Introduction to XML Processing: XPath, XSLT, XQuery

Curtis G. Pew
The University of Texas at Austin
The Natural Conference, August 2008

Introduction to XML Processing: XPath, XSLT, XQuery

ABSTRACT

XML has become a standard format for many information technology tasks. In this paper we will take a look at three closely related technologies for accessing and manipulating XML: XSLT, a language for transforming XML documents, XQuery, a language for accessing XML databases (like Software AG's Tamino), and XPath, a subset of XQuery used by XSLT and many other technologies for selecting parts of XML documents.

History and Relationships

XML (the *eXtensible Markup Language*) started out as a web-friendly simplification of SGML (*Standard Generalized Markup Language*). SGML had a standardized formatting language, DSSSL (*Document Style Semantics and Specification Language*) which was implemented as Scheme macros. (Scheme is a dialect of Lisp.) Shortly after the XML recommendation was published, the World Wide Web Consortium (W3C) began developing a formatting language for it, XSL (the *eXtensible Stylesheet Language*).

Early in its development, XSL was divided into two parts: an XML vocabulary (XSL Formatting Objects) describing page layout, formatting, and the text to include for a human-readable visual representation of a document, and another XML vocabulary (XSL Transformations, or XSLT) for converting input XML documents into the formatting object vocabulary. As it turns out, a language for transforming XML documents into other formats has many uses beyond styling the document for human consumption.

At the same time XSLT was being developed, another W3C group was working on XPointer, a way to specify fragments of an XML document in a URL. Since both XSLT and XPointer needed a way to select parts of a document, they decided to work together on that subset of their tasks, and the result was XPath.

When people started thinking about creating databases of XML documents, XPath proved a logical starting point for a query language. For example, XQL, the query language used by early versions of Tamino, was little more than XPath with a few extensions. The W3C XML Query group combined ideas from XQL and other proposed query languages, many of them strongly influenced by SQL, to come up with its final recommendation, XQuery.

By definition, XQuery 1.0 is a superset of XPath 2.0: the two recommendations were developed concurrently by the same W3C group and are published by applying special transformations to the same source documents. XSLT is a separate recommendation, but it references the XPath recommendation (XPath 1.0 for XSLT 1.0, and XPath 2.0 for XSLT 2.0) to provide the values and semantics of many of its constructs.

A brief overview of XML

If you are familiar with XML, you may wish to skip this section.

XML is a set of rules for designing text-based documents that can be processed by computer programs and, often, easily formatted for human viewing. The key construct for XML documents is called an *element*. Within a document, an element begins with a start tag and ends with an end tag; everything between the tags forms the element's *content*, which may consist of text, child elements, or a mixture of text and elements. An element *start tag* begins with a less-than sign ('<') followed by a name which identifies the element's type. It may contain name-value pairs called *attributes* and namespace declarations, and the start tag ends with a greater-than sign ('>'). An element *end tag* begins with a less-than sign followed by a slash ('</') and the element type name and concludes with a greater-than sign. Attributes consist of a name followed by an equal sign ('=') followed by a quoted text value. Here are some example elements:

```
<simple>This is a simple XML element.</simple>

<parent><child>This is a child element.</child>
  <child2>This is a second child element.</child2></parent>

<mixed>This element has <em>mixed</em> content.</mixed>
```

```
<example language='english'>This element has an attribute.
</example>

<empty desc="This element has no content."></empty>
```

There is an abbreviation for empty elements—by placing a slash at the end of the start tag you can omit the end tag. The last example could also be written:

```
<empty desc="This element has no content."/>
```

XML documents may also contain comments and processing rules, although these are not technically part of the XML. *Comments* begin with "`<!-- `" and end with "` -->`" and are intended for developers and other humans who need to view the raw XML. Here is an example:

```
<!-- Automatically generated by xyz; edit with care! -->
```

Processing rules begin with "`<?`" followed by a name and possibly data and end with "`?>`". They provide additional information for particular programs that may be processing the document. The most common processing rule is the `xml-stylesheet` rule that provides a browser with a stylesheet it may apply to the document:

```
<?xml-stylesheet type="text/css"
     href="http://example.com/styles/mystyle.css"?>
```

A well-formed XML document has a single, unique *root* element; other than comments or processing rules everything in the document must be an attribute or part of the content of the root element. Every element has a start tag and an end tag (or uses the abbreviation for empty tags described above) and elements cannot overlap: if the start tag for a particular element appears in the content of another element its end tag must also be found there. Attribute values must always occur within a pair of single or double quotes ("`'`" or '`"`'; in SGML the quotes were often optional.) The markup characters "<" and "&" may not appear in text content or attribute values; if they are needed the predefined *entities* `&lt;` and `&amp;` should be used instead. (It's also usually a good idea to use predefined entities `&gt;`,

`&apos;`, and `&quot;` for the greater-than sign, single quote, and double quote as well. These characters, and any other character, may also be represented by '&#' followed by the decimal representation of the character's Unicode code point and a semicolon or by '&#x' followed by the Unicode value in hexadecimal and a semicolon.)

XPath

The focus of the XPath language is on selecting parts of XML documents, although it does support arithmetic and logical operations. It really doesn't stand on its own—it depends on some kind of surrounding context to determine where it gets its input, what happens to its output, and other environmental issues. As an example of the later, while XPath has named variables (recognizable because they begin with a dollar sign ('`$`')) it does not provide a way to bind values to those variables.

The XPath data model

XPath's data model represents an XML document as a hierarchy of nodes; it recognizes seven different node types. Node properties, depending on the node type, may include a name, a parent, children, and a value. The node types are document nodes, element nodes, attribute nodes, text nodes, comment nodes, processing rule nodes, and namespace nodes. We will focus on the document, element, attribute, and text nodes, since the other node types are only used rarely.

At the root of the hierarchy representing an XML document is the document node, which has no name and (uniquely) no parent. It has one unique element node child, representing the root element of the document; it may also have comment node or processing rule node children if the document contains comments or processing rules before or after the root element.

Each element in the document is represented by an element node. The name of the node is the element type; for example, "`<q>data</q>`" in a document would be represented by an element node named "q". Each element node has a parent; this is the document node for the root element and the element node of the containing element for all others. An element node may have child nodes representing elements, text, comments, and processing rules directly contained by the element. It may also have attribute nodes and will have a namespace node for every namespace declaration in effect for the element. The value of an element node is all the text contained by it or its descendants, with all markup removed. For example, the value of the element node representing "`<p class="x">My dog has <em>42</em> fleas.</p>`" would be "My dog has 42 fleas."

Each attribute node has a parent, the element that it belongs to, even though attribute nodes are not considered children. Its name is the attribute name; for example, the `p` element above has an attribute named "class". The attribute node's value is the attribute value; in the example above it is "x". Attribute nodes have no children.

Text nodes represent the actual textual content of an element. In the example above, the `p` element node has two text node children with values "My dog has " and " fleas." and its `em` child has a text node child with value "42". Text nodes do not have either names or children.

XPath data types

XPath version 1.0 has four data types: boolean values, strings, numbers, and node sets. There are two boolean values, true and false. Strings are sequences of text characters. Numbers are 64-bit floating point values as specified by IEEE 759. Node sets are sets of nodes from an XML document as described above.

XPath has rules for converting booleans to numbers or strings, for converting numbers to booleans or strings, for converting strings to numbers or booleans, and for converting node sets to any other type. The number 0, zero length strings, and empty node sets convert to boolean false; any other value converts to true. Boolean false becomes the number 0 or the string "false", while boolean true becomes the number 1 or the string "true". If a string consists of decimal digits with optional sign or decimal point it is converted to the corresponding number; otherwise it becomes the special value "Not a Number". Converting a node set to a string returns the value of the first node in the set; that string may then be converted to a number.

XPath version 2 adds a "schema-aware" mode in which values may conform to any data type specifiable in the W3C's XML Schema Language (http://www.w3.org/TR/xmlschema-2/).

Path expressions

The most characteristic type of expression in XPath, the one from which it derives its name, is the *path expression*. Path expressions begin with a set of nodes (which may be empty or contain any number of nodes) and return a new set of nodes. There are two kinds of path expressions: absolute path expressions, which begin with a slash character ('`/`') and are evaluated from the document root node, and relative path expressions, which are evaluated from a node determined from the expression context.

Let's start with an example and then describe how path expressions work in more detail. Suppose we have this XML document:

```
<favorites>
  <books>
    <book by="Lewis Carroll">Alice in Wonderland</book>
    <book by="J. R. R. Tolkien">The Hobbit</book>
  </books>
  <music>
    <song by="Maurice Ravel">Bolero</song>
    <song by="J. S. Bach">Wachet Auf!</song>
  </music>
</favorites>
```

and we wish to evaluate the path expression "`/favorites/books/book[@by="Lewis Carroll"]`". Since this expression begins with a slash, it is an absolute path expression and we'll start from the document node. The next piece, "`favorites`", selects element nodes with that type, and since "favorite" is the type of the root element we select that element. After a slash we find "`books`", so we select all children of the current element of type "books". (In this case there is only one.) Then another slash and "`book`" indicating we select its children of type "book". Finally we have an expression in square brackets, which means that out of the "book" elements we've selected we only retain those that have a "by" attribute with value "Lewis Carroll".

Now the details: path expressions consist of one or more *path steps*. If it has more than one path step, slashes are used to separate the steps; steps are evaluated left-to-right. A path step has three parts: an axis, a node test, and optional filters. Two colons ('`::`') separate the axis from the node test, and any filters are enclosed in square brackets ('`[`' and '`]`'). A path step takes each node from the result of the previous step as a *context node* and and applies the step to select a new set of nodes which form the result of the step.

The *axis* tells where to look for new nodes relative to the current context node:

`self`	Only the context node itself appears on this axis
`child`	Contains all direct children of the context node
`descendant`	Contains all the children of the context node, their children, and their children, and so on recursively.

`descendant-or-self`	The union of the `self` and `descendant` nodes.
`parent`	The parent of the context node.
`ancestor`	The parent of the context node, and its parent, recursively to the document root.
`ancestor-or-self`	The union of the self and ancestor nodes.
`preceding`	Contains all the nodes (other than attributes or namespaces) that occur before the context node in the document, excluding its ancestors.
`following`	Contains all the nodes (other than attributes or namespaces) that occur after the context node in the document.
`preceding-sibling`	Contains all nodes with the same parent as the context node that appear before it in the document.
`following-sibling`	Contains all nodes with the same parent as the context node that appear after it in the document.
`attribute`	(only for element nodes) Contains all the attribute nodes belonging to the element.
`namespace`	(only for element nodes) Contains all the namespace nodes belonging to the element.

You may have noticed that our example path expression above has no axis specifications. That's because the child axis is the default (it's by far the most commonly used one) and may be omitted.

The *node test* uses node names or types to select from the nodes in the axis. The most common node test (used in all steps in our example above) is to specify a name; this means to select nodes with that name. You may also specify an asterisk ('`*`') as a node test to select all attributes or namespaces on the attribute or namespace axis or all elements on the other axes. The asterisk may also be used with a namespace prefix (for example, "`pfx:*`") to select elements or attributes belonging to a particular namespace. Finally, a set of pseudo-functions representing node types may be used. These are `node()` for selecting nodes of all types, `text()` for selecting text nodes, `comment()` for selecting comment nodes, and `processing-rule()` for selecting processing rule nodes.

Finally, a path step may have any number of *filters*. Each filter expression is evaluated in the context of each result node and only those nodes where all filters evaluate to true are included in the final result.

XPath defines several abbreviations to make common path steps simpler to code. We've already mentioned that the child axis need not be specified; an at sign ('`@`') may be used as the abbreviation for the attribute axis. A single period ('`.`') stands for the path step `self::node()` (that is, the current context node) and two periods ('`..`') represent `parent::node()`. Two slashes ('`//`') are equivalent to "`/descendant-or-self::node()/`" so an expression like "`.//abc`" means "select nodes named 'abc' that are descendants of the current node at any level." In a filter, if the expression evaluates to an integer it is compared by equality to the `position()` function that returns the node's sequence within its set, so for example "`qrs[1]`" selects the first "qrs" child of the current node.

XPath implementations

Besides the XSLT and XQuery implementations described below, many programming languages and environments have libraries or frameworks that allow programatic evaluation of XPath expressions. These include Java, .Net, ECMAScript (commonly called JavaScript), Perl, Python, Ruby, and Scheme. For example, the Ruby module for processing XML, REXML, includes an XPath class. If `el` is an object representing a node in an XML document and `path` is a string containing an XPath expression, then the Ruby expression "`REXML::XPath.match(el, path)`" will return an array of objects representing the nodes of the result set selected by evaluating `path` with `el` as the context node.

Although Natural does not support XPath, many aspects of Natural's `PARSE XML` statement are similar to XPath concepts. This statement creates a loop which iterates through every node in the document (plus element end tags) and returns up to three pieces of information. The `NAME` and `VALUE` variables mean much what they do in the XPath data model, while the `PATH` variable returns something similar to a path expression that would select the node, in some cases appended with characters representing the node type. For example if we process the XML document above with the following code:

```
PARSE XML INXML INTO PATH MYPATH NAME MYNAME VALUE MYVALUE
  DISPLAY MYPATH (AL=25) MYNAME (AL=10) MYVALUE (AL=20)
END-PARSE
```

we get the following output:

```
         MYPATH             MYNAME         MYVALUE
------------------------- ---------- --------------------

favorites                 favorites
favorites/books           books
favorites/books/book      book
favorites/books/book/@by  by         Lewis Carroll
favorites/books/book/$               Alice in Wonderland
favorites/books/book//    book
favorites/books/book      book
favorites/books/book/@by  by         J. R. R. Tolkien
favorites/books/book/$               The Hobbit
favorites/books/book//    book
favorites/books//         books
favorites/music           music
favorites/music/song      song
favorites/music/song/@by  by         Maurice Ravel
favorites/music/song/$               Bolero
favorites/music/song//    song
favorites/music/song      song
favorites/music/song/@by  by         J. S. Bach
favorites/music/song/$               Wachet Auf!
favorites/music/song//    song
favorites/music//         music
favorites//               favorites
```

XSLT

XSLT was designed for transforming XML documents from one format to another, although it can be used with non-XML input or output (HTML and plain text output is explicitly supported.) A program in XSLT is called a stylesheet and is an XML document in its own right; the root element is `xsl:stylesheet` or `xsl:transform`. An XSLT processor takes an input XML document and a stylesheet document and generates an abstract tree representing a new document. How the input and stylesheet documents are specified depends on the implementation.

XSLT TEMPLATES

The key idea for XSLT is the concept of templates, represented by `xsl:template` elements as direct children of the `xsl:stylesheet` root. XSLT templates function similarly to procedures or subroutines in other languages. When a template is invoked, its contents describe output or computations to be performed: it may contain elements that do not belong to the XSLT namespace, which are copied to the output tree, and it may contain directive elements that do belong to the XSLT namespace which describe further operations for the processor to perform. Templates may be called by name or, more commonly, may be matched by a source document node. The `match` attribute of the `xsl:template` element controls node matching. The value of this attribute is a *pattern*, an XPath path expression that only uses the child, descendant, and attribute axes. A node matches a pattern if it would be in the result of the path expression evaluated with any node in its document. The matched node is considered the context node when evaluating XPath expressions inside the template.

Within an `xsl:template` element the `xsl:apply-templates` element directs the processor to match nodes against the templates in the stylesheet. It has a `select` attribute containing an XPath expression specifying which nodes to try to match; if omitted it defaults to “`select='*'`”. When the XSLT processor begins transforming a document it starts by trying to match the document node to a template. If no template in the stylesheet matches it uses a default template that effectively says to try to match its child element. Similarly, if an element node does not match any templates in the document the default is to apply templates to its element and text node children. The default for a text node is to copy its value to the output.

Another important XSLT directive element is `xsl:value-of`. This element also has a `select` attribute which is evaluated as an XPath expression, converted to a string, and copied to the output. The `xsl:variable` and `xsl:parameter` elements bind values to XPath variables that can then be used in other XPath expressions within the template.

Here's an example of an XSLT stylesheet for converting our sample XML document to HTML:

```
<xsl:stylesheet version="1.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 xmlns="http://www.w3.org/1999/xhtml">

<xsl:template match="/favorites">
<html lang="en-US">
  <head><title>Favorites</title></head><body>
  <h1>Favorites</h1>
  <xsl:apply-templates/>
</body></html>
</xsl:template>

<xsl:template match="books|music">
  <h2><xsl:value-of select="name()"/></h2>
  <ul>
    <xsl:apply-templates/>
  </ul>
</xsl:template>

<xsl:template match="book|song">
  <li>
    <xsl:value-of select="."/>
    <xsl:text> by </xsl:text>
    <xsl:value-of select="@by"/>
  </li>
</xsl:template>

</xsl:stylesheet>
```

The first template matches the root element of the input document and creates the basic structure of the output HTML document, but where most of the body of the HTML would go there is an `xsl:apply-templates` element that will select its children. The next template matches those children, creates a header, and creates a list element that again has its contents supplied by `xsl:apply-templates`. The last template matches the low-level elements and generates list entries by extracting the value of the source element and its attribute.

XSLT IMPLEMENTATIONS

All modern web browsers include an implementation of XSLT which can be invoked by including an `xml-stylesheet` processing rule at the beginning of an XML document. Microsoft makes the XSLT processor used by Internet Explorer available to any Windows application via the Microsoft XML Core Services interface. Most Linux distributions, and many other *nix systems, come with libxslt, an embeddable C implementation that forms a part of the Gnome environment. XT by James Clark is a stand-alone XSLT processor written in C. There are several XSLT processors written in Java; the most prominent are Michael Kay's Saxon and Xalan from the Apache Foundation. (T here is also a C version of Xalan.)

XQuery

XQuery was designed to provide for XML databases and document collections what SQL does for relational data stores. It is a superset of XPath: every valid XPath expression is also a valid XQuery expression. XQuery also fills in most of the pieces XPath lacks as a stand-alone language: it provides a way to define user functions and libraries of functions and a way to bind values to variables.

The most characteristic type of XQuery expression is known as a FLWOR (pronounced "flower") expression from the five kinds of clauses it may contain. The `for` and `let` clauses bind values to variables, the optional `where` clause applies filters to narrow down the results, the optional `order by` clause sorts the results, and the `return` clause constructs the results. Here's a simple example:

```
for $book in document('sample.xml')/favorites/books/book
where $book/@by = "Lewis Carroll"
return $book
```

This FLWOR expression is essentially equivalent to the sample path expression we used above, and we just as easily could have written

```
document('sample.xml')/favorites/books/book[@by = "Lewis
Carroll"]
```

FLWOR expressions can get a lot more complicated, though, and can express things that a simple path expression can't. For example, this expression performs the equivalent of a join in SQL:

```
for $book in input()/book,
    $author in input()/author
let $auth_name := $author/name
where $books/author/name = $auth_name
return <title-author-birth>
{$book/title, $auth_name, $author/birth-date}
</title-author-birth>
```

The difference between `for` and `let` is that with `for` the variable is bound to each item in the defining sequence in turn, while `let` binds the variable to the whole sequence. For example, suppose our collection "x" contains three documents: "`<x>a</x>`", "`<x>b</x>`", and "`<x>c</x>`". Then the query:

```
for $x in collection('x')/x
return <result>{$x}</result>
```

would return "`<result><x>a</x></result> <result><x>b</x></result> <result><x>c</x></result>`" while the query:

```
let $x := collection('x')/x
return <result>{$x}</result>
```

would return "`<result><x>a</x> <x>b</x> <x>c</x></result>`".

XQuery implementations

XQuery was originally developed for native XML databases like Software AG's Tamino, MarkLogic's MarkLogic Server, and the open-source eXist database, but many relational database vendors now provide XQuery access in their products; these include IBM's DB2, Oracle, and Microsoft's SQL Server. There are also stand-alone implementations of XQuery: Michael Kay's Saxon (version 7 or higher), Galax (open source in OCAML), and Qexo (open source; compiles to Java bytecode) are some examples.

XQuery vs. XSLT

XQuery and XSLT contain a lot of common functionality, and many tasks can be solved equally easily in both languages. While both languages have partisans who advocate eschewing one language in favor of the other, their focus differs enough that having both available is a good thing. XSLT excels when all or most of the input comes from a single document and when the output is intended for human consumption, while the "sweet spot" for XQuery is retrieving fragments from multiple documents. It often makes sense to use XQuery to collect data into a single document and then use an XSLT stylesheet to format the result; Tamino is distributed with a standard server extension that supports this.

Conclusions

XML has become a common format for storing and exchanging data. XPath, XSLT, and XQuery form a closely related trio of technologies for manipulating XML data. XPath in particular is used in many XML processing contexts. As we've seen in the case of Natural, other environments often borrow ideas from these technologies when processing XML, so learning them can be valuable even if you never end up using them.