, identified by URI
Navigation: Sample Input Document

```xml
<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>
</bib>
```
Navigating with Path Expressions

- **Single slash, name test**
  - \( \text{doc("bib.xml")/bib} \)
  - "all bib elements that are children of document node"

- **Double slash**
  - \( \text{doc("bib.xml")//book} \)
  - "all books elements that are descended from the document node"

- **Predicates**
  - \( \text{doc("bib.xml")//book[@year="1994"]} \)
  - "all books elements, containing a year attribute equal to '1994', that are descended from the document node"
Numeric Predicates

- First book:
  - `input()/bib/book[1]`

- First author of each book:
  - `input()/bib/book/author[1]`
  - `input()//author[1]`

- First author in document:
  - `(input()/bib/book/author)[1]`
  - `(input()//author)[1]`
Step Expressions

- Single slashes separate "steps":
  ```xml
doc("bib.xml") / child::bib[ count(child::book) > 3 ]
  ```
- Basic components of a step:
  - Context node – sets the context for the step
  - Axis – states the direction of the step relative to context node (child, descendant, attribute, self, descendant-or-self, parent)
  - NodeTest – returns nodes that satisfy a condition on their name (NameTest) or kind (KindTest)
  - Predicates – filters nodes based on a condition
Steps and Axes

- The "axis" determines the direction in which a step navigates from the context node
- XQuery normally uses "abbreviated syntax" for axes. Single slash means child axis:
  
  \[ \text{doc("bib.xml")/bib} \]

  \[ \Rightarrow \]

  \[ \text{doc("bib.xml")/child::bib} \]

- Possible axes in XQuery: child, descendant, attribute, self, descendant-or-self, parent
- Explicit use of axes rare in XQuery
Steps: Context Nodes and Evaluation

- Context nodes are drawn from the expression on the left of the slash
- LeftExpr / RightExpr
  - Evaluate LeftExpr – must be a sequence of nodes, or error is raised.
  - Using each node from LeftExpr as a "context node"; evaluate RightExpr – must be a sequence of nodes, or error is raised.
  - Merge results for each context node; sort in document order (which eliminates duplicates).

Examples:
- `doc("bib.xml")/child::bib/child::book/child::author`
- `doc("bib.xml")/descendant::book/child::author[1]`
Referring to the Context Node

- In path expressions, "." refers to the context node.
- The self axis also refers to the context node.
- The following queries are equivalent:
  - `doc("bib.xml")/bib`
  - `doc("bib.xml")/bib/.
  - `doc("bib.xml")/bib/self::*`
Steps: Two Kinds of NodeTest

- **NameTest**
  - Element name: 'bib' matches bib elements
  - Attribute name: '@bib' matches bib attributes
  - Wildcard:
    - '*' matches any element
    - '@*' matches any attribute

- **KindTest**
  - node() matches any node
  - comment() matches any comment
  - text() matches any text node
  - processing-instruction() matches any pi
  - `processing-instruction("target")` matches pi for 'target'
Namespaces and NameTest

- Declaring and using prefixes
  
  declare namespace xhtml = "http://www.w3.org/1999/xhtml"
  
  doc("aspect.xhtml")//xhtml:table

- Matching based on URI, not prefix
Steps and \\

- Abbreviated notation is defined using axis notation
- Formal definition of `//`:
  - `/descendant-or-self::node()`
  - `a//b` has three steps!
- Examples:
  - `doc("bib.xml")//book`  
    => `doc("bib.xml")/descendant-or-self::node()/child::book`
  - `doc("bib.xml")//@year`  
    => `doc("bib.xml")/descendant-or-self::node()/attribute::year`
Steps and // – Ramifications

- Context node comes from the middle step – not expression on left of //
- position() is relative to context node
- First author in any given element:
  - `input()//author[1]`
  - `=> input()/descendant-or-self::node()/child::author[1]`
Predicates – Common Patterns

- Condition on element value
  \[
  \text{doc("bib.xml")//author[last="Stevens"]}
  \]

- Condition using function
  \[
  \text{doc("bib.xml")//book[count(author)>2]}
  \]
  \[
  \text{doc("bib.xml")//book[contains(title, "Web")]}\]

- Combining conditions
  \[
  \text{doc("bib.xml")//author[last="Stevens" and first="W."]}
  \]
  \[
  \text{doc("bib.xml")//author[last="Stevens"]}[first="W."]\]
  \[
  \text{doc("bib.xml")//author[last="Stevens" or first="W."]}\]
Predicates – Numeric Predicates

- position() is the position of the context item:
  \[\text{doc("bib.xml")//author[3]}\]
  \[\text{doc("bib.xml")//author[position()=3]}\]

- last() is the length of the sequence containing context item:
  \[\text{doc("bib.xml")//author[last()]}\]
  \[\text{doc("bib.xml")//author[position()=last()]}\]

- Expressions using position:
  \[\text{doc("bib.xml")//author[position() mod 2 = 1]}\]
  \[\text{doc("bib.xml")//author[(position()>2) and (position()<5)]}\]
Path Expressions in Constructors

- XQuery expressions are designed to be easily combined.
- Combining path expressions and element constructors:

```xml
<publishers count="{count(doc("bib.xml")//publisher)}">
  <head> Publishers </head>
  { doc("bib.xml")//publisher }
</publishers>
```
FLWOR Expressions

- for - let - where - order by - return
- Similar to SQL's SELECT - FROM - WHERE

```xml
for $book in doc("bib.xml")//book
let $title := $book/title
where $book/publisher = "Addison-Wesley"
order by $title
return
  <book>
    {
      $title,
      $book/author
    }
  </book>
```
for vs. let

- for iterates on a sequence, binds a variable to each node
- let binds a variable to a sequence as a whole

```xml
for $book in doc("bib.xml")//book
let $a := $book/author
where contains($book/publisher, "Addison-Wesley")
return
  <book>
    {
      $book/title,
      <count> Number of authors: { count($a) } </count>
    }
  </book>
```
Conditional Expressions

- IF expr THEN expr ELSE expr

for $h$ in input()//holding
return
  <holding>
    {
      $h/title,
      if ($h/@type = "Journal")
        then $h/editor
        else $h/author
    }
  </holding>
Inner Joins

    $quote in doc("www.bookstore.com/quotes.xml")//listing
where $book/isbn = $quote/isbn
order by $book/title
return
    <book>
        { $book/title }
        { $quote/price }
    </book>
Outer Joins

for $book in doc("bib.xml")//book
order by $book/title
return

    <book>
    { $book/title }
    {
        for $review in doc("reviews.xml")//review
            where $book/isbn = $review/isbn
            return $review/rating
    }
    </book>
Quantifiers

- EVERY var IN expr SATISFIES expr
- SOME var IN expr SATISFIES expr

```xquery
for $b in input()//book
  where every $p in $b//para satisfies contains($p, "sailing")
    and contains($p, "windsurfing")
  return $b/title
```
Transformations – Bibliography

<?xml version="1.0"?>
<bib>
  <book>
    <title> Harold and the Purple Crayon </title>
    <author>
      <lastname> Johnson </lastname>
      <firstname> Crockett </firstname>
    </author>
    <pubinfo>
      <publisher> Harper and Row </publisher>
      <price> 4.76 </price>
      <year> 1995 </year>
    </pubinfo>
  </book>
</bib>
Books by Author

<?xml version="1.0"?>
<bib>
  <book>
    <title>Harold and the Purple Crayon</title>
    <author>
      <last>Johnson</last>
      <first>Crockett</first>
    </author>
    <pubinfo>
      <publisher>Harper and Row</publisher>
      <price>4.76</price>
      <year>1995</year>
    </pubinfo>
  </book>
  <booksByAuthor>
    <author>
      <name>
        <last>Johnson</last>
        <first>Crockett</first>
      </name>
    </author>
    <title>Harold and the Purple Crayon</title>
    <title>Harold's Fairy Tale</title>
    <title>Harold and the Circus</title>
    <title>Harold's ABC's</title>
    <author></author>
  </booksByAuthor>
</bib>
Removing Duplicates

- `distinct-values()`

```
distinct-values(doc("bib.xml")//publisher)
```

```
<publisher>Addison-Wesley</publisher>
<publisher>Morgan Kaufmann Publishers</publisher>
<publisher>Kluwer Academic Publishers</publisher>
```

- Extracts distinct atomic values:

  Addison-Wesley
  Morgan Kaufmann Publishers
  Kluwer Academic Publishers
Inverting the Hierarchy

```
<results>
  
  let $a := doc("data/xmp-data.xml")//author
  for $last in distinct-values($a/last),
      $first in distinct-values($a[last=$last]/first)
  return
    <result>
      { $last, $first }
      
      for $b in doc("data/xmp-data.xml")/bib/book
          where some $ba in $b/author satisfies ($ba/last = $last and $ba/first=$first)
          return $b/title
    sort by (title)
  
</result>

sort by (last, first)

</results>
```
Combining Sequences

- **Union**
  - Combines two sequences, eliminates duplicates, returns in doc order
  - Two syntaxes
  - Examples
    - `doc("data/xmp-data.xml")//(author | editor)`
    - `doc("data/xmp-data.xml")//(author union editor)`

- **Intersection**
  - Returns intersection of two sequences in document order
  - Example
    - `author/last intersect editor/last`
Combining Sequences (Continued)

- **Except**
  - Removes items from a sequence
  - Handy for editing elements
  - Example:
    ```
    return $b/* except $b/author
    ```
SQL-like Queries
A Relational View

**users**

<table>
<thead>
<tr>
<th>USERID</th>
<th>NAME</th>
<th>RATING</th>
</tr>
</thead>
<tbody>
<tr>
<td>1243</td>
<td>humphrey</td>
<td></td>
</tr>
</tbody>
</table>

**items**

<table>
<thead>
<tr>
<th>ITEMNO</th>
<th>DESCRIPTION</th>
<th>OFFERED_BY</th>
<th>RESERVE_PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1066</td>
<td>unicycle</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**bids**

<table>
<thead>
<tr>
<th>USERID</th>
<th>ITEMNO</th>
<th>BID_AMOUNT</th>
<th>BID_DATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1243</td>
<td>1066</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
SQL vs. XQuery

"Find item numbers of Bicycles"

- SQL:

  SELECT itemno
  FROM items AS i
  WHERE description LIKE 'Bicycle'
  ORDER BY itemno;

- XQuery:

  for $i in doc("items.xml")//item_tuple
  order by $i/itemno
  where contains($i/description, "Bicycle")
  return $i/itemno
Let and Aggregates

"List item numbers that have more than 10 bids, and their bid counts"

- **SQL:**
  
  ```sql
  SELECT itemno, count(*) AS bid_count
  FROM bids
  GROUP BY itemno
  HAVING count(*) > 10
  ORDER BY bid_count DESC;
  ```

- **XQuery:**
  
  ```xquery
  for $i in distinct-values(doc("bids.xml")//bid_tuple/itemno)
  let $count := count(doc("bids.xml")//bid_tuple[itemno = $i])
  order by $count descending
  where $count > 10
  return <popular_item count="{$count}">{$i}</popular_item
  ```
Inner Join

"List names of users and descriptions of the items they offer"

- SQL:

```
SELECT u.name, i.description
FROM users AS u, items AS i
where u.userid = i.offered_by
ORDER BY name, description;
```

- XQuery

```
for $u in doc("users.xml")//user_tuple,
   $i in doc("items.xml")//item_tuple
where $u/userid = $i/offered_by
order by $u/name, $i/description
return
  <offering> {
    $u/name,
    $i/description
  } </offering>
```
Outer Join

"List names of users and descriptions of the items they offer, including users who have not offered any items"

- **SQL:**

```sql
SELECT u.name, i.description
FROM users u
  LEFT OUTER JOIN items i
  ON u.userid = i.offered_by
ORDER BY u.name, i.description
```

- **XQuery:**

```xml
for $u in doc("users.xml")//user_tuple
  order by $u/name
  return
  <seller>
    {
      $u/name,
      for $i in doc("items.xml")//item_tuple
        where $u/userid = $i/offered_by
        order by $u/description
        return $i/description
    }
  </seller>
```
Queries with Positional Variables

for $t$ at $i$ in doc("books.xml")//title
return <title pos="{$i}">{string($t)}</title>

<title pos="1">TCP/IP Illustrated</title>
<title pos="2">Advanced Programming in the Unix environment</title>
<title pos="3">Data on the Web</title>
<title pos="4">The Economics of Technology and Content for Digital TV</title>
### Position May Convey Meaning

<table>
<thead>
<tr>
<th>Title</th>
<th>publisher</th>
<th>price</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP/IP Illustrated</td>
<td>Addison-Wesley</td>
<td>65.95</td>
<td>1994</td>
</tr>
<tr>
<td>Advanced Programming in the Unix environment</td>
<td>Addison-Wesley</td>
<td>65.95</td>
<td>1992</td>
</tr>
<tr>
<td>Data on the Web</td>
<td>Morgan Kaufmann Publishers</td>
<td>39.95</td>
<td>2000</td>
</tr>
<tr>
<td>The Economics of Technology and Content for Digital TV</td>
<td>Kluwer Academic Publishers</td>
<td>129.95</td>
<td>1999</td>
</tr>
</tbody>
</table>
Position May Convey Meaning (Continued)

```xml
<table border="1">
  <thead>
    <tr>
      <td>title</td>
      <td>publisher</td>
      <td>price</td>
      <td>year</td>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>TCP/IP Illustrated</td>
      <td>Addison-Wesley</td>
      <td>65.95</td>
      <td>1994</td>
    </tr>
    <tr>
      <td>Advanced Programming in the Unix environment</td>
      <td>Addison-Wesley</td>
      <td>65.95</td>
      <td>1992</td>
    </tr>
  </tbody>
</table>
```
Extracting XML from HTML Table

let $t := doc("bib.xhtml")//table[1]
for $r in $t/tbody/tr
return
  <book>
    {
      for $c at $i in $r/td
        return element{ data($t/thead/tr/td[$i]) } 
        { string( $c) } 
    } 
  </book>
Output from Previous Query

<book>
  <title>TCP/IP Illustrated</title>
  <publisher>Addison-Wesley</publisher>
  <price>65.95</price>
  <year>1994</year>
</book>

<book>
  <title>Advanced Programming in the Unix environment</title>
  <publisher>Addison-Wesley</publisher>
  <price>65.95</price>
  <year>1992</year>
</book>
Arithmetic, Functions, Comparisons
Arithmetic

- Binary operators: +, -, *, div, idiv, mod
- Unary operators: +, -
- "Atomization" extracts values from nodes
  \[ 4 \times <\text{foo}>5</\text{foo}> \]
- If an operand is untyped, it is cast to double (see above example)
- If an operand is empty sequence, result is empty sequence (c.f. SQL nulls).
Atomization

- If the given value is already a single atomic value or an empty sequence, return the given value.
- If the given value is a single node, the typed value of the node is extracted and returned; if the result is a sequence containing more than one item, a type exception is raised.
- In any other case, atomization raises a type exception.
## Comparisons

<table>
<thead>
<tr>
<th>Value Comparison</th>
<th>General Comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td>eq</td>
<td>=</td>
</tr>
<tr>
<td>ne</td>
<td>!=</td>
</tr>
<tr>
<td>lt</td>
<td>&lt;</td>
</tr>
<tr>
<td>le</td>
<td>&lt;=</td>
</tr>
<tr>
<td>gt</td>
<td>&gt;</td>
</tr>
<tr>
<td>ge</td>
<td>&gt;=</td>
</tr>
</tbody>
</table>
Value Comparisons

- Nostalgic FORTRAN syntax
- Compares two single values:

  ```xml
  for $b in doc("books.xml")//book
  where $b/title eq "Data on the Web"
  return $b/price
  ```

- Atomization is applied to operands
- Error if not single values
- Does not cast untyped data:

  ```xml
  (: !! Raises an error !! :) 
  for $b in doc("books.xml")//book
  where $b/price eq 100.00
  return $b/title
  ```
General Comparisons

- Conventional operator syntax
- Implicit existential quantification:

    for $b in doc("books.xml")//book
    where $b/author/last = "Smith"
    return $b/price

    for $b in doc("books.xml")//book
    where some $l in $b/author/last
    satisfies $l eq "Smith"
    return $b/price

- Atomization is applied to operands
- Attempts to cast untyped data:

    for $b in doc("books.xml")//book
    where $b/price = 100.00
    return $b/title
Untyped Data in General
Comparisons

- If either operand is untyped, it is cast to a required type:
  - If the type of the other operand is numeric, the required type is xs:double.
  - If the most specific type of the other operand is xs:anySimpleType, the required type is xs:string.
  - Otherwise, the required type is the type of the other operand.

- If the cast fails, a dynamic error is raised.

- Example: 5 eq <foo>5</foo>
Document Order Comparisons

- Document order comparisons
  - $a << b, a >> b
  - $a is $b, $a isnot $b
  - Compare two values
  - Type error if not two values

- Example
  for $p in doc("surgery.xml")
    //section[section.title = "Procedure"]
    where not(some $a in $p//anesthesia
         satisfies $a << ($p//incision)[1] )
  return $p
User-declared Functions

- Defined in XQuery syntax
- May be recursive or mutually recursive
- May have typed parameters or returns
Writing and Calling a Function

declare function flatten-author( $a as element )
{
    <author>
    {
        string-value($a/last), ",", string-value($a/first)
    }
    </author>
}

for $a in doc("data/xmp-data.xml")//(author | editor)
return flatten-author($a)
Recursive Functions

declare function depth($e as element) returns integer
{
    (: An empty element has depth 1
    Otherwise, add 1 to max depth of children :)
    if (empty($e/*))
        then 1
    else max(depth($e/*)) + 1
}

depth(doc("partlist.xml"))
Library Modules

```xml
module "http://example.com/xquery/library/book"
declare function toc($b)
{
    for $section in $book-or-section/section
    return
        <section>
            { $section/@* , $section/title , toc($section) }
        </section>
}
```

```xml
import module namespace b = "http://example.com/xquery/library/book"
    at "file:///c:/xquery/lib/book.xq"

<toc>
{
    for $s in doc("xquery-book.xml"]/book
    return b:toc($s)
}
</toc>
```
Library Modules

- Every module is a main module or a library module.

[30] Module ::= MainModule | LibraryModule
[31] MainModule ::= Prolog QueryBody
[32] LibraryModule ::= ModuleDecl Prolog

- A main module must have a query expression; a library module may not have one.
- Importing a module imports its:
  - Functions
  - Variable declarations (from the prolog)
Types in Queries
Untyped Queries

- Untyped is best for purely structural functions

```xquery
declare function reverse($items)
{
    let $count := count($items)
    for $i in 0 to $count
    return $items[$count - $i]
}
reverse( 1 to 5 )
```
Predefined Types

- Predefined types require no schema
  - XML node types (element, attribute, *etc.*)
  - XML Schema built-in types

```xquery
declare function is-document-element($e as element())
  returns xs:boolean
{
  if ($e/.. instance of doc())
    then true()
  else false()
}
```
Predefined Types

An example using only values

declare function fibo(n as xs:integer) as xs:integer
{
  if ($n = 0) then 0
  else if ($n = 1) then 1
  else (fibo($n - 1) + fibo($n - 2))
}

let $seq := 1 to 10
for $n in $seq
return <fibo n="{$n}">{ fibo($n) }</fibo>
**Schema imports**

```xml
import schema "urn:examples:xmp:bib" at "c:/dev/schemas/eg/bib.xsd"
default element namespace = "urn:examples:xmp:bib"

declare function books-by-author($a as element(b:author))
as element(b:title) *
{
    for $b in doc("books.xml")/bib/book
    where some $ba in $b/author satisfies
        ($ba/last=$a/last and $ba/first=$a/first)
    order by $b/title
    return $b/title
}

(: The following function call raises an error – wrong type :) :

for $b in ("books.xml")/bib/book
return books-by-author($b)
```
Schema Imports

- Schemas are imported using the 'schema' expression in the prolog:

  ```
  import schema "http://www.w3.org/1999/xhtml"
  at "http://www.w3.org/1999/xhtml/xhtml.xsd"
  ```

- "at" clause is optional
- Built-in XML Schema types are predefined
- Each element name, attribute name, or type name may be defined only once.
- If a schema import tries to redefine an existing name, an error results.
Schema Import with Namespace Prefix

import schema namespace b = "urn:examples:xmp:bib"
at "c:/dev/schemas/eg/bib.xsd"

declare function books-by-author($a as element(b:author))
as element(b:title)*
{
  for $b in doc("books.xml")/b:bib/b:book
  order by $b/b:title
  where some $ba in $b/b:author satisfies
    ($ba/b:last=$l and $ba/b:first=$f)
  return $b/b:title
}
Schema Import Is Optional

- Users need not import schemas to query typed data
- Implementations need not allow schema import
- If schemas are not imported, only predefined types can be named in a query
Function Conversion Rules

- Used to convert function parameters or returns

- Basic intuitive definition:
  - If the types match exactly, the parameter is accepted.
  - Types derived from the required type are accepted.
  - Untyped data is cast to the required type (but the cast may fail).
  - If the required type is an atomic type, and the parameter is a node, atomization is applied before trying to match the types.

- For details, see the spec
External functions

- External functions are declared in the external environment (Java, SQL, etc.)
- Type signatures provide type safety

```
declare function outtie($v as xs:integer) as xs:integer external
```
Implicit Validation

- Element constructors with known types are implicitly validated

import schema "urn:examples:xmp:bib" at "c:/dev/schemas/eg/bib.xsd"
default element namespace = "urn:examples:xmp:bib"

<book year="1994">
  <title>Catamaran Racing from Start to Finish</title>
  <author><last>Berman</last><first>Phil</first></author>
  <publisher>W.W. Norton & Company</publisher>
</book>
Validating Locally Declared Elements

```
import schema namespace bib="urn:examples:xmp:bib"

validate context bib:book
{
  <bib:price>49.99</bib:price>
}
```
Validation Modes

- Lax (default) validates if type is known
- Strict requires a known type, always validates

```
import schema namespace bib="urn:examples:xmp:bib"
validation strict
<bib:price>49.99</bib:price>
```

- Skip never validates

```
import schema namespace bib="urn:examples:xmp:bib"
validation strict
<bib:price>49.99</bib:price>
```
Optional Static Typing

- Error if query is inconsistent with schema
- Type errors can be detected without any data

```xml
import schema "urn:examples:xmp:bib" at "c:/dev/schemas/eg/bib.xsd"
default element namespace = "urn:examples:xmp:bib"
declare function books-by-author($a as element(author))
as element(title)*
{
  for $b in doc("books.xml")/bib/book
  order by $b/title
  where some $ba in $b/author satisfies
    ($ba/last=$a/last and $ba/first=$a/flirts)
  return $b/title
}
```
## Predefined Types

<table>
<thead>
<tr>
<th>Sequence Type Declaration</th>
<th>What it Matches</th>
</tr>
</thead>
<tbody>
<tr>
<td>element()</td>
<td>Any element node</td>
</tr>
<tr>
<td>attribute()</td>
<td>Any attribute node</td>
</tr>
<tr>
<td>document-node()</td>
<td>Any document node</td>
</tr>
<tr>
<td>node()</td>
<td>Any node</td>
</tr>
<tr>
<td>text()</td>
<td>Any text node</td>
</tr>
<tr>
<td>processing-instruction()</td>
<td>Any processing instruction node</td>
</tr>
<tr>
<td>processing-instruction(&quot;xml-stylesheet&quot;)</td>
<td>Any processing instruction node whose target is &quot;xml-stylesheet&quot;</td>
</tr>
<tr>
<td>comment()</td>
<td>Any comment node</td>
</tr>
<tr>
<td>empty()</td>
<td>An empty sequence</td>
</tr>
<tr>
<td>item()</td>
<td>Any node or atomic value</td>
</tr>
<tr>
<td>QName</td>
<td>An instance of a specific XML Schema built-in type, identified by the name of the type; e.g., xs:string, xs:boolean, xs:decimal, xs:float, xs:double, xs:anyType…</td>
</tr>
</tbody>
</table>
Occurrence Indicators

- +, ?, *
- + means "one or more":
  element() +
- ? means "zero or one":
  xs:integer?
- * means "zero or more":
  document-node() *
## Types from Imported Schemas

<table>
<thead>
<tr>
<th>Sequence Type Declaration</th>
<th>What it Matches</th>
</tr>
</thead>
<tbody>
<tr>
<td>element(creator, person)</td>
<td>An element named creator of type person</td>
</tr>
<tr>
<td>element(creator)</td>
<td>Any element named creator of type &quot;xs:string&quot; – the type declared for creator in the schema.</td>
</tr>
<tr>
<td>element(*, person)</td>
<td>Any element of type person.</td>
</tr>
<tr>
<td>element(type(person)/last)</td>
<td>An element named last of type &quot;xs:string&quot; – the type declared for last elements inside the person type.</td>
</tr>
<tr>
<td>attribute(@price,</td>
<td>An attribute named price of type currency.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>attribute(@*, currency)</td>
<td>Any attribute of type currency.</td>
</tr>
<tr>
<td>bib:currency</td>
<td>A value of the user-defined type 'currency'</td>
</tr>
</tbody>
</table>
Nillable types

- The 'nillable' keyword matches nilled elements
- Example: `element(n, nillable person)`
  - Matches `<n xsi:nil="true" />`
  - Matches `<n><first>Jack</first><last>Canada</last></n>`
Typed Variables

- Type assertion - raise error if wrong type
- Example:
  
  ```
  for $b in doc("data/xmp-data.xml")//book
  let $c as element()+ := $b//author
  return <count>{ count($c) }</count>
  ```
Variable Declarations

- Occur in query prolog
- Not to be confused with variable bindings
- Internal variable declarations:
  declare variable $titles { doc("books.xml")//title }
- External variable declarations:
  declare var $x external
  declare var $i as xs:integer external
instance of

- Tests an item for a given type.
- Examples:
  - `<foo/>` instance of `element()`
  - `3.14` instance of `xs:decimal`
  - "foo" instance of `xs:string`
  - `(1, 2, 3)` instance of `xs:integer*`
  - `()` instance of `xs:integer?`
  - `(1, 2, 3)` instance of `xs:integer+`
typeswitch

- Chooses an expression to evaluate based on type.
- Example:

```xquery
declare function wrapper($x) returns element(wrap)
{
    typeswitch ($x)
    case $i as xs:integer
        return <wrap xsi:type="xs:integer">{ $i }</wrap>
    case $d as xs:decimal
        return <wrap xsi:type="xs:decimal">{ $d }</wrap>
    default
        return error("unknown type!")
}

wrapper(1)
```
Constructor Functions

- Used for construction or casting
- Built in schema types:
  - `xs:date("2000-01-01")`
- Constructor functions automatically generated for imported types:
  - `my:currency("5.23")`
- Facets are checked:
  - `import schema namespace bib="urn:examples:xmp:bib"
    bib:isbn("012345678X")`
treat as

- Tests for correct type at run-time
- Example:

$myaddress treat as element(*, USAddress)
Summary
Why XQuery?

- Powerful transformations
- Native XML programming
- XML Views of relational data
- Optimizable in many environments
- Related to concepts people already know
- Many implementations
- The accepted W3C XML Query Language
- Preliminary update proposal
XML Query Home Page

- Pointers to all current XQuery specifications
- Over 20 XQuery implementations (at various stages)
- Articles on XQuery
- Grammar test pages
- Comments lists and discussion lists
- [http://www.w3.org/XML/Query.html](http://www.w3.org/XML/Query.html)
Questions

- Today
- Later:
  - jonathan.robie@datadirect-technologies.com
- Feedback email list:
  - public-qt-comments@w3.org
- Public email list:
  - www-ql@w3.org