Introduction to XQuery

Priscilla Walmsley
Managing Director, Datypic
http://www.datypic.com
pwalmsley@datypic.com
About this course

• These slides are excerpted from a 2-day course on XQuery taught by Priscilla Walmsley.

• If you are interested in having Priscilla teach an XQuery course to your group, please contact her at pwalmsley@datypic.com.
XQuery in Context
What is XQuery?

• A query language
  – Pull information from a database or document
  – The "SQL of XML"

• A transformation language?
  – Restructure information from a database or document, for presentation, repurposing, etc.

• A search language?
  – Search across a database for relevant hits
What is XQuery?

• A query language that allows you to:
  – select elements/attributes from input documents
  – join data from multiple input documents
  – make modifications to the data
  – calculate new data
  – add new elements/attributes to the results
  – sort your results
XQuery Example

input document

```xml
<order num="00299432" date="2004-09-15" cust="0221A">
  <item dept="WMN" num="557" quantity="1" color="tan"/>
  <item dept="ACC" num="563" quantity="1"/>
  <item dept="ACC" num="443" quantity="2"/>
  <item dept="MEN" num="784" quantity="1" color="blue"/>
  <item dept="MEN" num="784" quantity="1" color="red"/>
  <item dept="MEN" num="784" quantity="1" color="red"/>
  <item dept="WMN" num="557" quantity="1" color="sage"/>
</order>
```

query

```xml
for $d in distinct-values(doc("ord.xml")//item/@dept)
let $items := doc("ord.xml")//item[@dept = $d] order by $d
return <department name="{$d}"
  totalQuantity="{sum($items/@quantity)}"/>
```

results

```xml
<department name="ACC" totalQuantity="3"/>
<department name="MEN" totalQuantity="2"/>
<department name="WMN" totalQuantity="2"/>
```
Use Case #1: Search and Browse

• Usually semi-structured, narrative content
  – sometimes combined with structured data
  – e.g. medical journals, poetry manuscripts, hotel reviews

• Usually stored in a "native" XML database
  – e.g. MarkLogic, Berkeley DB, eXist

• Example
  – What medical journal articles since 2004 mention "artery" and "plaque" within 3 words of each other?
Full-Text Search Capabilities in XQuery

• Fairly weak in standard XQuery 1.0
  – Regular expression matching
  – Simple "contains" function
• But, no typical full-text search capabilities:
  – Stemming, thesaurus, proximity, weighting
• Vendor-specific extensions make up for limitations
• XQuery and XPath Full-Text
  – a separate spec
  – very impressive functionality
Use Case #2: XML in Relational Database

- Usually a combination of highly structured data and more flexible data
- Supported by major relational database vendors
  - SQL Server 2005, Oracle, DB2
- Why?
  - include narrative content with structured data
    - e.g. product descriptions in the PRODUCT table
  - allow flexibility in content
    - e.g. changing set of product properties for different kinds of products
- Features
  - An XML data type that allows you to store XML in a column
  - Indexing, schema validation
  - Querying by embedding XQuery in SQL
Use Case #2: XML in Relational Database

<table>
<thead>
<tr>
<th>num</th>
<th>props</th>
</tr>
</thead>
<tbody>
<tr>
<td>557</td>
<td>&lt;properties&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;sleeveLength&gt;19&lt;/sleeveLength&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;/properties&gt;</td>
</tr>
<tr>
<td>443</td>
<td>&lt;properties&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;capacity&gt;80&lt;/capacity&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;/properties&gt;</td>
</tr>
<tr>
<td>784</td>
<td>&lt;properties&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;sleeveLength&gt;25&lt;/sleeveLength&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;extraButtons&gt;2&lt;/extraButtons&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;/properties&gt;</td>
</tr>
</tbody>
</table>

```sql
select num, props.query('//sleeveLength') as slength
from prod_properties
where props.exist('/properties/sleeveLength[. > 20]') = 1
```

<table>
<thead>
<tr>
<th>num</th>
<th>slength</th>
</tr>
</thead>
<tbody>
<tr>
<td>784</td>
<td>&lt;sleeveLength&gt;25&lt;/sleeveLength&gt;</td>
</tr>
</tbody>
</table>
Use Case #3: Integrating Disparate Data Sources

Office Applications
- save as XML 1

Relational Databases
- store as XML 2
- export as

XML Databases
- store as XML 3
- export as

Packaged Software
- store as XML 4
- export as

XQuery
- join, transform, aggregate

Result XML
Use Cases: Anything, really...

- Anywhere in application code you would currently use XPath, or XSLT, or DOM, e.g.:
  - in a pipeline process to split or subset an XML document
  - to narrow down results returned from a Web service
  - to manipulate or create a configuration file stored as XML
- Ad-hoc fact-finding about XML data
XQuery Design Goals

• A language that is:
  – useful for both structured and unstructured data
  – protocol independent, allowing a query to be evaluated on any system with predictable results
  – a declarative language rather than a procedural one
  – strongly typed
    • allows for optimization and better error detection
  – able to accept collections of multiple documents
  – compatible with other W3C standards
    • XML 1.1, Namespaces, XML Schema, XPath
The XQuery Processing Model (Simplified)

XML Input ➔ parse ➔ XQuery Processor ➔ analyze and evaluate (using context) ➔ serialize (or pass on) ➔ XML Output
XML Input

• Could be data that is:
  – a textual XML document on a file system
  – retrieved from a Web service
  – stored in an XML database
  – stored in a relational database as XML
  – created in memory by program code

• Can take the form of:
  – a single XML document
  – a collection of several documents
  – a fragment of a document (e.g. sequence of elements)
XQuery, XSLT and XPath

FLWOR Expressions
XML Constructors
Query Prolog
User-Defined Functions

Conditional Expressions
Arithmetic Expressions
Quantified Expressions
Built-In Functions & Operators
Data Model

Stylesheets
Templates etc.

Path Expressions
Comparison Expressions
Some Built-In Functions

XQuery 1.0
XPath 2.0
XSLT 2.0
XQuery vs. XSLT: Decision Factors

• Use case
• Language capabilities
• Availability of relevant implementations
• Performance
• Programming style
Current Status

• 1.0 is a W3C Recommendation as of January 2007
• Developed by the W3C XML Query Working Group
  – [http://www.w3.org/XML/Query](http://www.w3.org/XML/Query)
• Work in progress on version 3.0.
  – There is no 2.0.
<catalog>
  <product dept="WMN">
    <number>557</number>
    <name language="en">Linen Shirt</name>
    <colorChoices>beige sage</colorChoices>
  </product>
  <product dept="ACC">
    <number>563</number>
    <name language="en">Ten-Gallon Hat</name>
  </product>
  <product dept="ACC">
    <number>443</number>
    <name language="en">Golf Umbrella</name>
  </product>
  <product dept="MEN">
    <number>784</number>
    <name language="en">Rugby Shirt</name>
    <colorChoices>blue/white blue/red</colorChoices>
    <desc>Our <i>best-selling</i> shirt!</desc>
  </product>
</catalog>
<prices>
    <priceList effDate="2004-11-15">
        <prod num="557">
            <price currency="USD">29.99</price>
            <discount type="CLR">10.00</discount>
        </prod>
        <prod num="563">
            <price currency="USD">69.99</price>
        </prod>
        <prod num="443">
            <price currency="USD">39.99</price>
            <discount type="CLR">3.99</discount>
        </prod>
    </priceList>
</prices>
<order num="00299432" date="2004-09-15" cust="0221A">
  <item dept="WMN" num="557" quantity="1" color="beige"/>
  <item dept="ACC" num="563" quantity="1"/>
  <item dept="ACC" num="443" quantity="2"/>
  <item dept="MEN" num="784" quantity="1" color="blue/white"/>
  <item dept="MEN" num="784" quantity="1" color="blue/red"/>
  <item dept="WMN" num="557" quantity="1" color="sage"/>
</order>
Easing into XQuery
Selecting Nodes from the Input Document

- Open the product catalog
  
  \[ \text{doc("cat.xml")} \]

  calls a function named \text{doc} to open the \text{cat.xml} file

- Retrieve all the product names
  
  \[ \text{doc("cat.xml")/catalog/product/name} \]

  navigates through the elements in the document using a \textit{path expression}

- Select only the product names from department ACC
  
  \[ \text{doc("cat.xml")/catalog/product[@dept='ACC']/name} \]

  uses a \textit{predicate} to limit the products
The Results

doc("cat.xml")/catalog/product[@dept='ACC']/name

<name language="en">Ten-Gallon Hat</name>
<name language="en">Golf Umbrella</name>
Path Expressions and FLWOR Expressions

- Another way of saying the same thing:

  ```
  doc("cat.xml")/catalog/product[@dept='ACC']/name
  ```

  **path expression**

  ```
  for $product in doc("cat.xml")/catalog/product
  where $product/@dept='ACC'
  return $product/name
  ```

  **FLWOR expression**
for $product in
doc("cat.xml")/catalog/product
where $product/@dept='ACC'
order by $product/name
return $product/name

<name language="en">Golf Umbrella</name>
<name language="en">Ten-Gallon Hat</name>
Wrap the Results in a **ul** Element

```xml
<ul type="square">
  {for $product in doc("cat.xml")/catalog/product
   where $product/@dept='ACC'
   order by $product/name
   return $product/name}
</ul>
```

```xml
<ul type="square">
  <name language="en">Golf Umbrella</name>
  <name language="en">Ten-Gallon Hat</name>
</ul>
```
Wrap Each Name in an \textit{li} Element

\begin{verbatim}
<ul type="square">
  for $product in doc("cat.xml")/catalog/product
  where $product/@dept='ACC'
  order by $product/name
  return <li>{$product/name}</li>
</ul>
\end{verbatim}

\begin{verbatim}
<ul type="square">
  <li><name language="en">Golf Umbrella</name></li>
  <li><name language="en">Ten-Gallon Hat</name></li>
</ul>
\end{verbatim}
Eliminate the name Elements

```xml
<ul type="square">
  for $product in doc("cat.xml")/catalog/product
    where $product/@dept='ACC'
  order by $product/name
  return <li>{data($product/name)}</li>
</ul>

<ul type="square">
  <li>Golf Umbrella</li>
  <li>Ten-Gallon Hat</li>
</ul>
```
The Data Model
• Nodes
  – elements, attributes and other XML components

• Atomic values
  – individual data values, not an "element" or "attribute"

• Items
  – Atomic values or nodes
Components of the Data Model

- sequence
- item
- contains
- 0..n
- node
  - kind
  - string value
- atomic value
  - type
- document node
- attribute node
  - name
  - type
  - typed value
- element node
  - name
  - type
  - typed value
- text node
  - typed value
- comment node
- processing instruction node
  - name
An XML Hierarchy of Nodes

document node

  element node (catalog)

    element node (product)

      attribute node (dept)

      element node (number)

        text node (784)

      element node (name)

        attribute node (language)

        text node ("Rugby Shirt")

      element node (colorChoices)

        text node ("blue/white blue/red")

    element node (desc)

      text node ("Our ")

      element node (i)

        text node ("best-selling")

      text node (" shirt!")

  </catalog>
Nodes:

- have a "kind"
  - element, attribute, text, document, processing instruction, comment
- may have a name
  - number, dept
- have a string value
  - "557", "MEN"
- may have a typed value
  - integer 557, string MEN
- have a unique identity

<number>557</number> dept="MEN"
Family Relationships of Nodes

- **Children**
  - Element nodes can have zero, one or more children
    - Attributes are not considered children of an element node
- **Parent**
  - Each element and attribute node has one parent
    - Even though attributes are not children of elements, elements are parents of attributes!
- **Ancestors**
  - A node's parent, parent's parent, etc. all the way back to the document node
- **Descendants**
  - A node's children, children's children, etc.
- **Siblings**
  - Other nodes that have the same parent
Atomic Values

• Individual data values
  – no association with any particular node

• Every atomic value has a type
  – based on the XML Schema atomic types
    • e.g. `xs:string`, `xs:integer`
  – can also be the generic type `xs:untypedAtomic`
    • when not validated with a schema
How Atomic Values Are Created

- using a literal value
  - "Catalog", 12

- the result of a function
  - \texttt{count(//number)}

- explicitly extracting the value of a node
  - \texttt{data(<number>557</number>)}

- automatically extracting the value of a node
  - process is called \textit{atomization}
  - \texttt{substring(<number>557</number>,1,2)}
Sequences

• Ordered lists of zero, one or more items
• A sequence of one item is exactly the same as the item itself
• A sequence of zero items is known as "the empty sequence"
  – different from zero, a zero-length string ("")
• There are no sequences within sequences
• Similar to XPath 1.0 node sets, except that they:
  • are ordered
  • can contain duplicates
  • can contain atomic values as well as nodes
How Sequences Are Created

• Result of an expression that returns nodes
  - `catalog//product`
    • returns a sequence of `product` element nodes
  - `catalog//foo`
    • returns the empty sequence

• Constructed manually
  - `(1, 2, 3)`
    • returns a sequence of 3 atomic values (integer)
  - `(1 to 6)`
    • returns a sequence of 6 atomic values (integer)
  - `()`
    • returns the empty sequence
Expressions: Basic Building Blocks

for $d$ in distinct-values(doc("ord.xml")//item/@dept)
let $items := doc("ord.xml")//item[@dept = $d]
order by $d
return <department name="{$d}"
    totalQuantity="{sum($items/@quantity)}"/>
The XQuery Syntax

- A declarative language of nested expressions
- Compact, non-XML syntax
- Case-sensitive
- Whitespace
  - tabs, space, carriage return, line feed
  - allowed (ignored) between language tokens
  - considered significant in quoted strings and constructed elements
- No special end-of-line character
Keywords and Names

• Keywords and operators
  – case-sensitive, generally lower case
  – may have several meanings depending on context
    • e.g. "*" or "in"
  – no reserved words

• All names must be valid XML names
  – for variables, functions, elements, attributes
  – can be associated with a namespace
Evaluation Order

- Every kind of expression has an evaluation order
  - e.g. \texttt{and} takes precedence over \texttt{or}

\[
\text{true()} \text{ and true()} \text{ or false()} \text{ and false()}
\]

- Parentheses can be used around any expression to affect evaluation order

\[
\text{true()} \text{ and (true()} \text{ or false())} \text{ and false()}
\]
Literal Values and Constants

• Literal values can be expressed as:
  – strings (in single or double quotes)
    \[
    \text{doc("cat.xml")//product/@dept} = \text{"WMN"}
    \]
  – numbers
    \[
    \text{doc("ord.xml")//item/@quantity} > 1
    \]
  – values constructed to be of a specific type
    \[
    \text{doc("prc.xml")//@effDate} > \text{xs:date("2004-10-11")}
    \]
Most Commonly Used Types

- String: `xs:string`
- Numeric types
  - e.g. `xs:integer`, `xs:decimal`, `xs:double`
- Date and time types
  - `xs:date` (YYYY-MM-DD)
  - `xs:time` (HH:MM:SS)
  - `xs:dateTime` (YYYY-MM-DDTHH:MM:SS)
- Others
  - `xs:boolean` (true/false)
  - `xs:anyURI`
Variables

- Identified by a name preceded by a $ $
- Variables are defined in several places
  - FLWOR expressions
  - Query prologs
  - Outside the query by the processor
  - Function signatures

```xml
for $prod in (doc("cat.xml")//product)
  return $prod/number
```

```xml
declare function local:getProdNum ($prod as element()) as element()
  {
    $prod/number
  };
```
Function Calls

- An argument can be any single expression – e.g. a variable reference, a path expression
- An argument may be required to have a certain type

```
substring($prodName, 1, 5)
```
• XQuery comments
  – Delimited by ( : and : )
  – Anywhere insignificant whitespace is allowed
  – Do not appear in the results

• XML comments
  – May appear in the results
  – XML-like syntax

(: This query... :)

<!-- This element... -->
Comparisons

• Two kinds:
  – *value* comparisons
    • `eq, ne, lt, le, gt, ge`
    • used to compare single values
  – *general* comparisons
    • `=, !=, <, <=, >, >=`
    • can be used with sequences of multiple items
    • recommended for general use
Value vs. General Comparisons

\[ \text{doc("books.xml")//book[1]/author = 'Smith'} \]

– true if the book has at least one \texttt{author} child equal to 'Smith'

\[ \text{doc("books.xml")//book[1]/author eq 'Smith'} \]

– true if there is only \texttt{one author} child, and its value is equal to 'Smith'

– false if there is no \texttt{author} child, or one \texttt{author} child that's not equal to 'Smith'

– raises an error if there is more than one \texttt{author} child
Conditional Expressions

• if-then-else syntax

```xml
for $prod in (doc("cat.xml")/catalog/product) return if ($prod/@dept = 'ACC')
  then <acc>{data($prod/number)}</acc>
  else <other>{data($prod/number)}</other>
```

• parentheses around if expression are required
• else is always required
  – but it can be just else ()
**Effective Boolean Value**

- "if" expression must be boolean
  - if it is not, its *effective boolean value* is found
- effective boolean value is false for:
  - the `xs:boolean` value `false`
  - the number 0 or `NaN`
  - a zero-length string
  - the empty sequence
- otherwise it is true (e.g. a list of elements)

```xml
if (doc("ord.xml")//item) then "Item List: " else ""
```
Nesting Conditional Expressions

- Conditional expressions can be nested
  – provides "else if" functionality

```xml
if ($prod/@dept = 'ACC')
then <accessory>{data($prod/number)}</accessory>
else if ($prod/@dept = 'WMN')
  then <womens>{data($prod/number)}</womens>
  else if ($prod/@dept = 'MEN')
    then <mens>{data($prod/number)}</mens>
    else <other>{data($prod/number)}</other>
```
Logical Expressions

- **and** and **or** operators
  - **and** has precedence over **or**
  - use parentheses to change precedence

```bash
if ($isDiscounted and
    ($discount > 10 or $discount < 0))
    then 10 else $discount
```

- Use **not** function to negate

```bash
if (not($isDiscounted)) then 0 else $discount
```

- As with conditional expressions, effective boolean value is evaluated
The Query Prolog

• Declarations of various settings, such as:
  – namespace declarations
  – function declarations
  – imports of external modules and schemas
  – default collation

• Appears before the body of the query
  – each declaration separated by a semicolon
xquery version "1.0";
declare boundary-space preserve;
declare namespace ord = "http://datypic.com/ord";
declare function local:getProdNums
    ($catalog as element()) as xs:integer*
    {for $prod in $catalog/product
        return xs:integer($prod/number)};
<title>Order Report</title>,
(for $item in doc("ord.xml")//item
    order by $item/@num
    return $item)
Path Expressions

- Used to traverse an input document, selecting elements and attributes of interest

```
doc("ord.xml")/order/item/@dept

doc("ord.xml")/order/item[@dept = 'ACC']

item

item[3]

//order/item

doc("ord.xml")//item
```
Structure of a Path Expression

Path Expression

Step 1 (filter expression)
Step 2 (axis step)
Step 3 (axis step)

```
doc("ord.xml")/order/item[@dept = 'ACC']
```

function call
node tests
predicate
Components of an Axis Step

1. The axis (optional)
   - the direction to navigate
2. The node test
   - the nodes of interest by name or node kind
3. The predicates (optional and repeating)
   - the criteria used to filter nodes

/child::item[@dept = 'ACC']
Axes

- Twelve axes allow you to specify the "direction" to navigate
  - e.g. child::, parent::, descendant::
- Their names are followed by ::
  - some have abbreviations
- If none is specified, child:: is the default
child Axis

• Returns children
  – child elements, PIs, comments, text
  – but not attributes

• The default axis if none is specified

\[
\text{doc("ord.xml")/order/item} \\
\text{doc("ord.xml")/child::order/child::item} \\
\text{doc("ord.xml")/*/item}
\]
attribute Axis

- Returns the attributes of an element
- Abbreviated with "@"

```
doc("ord.xml")/order/item/@dept
```

```
doc("ord.xml")/order/item/attribute::dept
```

```
doc("ord.xml")/order/item/@*
```
descendant, descendant-or-self Axes

- Returns the children, the children's children, etc.
- Abbreviated by "//"

**Examples**

```
doc("ord.xml")//item
```

```
doc("ord.xml")/descendant::item
```
parent Axis

- Returns the parent element of a node
- Applies to all node kinds
- Abbreviated with ".."

- $prodNum/..
  - Returns the parent of the node bound to $prodNum

- parent::*
  - Returns the parent of the current context

- parent::*name
  - Returns the parent of the current context if it is a name
• **self**
  - the node itself (abbreviated as ".")

• **ancestor, ancestor-or-self**
  - the parent, the parent's parent, and so on

• **following and preceding**
  - all nodes that follow/precede it (except descendants or ancestors), in document order

• **following-sibling and preceding-sibling**
  - siblings that follow/precede it, in document order
Overview of Axes

- preceding
- preceding-sibling
- ancestor
- parent
- self
- following
- following-sibling
- child
- descendant
- descendant-or-self
• Can consist of a:
  – node name (for elements/attributes)
    
    ```xml
    doc("ord.xml")/order/item/@dept
    ```
  – node kind
    
    ```xml
    doc("cat.xml")/catalog/element()
    doc("cat.xml")//number/text()
    doc("cat.xml")//desc/node()
    ```
  – wildcard (*)
    
    ```xml
    doc("ord.xml")/order/*/@*
    ```
    all the attributes of all the element children of order
Filter Expressions

• Steps that use expressions instead of axes and node tests

- `doc("cat.xml")/catalog/product`  
- `$catDoc/catalog/product`  
- `product/(number | name)`  
- `product/(if (desc) then desc else name)`
Predicates

- Filter nodes based on specific criteria
- Enclosed in square brackets [ and ]
- Zero, one or more in each step

```xml
doc("cat.xml")//product[number < 500]
```
• Using `number` in a predicate does not mean that `number` elements are returned:

  All `product` elements whose `number` child is less than 500:
  ```
  product[number < 500]
  ```

  All `number` elements whose value is less than 500:
  ```
  product/number[. lt 500]
  ```

  A period ("."') is used to indicate the context item itself
Predicates and Comparison Operators

• General comparisons vs. value comparisons

All products that have only one number child, whose value is less than 500:

\[ \text{product[number \text{ lt } 500]} \]

All products that have at least one number child, whose value is less than 500:

\[ \text{product[number < 500]} \]

may have other number children whose values are greater than 500
Comparisons: != vs. not(=)

- Beware of the != operator

The product elements that have a number and it is not equal to 528

```plaintext
product[number != 528]
```

The product elements that do not have a number equal to 528 (they may or may not have a number)

```plaintext
product[not(number = 528)]
```
Predicates and Boolean Values

- Expression evaluates to a boolean value
  - effective boolean value (EBV) is used

All items that have a $dept$ attribute that is equal to 'ACC':

```xml
doc("ord.xml")//item[@dept = 'ACC']
```

true if the $dept$ attribute exists and is equal to 'ACC'
false if the $dept$ attribute exists and is not equal to 'ACC'
() if the $dept$ attribute doesn't exist ==> converted to false

All items that have a $dept$ attribute:

```xml
doc("ord.xml")//item[@dept]
```

the $dept$ attribute if it exists ==> converted to true
() if the $dept$ attribute doesn't exist ==> converted to false
• Can use a number in the predicate to indicate the position of the child

The 4th product child of catalog:

`catalog/product[4]`

The 4th child of catalog (regardless of its name):

`catalog/*[4]`
• The position refers to the position within the parent, not within the document as a whole.

  The 4\textsuperscript{th} number child within any given parent (\texttt{product} in this case), so it returns the empty sequence.

  \texttt{/catalog}//\texttt{number}[4]

  The 4\textsuperscript{th} number child in the document.

  \texttt{(/catalog}//\texttt{number})[4]
Using the `position` and `last` Functions

- **position** returns the position of the node in the current sequence

  The first three `product` children of `catalog`:

  ```
  catalog/product[position() < 4]
  ```

- **last** returns the number of items in the current sequence

  The last `product` child of `catalog`:

  ```
  catalog/product[last()]
  ```
XPath Order

• XPaths return nodes in document order
  – i.e. the order of their start tags in the input doc

/catalog/product[last()]
returns last node using the same order as in the input document

• Exception: "reverse" axes (ancestor, preceding, etc.) return nodes in reverse document order

/catalog/product[4]/preceding-sibling::*[last()]
returns the first sibling in document order
Multiple Predicates

• More than one predicate can be used to specify multiple constraints in a step — evaluated left to right

Products whose number is less than 500 and whose department is ACC:

\[
\text{product[number < 500][@dept = "ACC"]}
\]

Of the products whose department is 'ACC', select the 2\textsuperscript{nd} one:

\[
\text{product[@dept = "ACC"]}[2]
\]

Take the 2\textsuperscript{nd} product, if it's in the ACC department, select it:

\[
\text{product}[2][@dept = "ACC"]
\]
A predicate can appear as part of an XPath within another predicate

catalog elements that contain at least one product whose number is less than 500

catalog[product[number < 500]]

catalog elements that contain any element that is equal to 528

catalog[.//*[.= '528']]
More Complex Predicates

- Predicate can contain any expression

- Products whose department contains 'A':
  
  \[
  \text{product} \quad [\text{contains}(@\text{dept}, \ "A")]
  \]

- Products that have at least one child other than number:
  
  \[
  \text{product}\[* \text{except} \text{number}]
  \]

- Every other product:
  
  \[
  \text{product} \quad [\text{position()} \text{mod} \ 2 = 0]
  \]

- Products that have desc children if the variable $\text{descFilter}$ is true, otherwise all products:
  
  \[
  \text{product}[\text{if} \ (\text{descFilter}) \ \text{then} \ \text{desc} \ \text{else} \ \text{true}()]
  \]

- Integers from one to a hundred that are divisible by 5:
  
  \[
  (1 \text{ to } 100)[. \text{mod} \ 5 = 0]
  \]
Paths Returning Atomic Values

• The last step in a path can return an atomic value (unlike in XPath 1.0)

  product/name/substring(.,1,9)
  sum(//item/(@price * @qty))

• But only the last step

  product/name/substring(.,1,9)/replace(.,'x','y')
Collections and Documents

Collection whitman

Collection manuscripts

- Document loc.00002.xml
- Document loc.00004.xml
- Document loc.00006.xml
- Document loc.00009.xml

Collection works

Collection leaves55

- Document ppp.00271.1.xml

... 

Collection leaves56

- Document ppp.00237.13.xml

... 

The `doc` Function

- References a single document via a URI
- Using some XML databases, the URI is a name assigned to that document in the DB – like a file system, may depend on context

```xml
doc("loc.00002.xml")
doc("/whitman/manuscripts/loc.00002.xml")
```

- Other processors will dereference the URI

```xml
doc("file:///C:/cat.xml")
doc("http://www.datypic.com/cat.xml")
```
Collections

• The `collection` function references a collection via a URI
  – Returns a sequence of document nodes

  `collection("/whitman/works")`

• Collections are implementation defined
  • MarkLogic accepts a URI that serves as the name of a collection defined within the database
  • Saxon accepts either:
    – a directory name, or
    – the URI of an XML document that lists the documents that make up the collection
Accessing Collections

- Calls to collection functions can be combined with path expressions

```
collection("/whitman/works")/TEI.2
```

- ...or iterated in FLWOR expressions

```
for $doc in collection("/whitman/manuscripts")
  return $doc//relations
```
Paths and Context

- Path expressions are always evaluated in a particular context item.
- The initial context item (if any) is set outside the scope of the query.

\[ \text{doc("ord.xml")/order/item} \]

- sets the context in the first step to the document node of order.xml.

\[ \text{order/item} \]

- relies on the processor to provide the context (the order child of what?).

\[ /\text{order/item} \]

- relies on the processor to provide the context (the order root element in what document(s)?)
Adding Elements and Attributes
3 Ways to Add Elements/Attributes to Results

• Including them from the input document
  – like most of our previous examples

• Using direct constructors
  – XML-like syntax

• Using computed constructors
  – special syntax using curly braces
  – allows for dynamic names
Including from the Input Document

```xml
for $prod in
    doc("cat.xml")/catalog/product[@dept='ACC']
return $prod/name
```

```
<name language="en">Ten-Gallon Hat</name>
<name language="en">Golf Umbrella</name>
```

- **name** elements are included as is
  - along with their attributes (and children if any)
  - not just their atomic values
- no opportunity to change attributes, children, namespace
Direct Element Constructor Example

```html
<html>
  <h1>Product Catalog</h1>
  <ul>
    {for $prod in doc("cat.xml")/catalog/product return <li>#{data($prod/number)} is {data($prod/name)}</li>}</ul>
</html>
```

```html
<html>
  <h1>Product Catalog</h1>
  <ul>
    <li>#557 is Linen Shirt</li>
    <li>#563 is Ten-Gallon Hat</li> ...
  </ul>
</html>
```
Direct Element Constructors

- Use XML-like syntax
  - and follow the same rules (proper nesting, case sensitivity, etc.)
- Can contain:
  - literal content
  - other direct element constructors
  - enclosed expressions (in curly braces)
    - can evaluate to elements, attributes or atomic values
  - a mixture of all of the above
• All characters appearing outside curly braces are taken literally

\[ \text{Product number} \ {\text{data}($\text{prod/number}$)} \]

• Can contain:
  – character references, predefined entity references
  – whitespace (significant if combined with other chars)

• Cannot contain:
  – unescaped < and & characters
  – unescaped curly braces (double them to escape)
Containing Other Direct Element Constructors

<html>
  <h1>Product Catalog</h1>
  <p>A <i>huge</i> list of { count(doc("cat.xml")//product) } products.</p>
</html>

- No curly braces, no special separators
• Enclosed in curly braces { and }
• Can evaluate to:
  – element nodes
    • they become children of the element
  – attribute nodes
    • they become attributes of the element
  – atomic values
    • they become character data content of the element
  – a combination of the above
Enclosed Expressions Example

for $prod in doc("cat.xml")/catalog/product
return <li>{$prod/@dept}
    {concat("num", ": ")}
    {$prod/number}</li>

attribute node becomes an attribute
atomic value becomes character data content
element node becomes child

<li dept="WMN">num:  <number>557</number></li>
<li dept="ACC">num:  <number>563</number></li>
<li dept="ACC">num:  <number>443</number></li>
<li dept="MEN">num:  <number>784</number></li>
Specifying Attributes Directly

– Attributes can also have XML-like syntax

```xml
<h1 class="itemHdr">Product Catalog</h1>,
<ul>
{for $prod in doc("cat.xml")/catalog/product
   return <li class="{$prod/@dept}">
       {data($prod/name)}
   </li>
}{/for}
</ul>
```

– Like element constructors, can contain:

  • literal content
  • enclosed expressions
    – but always evaluated to atomic values
Computed Constructors

• Allow dynamic names and values
• Useful for:
  – copying elements from the input document but making minor changes to their content
    • e.g. generically adding an `id` attribute to every element, regardless of name
  – turning content from the input document into element or attribute names
    • e.g. create an element whose name is the value of the `dept` attribute in the input document
`element product {
    attribute dept { "ACC" },
    element {concat("num","ber")} { 563 },
    element name { attribute language { "en"},
        "Ten-Gallon Hat"}
}

<product dept="ACC">
    <number>563</number>
    <name language="en">Ten-Gallon Hat</name>
</product>`
Use Case: Turning Content into Markup

```xml
for $dept in distinct-values(
  doc("cat.xml")//product/@dept)
return element {$dept}
  {doc("cat.xml")//product[@dept = $dept]/name}
```

```xml
<WMN>
  <name language="en">Linen Shirt</name>
</WMN>

<ACC>
  <name language="en">Ten-Gallon Hat</name>
  <name language="en">Golf Umbrella</name>
</ACC>

<MEN>
  <name language="en">Rugby Shirt</name>
</MEN>
```
Selecting and Filtering using FLWORS
2 Ways to Select

• Path Expressions
  – great if you just want to copy certain elements and attributes as is

• FLWOR Expressions
  – allow sorting
  – allow adding elements/attributes to results
  – more verbose, but can be clearer
Clauses of a FLWOR Expression

- **for clause**
  - iteratively binds the $\text{prod}$ variable to each item returned by a path expression.

- **let clause**
  - binds the $\text{prodDept}$ variable to the value of the dept attribute

- **where clause**
  - selects nodes whose dept attribute is equal to "WMN" or "ACC"

- **return clause**
  - returns the name children of the selected nodes

```xml
for $\text{prod}$ in doc("cat.xml")///product
let $\text{prodDept} := $\text{prod}/@dept
where $\text{prodDept} = "\text{ACC}" or $\text{prodDept} = "\text{WMN}"
return $\text{prod}/name
```
for Clauses

• Iteratively binds the variable to each item returned by the `in` expression
• The rest of the expression is evaluated once for each item returned
• Multiple `for` clauses are allowed in the same FLWOR

```xml
for $prod in doc("cat.xml")//product
```

expression after `in` can evaluate to any sequence
Range Expressions

- Create sequences of consecutive integers
- Use **to** keyword
  - \((1 \text{ to } 5)\) evaluates to a sequence of 1, 2, 3, 4 and 5
- Useful in **for** clauses to iterate a specific number of times

```
for $i$ in (1 to 5)
...
```

```
for $i$ in (1 to $\text{prodCount}$)
...
```
Multiple for Clauses

– Two syntaxes
  • repeat the for keyword

```xml
for $prod in doc("cat.xml")//prod
for $prc in doc("prc.xml")//price
```

• use a comma separator

```xml
for $prod in doc("cat.xml")//prod,
$prc in doc("prc.xml")//price
```
Multiple for Clause Example

– Essentially a loop within a loop
– Return clause evaluated for every combination of variable values

```plaintext
for $i$ in (1, 2)
for $j$ in (11, 12)
return <eval>i is {$i} and j is {$j}</eval>
```

```plaintext
<eval>i is 1 and j is 11</eval>
<eval>i is 1 and j is 12</eval>
<eval>i is 2 and j is 11</eval>
<eval>i is 2 and j is 12</eval>
```
Positional Variables in for Clauses

- Positional variable keeps track of the iteration number
- Use `at` keyword

```xml
for $prod at $i in doc("cat.xml")//
    product[@dept = "ACC" or @dept = "WMN"]
return <eval>{$i}. {data($prod/name)}</eval>
```

<eval>1. Linen Shirt</eval>
<eval>2. Ten-Gallon Hat</eval>
<eval>3. Golf Umbrella</eval>
**let Clauses**

- Convenient way to bind a variable
  - avoids repeating the same expression many times
- Does not result in iteration

```
for $i in (1 to 3)
  return <eval>{$i}</eval>
```

```
<eval>1</eval> <eval>2</eval> <eval>3</eval>
```

```
let $i := (1 to 3)
  return <eval>{$i}</eval>
```

```
<eval>1 2 3</eval>
```
Multiple *for* and *let* clauses

- *for* and *let* can be repeated and combined

```xml
let $catDoc := doc("cat.xml")
for $prod in $catDoc//product
let $prodDept := $prod/@dept
where $prodDept = "ACC" or $prodDept = "WMN"
return $prod/name
```
where Clause

- Used to filter results
- Can contain many subexpressions
- Evaluates to a boolean value
  - effective boolean value is used
- If true, return clause is evaluated

```xml
where $prod/number > 100
  and starts-with($prod/name, "L")
  and exists($prod/colorChoices)
  and ($prodDept="ACC" or $prodDept="WMN")
```
return Clause

• The value that is to be returned

\[
\text{for } \prod \text{ in doc("cat.xml")//product}
\]
\[
\text{return } \prod/\text{name}
\]

• Single expression only
  – can combine multiple expressions into a sequence

\[
\text{return } <a>\{i\} </a> \\
\text{ } \\
\text{<b>\{j\} </b>}
\]

\[
\text{return } (<a>\{i\} </a>, \\
\text{<b>\{j\} </b>)}
\]
Variable Binding and Referencing

- Variables are *bound* in the let/for clauses
- Once bound, variables can be *referenced* anywhere in the FLWOR

```xml
for $prod in doc("cat.xml")//product
let $prodDept := $prod/@dept
where $prodDept = "ACC"
return $prod/@Name
```

- Values cannot be changed once bound
  - e.g. no let $count := $count + 1
order by Clause

- Only way to sort results in XQuery
- Use `order by` before `return` clause
- Order by
  - atomic values, or
  - nodes that contain individual atomic values

```xml
for $item in doc("ord.xml")//item
order by $item/@dept
return $item
```
order by Options

- **Sort order**
  - ascending (default) or descending

- **Placement of empty sequence**
  - empty least or empty greatest

- Can specify multiple values to sort on

```xml
for $item in doc("ord.xml")//item
order by $item/@dept descending,
        $item/@num
return $item
```
• Not limited to a simple path expression
  – function calls

order by concat($per/lname, ', ', $per/fname)

– conditional expressions

order by (if (starts-with($title,'The '))
  then substring-after($title,'The ')
  else $title)
• The document order of a set of nodes is:
  – the document node itself
  – each element node in order of the appearance of its first tag, followed by:
    • its attribute nodes, in an implementation-defined order
    • its children (text nodes, child element nodes, comment nodes, and processing-instruction nodes in the order they appear)
When Document Order is Applied

- Certain expressions return results in document order:
  - path expressions (slash operator)
  - union, except and intersect operators
- Beware of inadvertent re-sorting

```xml
let $sortedProds := for $prod in doc("cat.xml")//product
    order by $prod/number
    return $prod
for $prodName in $sortedProds/name
return <li>{string($prodName)}</li>
```
Comparing on Document Order

• To compare nodes based on document order, use
  – \(<<\) for precedes, \(>>\) for follows

\[
(doc("text.xml")//p)[1] << (doc("text.xml")//h1)[1] \\
\]

\[
doc("text.xml")//p[. << (doc("text.xml")//h1)[1]]
\]

• Works for nodes only
  – no document order on atomic values
The Rest
Further Topics

• The rest of this XQuery course covers the following topics:
  • Grouping, Combining and Joining Results
  • Namespaces in XML and XQuery
  • Functions, Modules and Variables
  • Working with Text and Strings
  • Full-Text Searches (MarkLogic or eXist)
  • A Closer Look at Types and Schemas

• Please contact Priscilla Walmsley at pwalmsley@datypic.com to arrange training.
XQuery Resources

• The W3C recommendation:
  - http://www.w3.org/TR/xquery

• XQuery implementations
  - http://www.w3.org/XML/Query

• Reusable XQuery functions
  - http://www.xqueryfunctions.com

• xquery-talk mailing list
  - http://x-query.com/mailman/listinfo/talk

• XQuery, a book by P. Walmsley (O'Reilly 2007)
  - http://www.datypic.com/books/xquery/