XQuery: An Introduction to "SQL" for XML

Rob Rucker

2009-12-10

[Note: this was first written in 2005 using an proprietary XML database called "XStreamDB". Currently, in 2009, I have started using eXist, an open source DB. The examples are from the former DB but the ideas are the same.

XQuery in Two Paragraphs

The purpose of this introduction is to describe and give examples of a new XML based language called *XQuery*. You can initially think of it as the "SQL" for XML documents, in terms of its comprehensiveness and capabilities. The XQuery language is intended to provide an interface to any XML document (or its representation) and allow its' manipulation and querying regardless of its type of XML structure. From the unstructured, strictly textual document, to the precisely specified data types, perhaps extracted from a relational database, XQuery is designed to allow both querying as well as transformations of the original document or data stream. It is designed to be able to search, transform, construct, and retrieve all of the various nodes, such as elements, attributes, and primitives, found in XML documents or XML based data streams.

To do all this, XQuery builds on XML standards as well as extends and incorporates several other XML languages such as XPath and XML-Schema. XQuery extends XPath version 2.0 as its search sublanguage, and uses XML-Schema as its type system. XPath is used as a sublanguage to locate items in a source XML stream while XML-Schema provides type and structure information to the XQuery processor. As an aid to finding both static and dynamic type errors, XQuery makes extensive use of both built-in typing and user defined typing incorporated within XML-Schema, DTDs, or perhaps other processors such as Relax/NG.

What You Are In For?

I will describe some of the language's theoretical features as well as some practical examples using XQuery to search a "live" XML database. I will assume the reader is familiar with basic XML syntax as well as having had some exposure to XPath and XML-Schema. Most of the material here is understandable on its own, but the reader may want to refer to other tutorials in this series or consult the very extensive XML literature found in books and on the web. (See "Bibliography" on page 13.)

All the queries in this tutorial have been done using a “Native XML” database. The result is then realistic but it does mean that some of the syntax is vendor specific. I will point this out as I go along. QuickStart to XQuery

I think that the best way to explain something is to show an example, so I have cooked one up for you immediately. Please realize that some parts of this example may not be totally understandable as yet, but your understanding will deepen as you progress through this document, as well as studying related materials.

An XQuery Snippet

To whet your curiosity and start you thinking of questions you may want to ask, here is an easy query using XQuery against an XML document. The query is shown below and labeled as Example B1: The Query. The document to be queried is stored in a native XML Database that will be described later (*See “A Native XML Database” on page 7.*) The document the query is searching against is shown as “Example C1: books.xml” on page 4. This XQuery will return all the titles from all of the books held in a particular collection of documents, with that collection being held in a particular database.

For now, simply be aware that this query specifies the *database* to query against, *BookDB,* and then, within that, the *root*, called *BooksRoot*, which will further narrow down the document collection to interrogate. The syntax bracketed by “(:” and “:)” encloses an internal XQuery comment, which won’t appear in the output stream.

Finally, the second line of the query “//book/title” is an *XPath* expression that instructs the XQuery processor to search within the BooksRoot all the documents found in that database, within that root collection, then find all books, no matter where they are in the hierarchy and then return the title found under each such book.

In this particular case, the database, *BookDB*, had only the one root, *BooksRoot*, and that root had only one document in its collection, `books.xml`. The query searched this document’s representation and extracted all of the `title` elements that it found within each `book` element. That sequence of element nodes constitutes a *value* and was returned within a vendor specific “container” element, `Resultset`.

Example B1: The Query

```
Root(“/BookDB/BooksRoot”) (:vendor specific entry point to a document collection:)
//book/title
```

Example B2: The Result

```
<Resultset> {: vendor specific container element:)
<title>
      TCP/IP Illustrated
   </title>
<title>
      Advanced Programming in the UNIX Environment
   </title>
<title>
      Data on the Web
   </title>
<title>
      The Economics of Technology and Content for Digital
   </title>
```

```
</Resultset>
```

For the reader who is curious at this point as to how this query and result were obtained, here is the sequence of steps:

- The `books.xml` document was typed using an XML editor (any editor will do, but some are easier to use than others)
- The native XML database management system that I am using (XStreamDB), was started.
- A GUI program interface (Explorer) from the database vendor allowed the creation of a database that I have called *MyBooksDB* and a sub-object container called a "root". I named this root object *BooksRoot*. (This subobject is designed to hold *collections* of documents. In this example though, I have only inserted one document.)
- I then uploaded the `books.xml` document into this root object. At this point, the `books.xml` document is available for XQuery manipulation.
- The GUI interface also allows XQueries to be written in a windows environment and the result to be captured and saved in a file. That was the procedure adopted here.

C) The Data Model for XQuery - A Sequence of Nodes and/or Primitives

Just as Structured Query Language (SQL) relies on a specific data model, a relational (table), on which to perform queries, XQuery has its own data model to perform queries over. XQueries' data model is based on a sequence of nodes and/or primitives. I will go into detail on both of these ideas later in this manual but for now think of the nodes as coming from a source document that has been pre-processed by a *parser* into a tree structure. That processor has read the XML document and produced a tree that contains labeled *nodes*, that are of types: document, element, attribute, text, namespace, processing-instructions, and comments. Additionally, the nodes are labeled with any type information that the processor has been provided with such as an associated XML-Schema or DTD. (Note: the information at this stage is called the Post Schema Validation Infoset (PSVI))

The data model contains selections from this tree as well as possibly dynamically constructed nodes that you can add to the model. Additionally, the data model has been extended to accommodate primitives.The primitives allowed here correspond to the simple types found in XML-Schema. As for these primitives, think of them as single values that are numerics such as integers, floats, doubles; dates, strings, some legacy types, and a few miscellaneous ones like booleans, or URIs. (Note that restrictions of these primitive types are also considered simple types. That is, perhaps a user-coded "BoxWeight" type, restricts the weight to be between 2 and 25 pounds. This type is also considered to be a simple type since it's a restriction of a primitive type, "xs:integer".

> Combining these two types of components, selected/constructed nodes, and/or primitives, the data model of XQuery is described as: *a sequence of nodes and/or primitives*. More compactly, the data model of XQuery is a *sequence of items* where an item can be either a node or a primitive. A sequence of items is then called a *value*. So finally, a shorthand description of the XQuery data model is to call it a "*value"*.

For a possible memory aid in learning these types, check out the section "A Biological Metaphor for the XQuery Datamodel. A "Spiced Ant" Line" on page 12.

XQuery Transforms from Datamodels back to Datamodels

The second analogy with SQL is the idea of *closure*. In SQL, you may remember, SQL operates on table(s) and produces a table result. This property of having the operation return the same output type as input type, is called *closure* and allows sequential or "pipelined" processing. Since the datamodel stays the same, an output can feed another input. XQuery does the same thing with its' datamodel. XQuery starts with its' data model as described above, and returns a datamodel of the same type, closure again. That is, XQuery works on a sequence of nodes and/or primitives and returns another model of the same type.

A BaseLine XML Document

The basic XML document that will be used to illustrate XQueries in this tutorial is shown next as Example C1. This example matches pretty much the XQuery Use Case examples so that you can compare those outcomes with the ones obtained below.

Example C1: books.xml

```
<?xml version=”1.0” encoding=”UTF-8”?>
<!-- books.xml
* Design intent: baseline example for XQuery
*-->
<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>
   <book year=”1992”>
      <title>Advanced Programming in the UNIX Environment</title>
      <author>
         <last>Stevens</last>
         <first>W.</first>
      </author>
      <publisher>Addison-Wesley</publisher>
      <price>65.95</price>
   </book>
   <book year=”2000”>
      <title>Data on the Web</title>
      <author>
         <last>Abiteboul</last>
         <first>Serge</first>
      </author>
      <author>
         <last>Buneman</last>
         <first>Peter</first>
      </author>
      <author>
         <last>Suciu</last>
         <first>Dan</first>
      </author>
      <publisher>Morgan Kaufman Publishers</publisher>
      <price>65.95</price>
```

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

D) Basic DataTypes

The basic data types used in XQuery include those of XML-Schema. In addition, XQUery has a few of its own. I will write them down here for convenience.

XML-Schema Primitive types

- numeric - integer, decimal, float, double, positiveInteger, negativeInteger, nonNegativeInteger, nonPositiveInteger, short, byte
- gDay, gDuration, . . .

E) XPath Expressions

XQuery is a superset of XPath 2.0 and so knowledge of XPath will be essential for any progress to be made in XQuery. What I will do here is give a few examples of XPath expressions that will be used within typical queries later in this document. For a more detailed discussion of XPath the reader is referred to "XSLT & XPath: An Introduction and Tutorial" in this series or consult the books and web sites listed in the bibliography section on page 13.

Next I am going to show how to do an actual query using a native XML database. The database creation and document collection syntax will differ slightly from vendor to vendor however the basic XQuery language won't. (This is analogous to the way SQL doesn't specify how a database is to be set up or named. The ***Data Definition Language is not specified.)

Suppose I want to search the `books.xml`document of Example C1 for some information about authors or perhaps book titles. Before I can implement a query for this information, I need to give the processor two higher levels of context regarding the location of this file as follows:

- Declare which collection of documents that `books.xml` may be found in. This is called the Root and I have set up and given this collection the name `BooksRoot.`
- Declare in what database the Root collection of documents is to be found. I have set up and named a database I called `MyBooksDB.`

Operationally then, what I did was create the database, `MyBooksDB,` then added a "Root" to that database that can contain collections of documents. I called this root object `BooksRoot.` Finally I uploaded documents into this root container. In this particular case, I uploaded only one document, `books.xml.` To summarize to this point, the native XML base that I am using specifies the syntax to carry out this initial context setting as:

Example E1 Query:

`Root("/MyBooksDB/BooksRoot")` `(:vendor specific entry point to a document collection:)`

This query will return the top level element in the book.xml document, namely `bib`. Notice that returning this node has the effect of returning all of its contained nodes as well, resulting in the whole set of books being returned. (The vendor software has supplied a top level utility element `Resultset` that acts as a container for the returned elements.)

Example E2 Result

```
<Resultset> {: vendor specific container element :)
<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>
<!-- *** the other three books from the XML document were displayed here *** -->
</Resultset>
```

F) FLWOR Expressions

When it comes to manipulating input sequences of nodes and primitives, the operations whose first letters correspond to FLWOR (pronounced 'flower'), allow powerful expressions to be invoked. Remember that the data model of XQuery is a sequences of nodes and/or primitives. A more abstract designation for this choice is to call each member an *item*. The reader might also want to keep in mind that a *value* is a sequence, each member of which is an *item*. Another term that will come up in these explanations below is *tuple*: A tuple is a pair consisting of a variable and an associated item, written as {variable, item}. A tuple represents a *binding* between a variable and an item.

The FLWOR expressions are:

- `for` clauses- associate one or more variables to expressions. Expressions are evaluated by the processor and result in a sequence of items. The `for` clause then binds the variable to each item in turn, thus creating a stream of *tuples*.
- `let` clauses- bind variables to the result of an expression, and if used in conjunction with a `for` clause, add these tuples to the `for` clause tuples.
- `where` clauses-

- `order by` clauses-
- `return` clauses-

So, just as SQL specifies what table(s) are to be worked on by specifying table names in the Select clause, so does XQuery allow you to specify *values* to be worked on by specifying a *binding* between variables and values to be worked on. This capability is available from the *For* and *Let* expressions.

These (variable, value) pairs then comprise the input to further processing using combinations of the rest of the expressions.

The reader may detect in these expressions the capabilities of the SQL language with its Select, Where, and Order By statements. The idea here is that the designers of XQuery were heavily influenced by the SQL paradigm and intended to emulate as well as extend it with capabilities focused on hierarchy and order. Hierarchy and order are concepts that are not applicable to the relational structure and so with these additions, XQuery has surpassed SQL's capability as a manipulation language.

Some FLWOR Examples

Example F1: Query Using For

This next query *binds* the variable $b to each of the book nodes. That is, the expression "/bib/book" returns a set of four nodes, the book element nodes. Then, the $b is bound to each of these items in turn, resulting in a tuple series that conceptually looks like: {$b, firstbooknode}, {$b, secondbooknode}, {$b, thirdbooknode}, {$b, fourthbooknode}. In other words, the XPath expression "/bin/book" returns a set of book element nodes and the $b is bound to each of them in turn. Below is shown a truncated output. The result returns all of the books. Compare this output with that of Example E2 which also returned all of the books, but they were contained within a higher level element, the `bib` element. The syntax shown below directs the processor to "bind" the $b variable to the evaluated expression that follows the keyword "in". That is, up to the keyword "return". This last keyword, then writes out the tuples to a result tree, as shown in Example F2: Result.

```
for $b in
Root("/MyBooks/BooksRoot")/bib/book
return $b
```

Example F2: Result

```
<Resultset> {: vendor specific container element:)
<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>
<!-- continued with the other three books *** >
```

Example F3: Query Using For and Primitive Data

This query points out the *primitive* items within the XQuery model. In this case I am going to set up a data model that isn't a set of nodes from an input tree, but rather is a sequence of primitives, the integers, 1, 2, and 3. Remember that the data model allows both nodes *and* primitives. This example illustrates a couple of points. First, XQuery allows a sequence of integers to be an instance of the data model and sets up the syntax to implement that. Second, the `return` is a clause that is building the output from our literal specification of XML elements. Note also that the XQuery system will simply take our explicitly written <tuple> and <i> element tags, pass them to the output and insert the sequenced values of $i, where $i has been bound to each item associated with the expression (1,2,3). Note that to evaluate the $i variable, I had to enclose it within braces, "{", "}" which is the evaluation syntax for XQuery.

Here is a case where the data model is a sequence of items and the items are the integers 1, 2, and 3. Further, the `for` clause associated (bound) the variable $i to each of these items of the expression (1,2,3) in turn. This resulted in a sequence of three tuples: {$i,1}, {$i,2}, {$i,3}. Finally, I used the return clause to construct a literal output of elements together with an evaluation of the $i variable.

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

Example F4 Result

```
<Resultset> {: vendor specific container element:)
<tuple><i>1</i></tuple>
<tuple><i>2</i></tuple>
<tuple><i>3</i></tuple>
</Resultset>
>
```

Example F5: Query Using For and Multiple Let Clauses

This next query demonstrates the aggregation of the `let` clause's bindings with those of a `for` clause. Note that the single $k binding with "ab cd", is associated with each of the `for` tuples as is the single $m binding. Notice that the "let" tuples are bound to each item in the "for" clause. That is, the first "for" item, which is the integer "1", is associated with "ab cd" and "x". Exactly the same process associates the "2" with "ab cd" and "x".

```
for $i in (1,2)
let $k:= "ab cd"
let $m:= "x"
return <tuple><i>{$i}</i><k>{$k}</k><m>{$m}</m></tuple>
```

Example F6: Result

```
<Resultset> {: vendor specific container element :)
<tuple><i>1</i><k>ab cd</k><m>x</m></tuple>
<tuple><i>2</i><k>ab cd</k><m>x</m></tuple>
</Resultset>
```

Example F7: Query Using Both For and Let

This example uses the `for` and `let` clauses to illustrate how the `let` clause can associate a complicated expression with the `let` variable, that can be used later.

```
for $b in
Root("/MyBooksDB/BooksRoot") (:vendor specific entry point to documents:)
/bib/book
let c:= $b/title
return <book> { $c} </book>
```

Example F8: Result

```
<Resultset> {: vendor specific container element :)
<book><title>TCP/IP Illustrated</title></book>
<book><title>Advanced Programming in the UNIX Environment</title></book>
<book><title>Data on the Web</title></book>
<book><title>The Economics of Technology and Content for Digital</title>
</book>
</Resultset>
```

Example F9 Query

```
for $b in
Root("/MyBooksDB/BooksRoot")(:vendor specific entry point to documents:)
/bib/book
let $c := $b/title
return <book> {$c, <count>{count($b/author)}</count> }</book>
```

Example F10 Result

```
<Resultset> (:vendor specific container element :)
<book><title>TCP/IP Illustrated</title><count>1</count></book>
<book><title>Advanced Programming in the UNIX Environment</title><count>1
</count></book>
<book><title>Data on the Web</title><count>3</count></book>
<book><title>The Economics of Technology and Content for Digital</title>
<count>0</count></book>
</Resultset>
```

Joins and Cross Products

When we deal with multiple "for" clauses we can emulate the "join" features of the relational world, if we choose to. That is, we can combine data from one structure with another, matching on some common feature. This next example searches two documents that are in the *MyBooksDB:BooksRoot* management container. Up to this point there has only been one document in the collection, books.xml. Now I have placed a second document in this collection called `reviews.xml`, shown next.

Example G1: reviews.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- reviews.xml
```

```
* Design intent: example for “XQuery: An  Introduction”
* to illustrate joins together with books.xml
* 2004-02-16 r.r
-->
<reviews>
<review>
<reviewer>r.r.</reviewer>
<title>TCP/IP Illustrated</title>
<comments>Great Technical Content</comments>
</review>
<review>
<reviewer>r.r.</reviewer>
<title>Data on the Web</title><comments>Good Overview</comments>
</review>
</reviews>
```

Making the (Inner) Join Code

The code shown next matches the “title” text from the `books.xml` with the title from the `reviews.xml` file. Given this match, then the associated review is returned. This is an example of an inner join since for those books that don’t have a review (there are two of these), no elements are returned.

Example F2: Inner Join Query

```
for $title in Root("/MyBooksDB/BooksRoot")/bib//title
for $review in Root(“/MyBooksDB/BooksRoot")/reviews/review
where $title = $review/title
return <reviews> {$title,$review/comments} </reviews>
```

Example F3: Result of Inner Join

```
<Resultset> (:vendor specific container element :)
<reviews>
      <title>
         TCP/IP Illustrated
      </title>
      <comments>
         Great Technical Content
      </comments>
   </reviews>
<reviews>
      <title>
         Data on the Web
      </title>
      <comments>
         Good Overview
      </comments>
   </reviews>
</Resultset>
```

Making an (Left) Outer Join

The previous query returned only books that had reviews. If we would like to return all books whether or not they have a review we need a *left outer join* as shown next. Note that the result returns all four books that are in the `books.xml` and includes comments where they are available in the `reviews.xml` file.

Example F5: Query Outer Join

```
for $title in Root("/MyBooksDB/BooksRoot")/bib//title
return
<reviews>
{$title}
{
for $review in Root("/MyBooksDB/BooksRoot")/reviews/review
where $title = $review/title
return $review/comments, $review/reviewer
}
 </reviews>
```

Example F6: Result Outer Join

```
<Resultset>
<reviews>
      <title>
         TCP/IP Illustrated
      </title>
      <comments>
         Great Technical Content
      </comments>
   </reviews>
<reviews>
      <title>
         Advanced Programming in the UNIX Environment
      </title>
   </reviews>
<reviews>
      <title>
         Data on the Web
      </title>
      <comments>
         Good Overview
      </comments>
   </reviews>
<reviews>
      <title>
         The Economics of Technology and Content for Digital
      </title>
   </reviews>
</Resultset>
```

Quantification, Existential and Otherwise

In first order predicate logic, there are two major quantifiers, existential and universal. The first one asserts that something exists and the second says that everything exists. Both statements are restricted to a particular *context*. In our case this context will be the collection of documents that comprise our search space. In other words we will be dealing with quantification that is restricted to a document set. An example will help here: ******* not supported yet***********

Example G Query Existential Quantification

```
for $b in Root("/MyBooksDB/BooksRoot")/bib/book
where some $a in $b/author
   satisfies ($a/last="Stevens" )
return
$b/title
```

A Native XML Database

The XML database chosen for this tutorial is a product from Bluestream software headquartered in Vancouver Canada.It is called XStreamDB, version 3.2. The reason to call it a native XML database is that it exposes a uniform XML interface to queries. This interface is expressed using XQuery. So far as possible, all of the data manipulations and definitions use XML and XQuery syntax. This is in distinction to some XML bases that have separate languages for the definition parts of their overall system.

A Biological Metaphor for the XQuery Datamodel. A “Spiced Ant” Line

For those of us who appreciate a little more “life” in a technical description, let me offer this as a metaphor to the XQuery data model. Remember, the XQuery datamodel is a *sequence*. It is a sequence, that is, a linear series of *items,* where an item can be either a primitive (simpleType) or a node. So what I am going to do here is list all the possible types and then put together an acronym that is suggestive of the linearity and the recursiveness of an item in a sequence. Let me list the possibilities that an *item* can be:

- SimpleType - this is one of the XML-Schemas built-in 44 types or a restriction of one of those, or one of five more special types added by the XQuery spec. I will refer to all of these as a *simpleType*.
  Example: xs:integer,
- Processing-Instruction - this is one of the node types that specifies a processing instruction
  Example: <?xml-stylesheet href=”myxsltsheet” type=”text/xsl” ?>
- Comment - A comment node. Note that XQuery had a comment syntax that uses the “(:” and “:)” symbols to bracket a comment. These XQuery comments are internal to the data model and are not available for output. Regular XML comments are also possible and are available for output.
  Example: <!-- An XML comment goes here and can be output-->
  Example: (: An XQuery comment goes here and cannot be output :)
- Element - An element node. The example shows a `name` element, that happens to have sub elements of its own.
  Example: <name><first>Jo</first><last>Kim</last></name>
- Document - This is the (invisible) document node and conceptually contains the whole document. Often called the *Document Root*. The example shows a symbol often used to indicate the document root.
  Example: /
- Attribute - The attribute node provides metadata about an element and is associated with an element (although not considered its child). Below, the attribute is named *status*, with value of “checked-out”. Its associated element is `book` and will be so treated by the document processor.
  Example: <book status=”checked-out”> . . .
- Namespace - This node is associated with every element and attribute that lies within the scope of a namespace declaration.
  Example: “http://www.w3.org/1999/xhtml”

- Text - This is the text node and is used to contain all of the character data outside of "tags". Example: <title>Fractals or Everyone</title>. The text node would contain the text string "Fractals for Everyone".

If you read down this list, you will see: *Spiced Ant.* This is a handy mnemonic and when combined with the idea of "order" in the form of a line, I have the visual metaphor of a *Line of Spiced Ants*. The metaphor continues when I pick out one of these spiced ants and discover that that ant can again be in any one of these eight categories.

Glossary

term	definition
for	A clause that binds a variable to each item in an associated, evaluated expression. The result is a series of tuples of the form {variable, item}. This clause can have multiple variables, in which case there will be multiple series of tuples.
let	A clause that binds a variable to the whole result of an evaluated expression. If used in conjunction with a `for` clause, then the `let` bindings are added to those of the `for`.
where	A clause that filters tuples according to a condition.
order by	A clause that sorts the tuples in a tuple series
return	A clause that constructs the result of a FLWOR expression. The construction uses a generated tuple.
tuple	An ordered pair of variable, item bindings, written here as {variable, item}.
binding	An association between a variable and an item. This is also called a tuple (see tuple)

Bibliography

Here are references to tutorials in this series as well as other books and web sites I have found useful in writing this tutorial and applying these ideas to practical projects.

Chamberlin, D. et al, *XQuery From the Experts*, 2004, Addison-Wesley, isbn 0-321-18060-7.

www.w3c.org is the definitive source for the XML specifications discussed in this tutorial.

Rucker, R. XSLT & XPath, An Introduction and Tutorial, 2003, Internal Publication.

Rucker, R. XML-Schema, An Introduction and Tutorial, 2002, Internal Publication.

Rucker, R. XQuery, An Introduction and Tutorial, 2004, Internal Publication.

Extra Verbiage

**********************Extra Verbiage****************************

containing numerical based form data.

Note: **Simple types can also be restrictions of the primitives as found in XML-Schema. The result of these extensions is that the XQuery data model can be thought of as a sequence of nodes and/or primitives. A single node or a simple type is called an item and so the sequences can be described as sequences of items.

That model Equally importantly, just as SQL transforms relational table into relational tables, so does XQuery transform its data model into another instance of its data model.

The model that XQuery

Finally, the syntax for specifying the database and the root collection the `books.xml` document is to be found in is written as:

Note that this will work because I have already set up a database

as what database it is to be found in, namely `MyBooksDB,`and then, within that, I must specify the root of the document collection I wish to search, namely, `BooksRoot.`

Let me further assume that I have already specified that the search is to begin in the *MyBooks* database and within that, the *BooksRoot* root. That specification is entered as `Root("MyBooksDB:BooksRoot")`. This places us ready to search all the documents, (there is only one in this case), at this level.

Here are some common XPath expressions that will be used in subsequent XQueries:

Also note that the work “value” is used in a more general sense as described next).

XQuery is a query language designed to be flexible enough to probe, transform, and construct, all the various structural and content to be found in XML document representations.

A Biological Metaphor for the XQuery Data Model - Like Peas in a Pod ***** I think this one is too limited, see the spiced ant line ******

Consider the common vegetable, the garden variety pea bush. Further, consider the pea pod and the pea’s within it. The linear arrangement of the peas in the pod comprise a sequence, just as the XQuery “value” is a sequence. Further, let me suggest a memory device, where P= Primitive, E=Element, A= Attribute. Each pea in the pod can be one of a Primitive, an Element, or an Attribute. So, the pod corresponds to the sequence or “value” of the data model while the individual peas inside the pod correspond to one of the main components of a data model, one of a Primitive, Element, of Attribute.

To be even more outrageous, let me propose that there is a “pea processor” that can process pods from a number of sources as well as return one or more pods as well as insert peas into an existant pods during processing. (Note that our pea processor can’t insert a “pod” into another pod, but it can insert raw peas. (This parallels the XQuery restriction whereby a sequence can contain nodes or primitives, but not embedded sequences)

Note: You might keep in mind though, that after all is said and done, the main building blocks of the the document and the resultant data model, consist of *Primitives*, *Elements*, and *Attributes*. That is, most XQuery "*values*" are made up of sequences of Primitives, Elements, or Attributes.

Consider all of the datatypes that can appear as a member

The range of the possibilities of XQuery will be discussed and illustrated below.

***Notes to Rob -- I will do the XML handbook using the idea of multiple tutorials as below. Also the idea of an XXXX in one paragraph followed by a snippet will be standard

(*** see other tutorials and use similar heads**)

****