XQuery 1.0: Primer

by Juliane Harbarth, Technical Consultant R&D Technology, Software AG

Abstract

This document is intended to provide an easily readable description of the XQuery XML Query language, and is oriented towards quickly understanding how to create queries. The normative description is mostly provided in the following W3C Working Drafts [*MOS]:

- XQuery 1.0: An XML Query Language,
- XQuery 1.0 and XPath 2.0 Data Model,
- XQuery 1.0 and XPath 2.0 Functions and Operators.

This primer describes the language features through a contiguous example that roughly sketches an XML system for scheduling concerts. The example is complemented by extensive references to the normative texts.

Status of this Document

Since all the normative texts this primer is introducing are still Working Drafts, this text is also work in progress and will have to be adapted to future updates in the drafts. Even in its final version it will probably not be very exhaustive with respect to user-defined types and XQuery modules, because these are topics I have no experiences in. Otherwise it is supposed to cover the whole of XQuery. If a section does not yet cover what it is supposed to deal with in the end, the list of topics still to be discussed is added as an unordered list to the end of the section.

Table of Contents

1. Introduction

This document, XQuery 1.0: Primer, provides an easily approachable description of the XQuery query language, and should be used alongside the formal description of the language contained mainly in the Working Draft XQuery 1.0: An XML Query Language. The intended audience is application developers querying XML data stored in files and/or XML databases. The text assumes that you have a basic understanding of XML 1.0, XML-Namespaces and XML Schema. A basic understanding of XPath 1.0 is also helpful, but not required.

Section 2 covers the basic mechanisms of selecting nodes from XML documents using XQuery's path expressions. The first half of this section covers roughly what is contained in the XQuery-preceding XPath 1.0 Recommendation concerning the abbreviated syntax plus XQuery's doc() function. For those being familiar with XPath 1.0, it is still not recommended to skip this part, since it introduces the example and furthermore contains lots of remarks pertaining to XQuery rather than XPath 1.0. The second part deals with XQuery's concept of document collections and especially the two XQuery built-in functions input() and collection().

Section 3 contains the introduction to the most important mechanisms that XQuery 1.0 adds on top of path expressions. It describes how to use XQuery's FLWOR expressions to express joins and explains how to construct XQuery results using node generation. This is considered the most important part of the primer.

Section 4 contains an overview of the XQuery built-in functions. This forms a whole chapter since there are very many built-in functions in XQuery and consequently they are presented in a separate draft, i.e. XQuery 1.0 and XPath 2.0 Functions and Operators.

Section 5 covers all the rest of XQuery, i.e. the purpose of the XQuery prolog, an introduction to the type system, an overview of the XQuery operators, etc.

2. Node Selection with Path Expressions

This section describes how to select nodes from XML documents using path expressions. Although XQuery in principal supports both the abbreviated and non-abbreviated syntax of path expressions, we'll only deal with abbreviated path expressions until further notice. We exemplify the selection techniques using the sample instance m1p.xml. The instance represents an item of a musical library. Such an item usually comprises one-to-many pieces which again consist of one-to-many movements. Additional information such as title, publisher, composer, etc. may be added at various positions. For details see the schema musicp.xsd.

2.1 Node Selection within Single Documents

Generally (and typically) XQuery is meant to work on top of an XML database, i.e. more than one XML instance. However, within the following we presume that the whole XML data to be queried is contained in the single file m1p.xml. XQuery provides the doc() function to obtain data contained in a file. So our very first XQuery in this primer is just

```
doc('m1p.xml')
```

- 1 -

the result of which is the whole m1p.xml document. The result of an XQuery path expression mostly being a node-set [*TXT], the result of the above XQuery, strictly speaking, is a node-set that contains as its single entry a document node.

2.1.1 Navigation

Navigation towards items to be selected is allways started at the document's root, i.e. the node beyond the uppermost element node contained in the document. From the seven distinct kinds of nodes defined in XQuery's data model draft the document root is of the kind 'document node'. The unique element node below the document root is called the 'root element' and naturally is of type 'element node'. We will discuss node kinds in more detail in section 5.2 The XQuery Typesystem. Navigation proceeds from a node to its children using a slash, i.e. '/'. Thus the following XQuery selects the document's root element, i.e. the *music* element.

```
doc('m1p.xml')/music
```

- 2 -

The next XQuery uses subsequent slashes to select all piece's titles.

```
doc('m1p.xml')/music/pieces/piece/title
```

- 3 -

Two slashes ('//') allow navigation to a node's descendants, i.e. its children, the children of these children and so on. Thus the following query also delivers the piece's titles.

```
doc('m1p.xml')//piece/title
```

- 4 -

The parts a path expression is constructed from, i.e. the bits separated by slashes or double slashes, are called steps [*LOC]. The following query selects all *title* elements in the document, regardless of where these titles are located within the document tree, i.e. the result comprises the music's title, the piece's titles and the movement's titles.

```
doc('m1p.xml')//title
```


Depending on the XQuery processor of choice the result of the above XQuery will amount to something like [*SER] :

```
<title>Drei Sonaten und drei Partiten für Violine solo</title>
<title>Sonata I</title>
<title>Adagio</title>
<title>Fuga Allegro</title>
<title>Siciliana</title>
<title>Presto</title>
<title>Partita I</title>
<title>Allemanda</title>
<title>Double</title>
<title>Corrente</title>
<title>Double Presto</title>
<title>Sarabande</title>
<title>Double</title>
<title>Tempo di Boreae</title>
<title>Double</title>
<title>Sonata II</title>
<title>Grave</title>
<title>Fuga</title>
<title>Andante</title>
<title>Allegro</title>
<title>Partita II</title>
<title>Allemanda</title>
<title>Corrente</title>
<title>Sarabanda</title>
<title>Giga</title>
<title>Ciaccona</title>
<title>Sonata III</title>
<title>Adagio</title>
<title>Fuga</title>
<title>Largo</title>
<title>Partita III</title>
<title>Preludio</title>
<title>Loure</title>
<title>Gavotte en Rondeau</title>
<title>Menuet I</title>
<title>Menuet II</title>
<title>Bourée</title>
<title>Gigue</title>
```


So far we have selected document and element nodes. To select attribute nodes from a document, precede the name of the attribute to be selected with an 'at' sign, i.e. '@'. Thus the next XQuery selects the *music* element's *ismn* attribute :

```
doc('m1p.xml')/music/@ismn
```

Path expressions enable the use of the '*' wildcard. A '*' in a path expression denotes any node of the kind that is actually searched regardless of its generic identifier [*NSP]. Thus the next query selects any attribute in the document.

```
doc('m1p.xml')//@*
```

- 8 -

Other navigation means are the parent step '..' and the single dot '.' denoting the current node. The latter will become interesting as soon as we deal with filters. The parent step navigates to the parent node, as in the following XQuery that selects all nodes that contain *title* element children.

```
doc('m1p.xml')//title/..
```

- 9 -

2.1.2 Filters and Indices

One big issue within this very brief introduction to XQuery's path expressions is filters (or predicates). This is a means of filtering a set of nodes using additional properties. The following query exhibits how to use a filter to select all movements that have the title 'Allegro'.

```
doc('m1p.xml')//movement[title = 'Allegro']
```

- 10 -

The result of this is a subset of all *movement* nodes, so the expression in square brackets filters the original set and retains only those nodes that fulfill a certain condition. Thus the next query retrieves a set of *piece* elements containing an '*Allegro*' movement, since each *movement* element is contained in a *movements* container that is again contained in a *piece* element, see musicp.xsd.

```
doc('m1p.xml')//movement[title = 'Allegro']/../..
```

- 11 -

[*UNI]
This is also where the dot '.' comes in. The next query selects all 'Allegro' movement titles.

```
doc('m1p.xml')//movement/title[. = 'Allegro']
```

- 12 -

Since we might be looking not only for mere 'Allegro' entries but more generally for titles containing the string 'Allegro' somewhere, we use the XQuery built-in function contains() as in

```
doc('m1p.xml')//movement/title[contains(.,'Allegro')]
```

- 13 -

The result of the above query is

```
<title>Fuga Allegro</title>
<title>Allegro</title>
```

- 14 -

contains() is another example for an XQuery/XPath function. A complete list of functions is provided in the W3C's Working Draft XQuery 1.0 and XPath 2.0 Functions and Operators. This primer will introduce functions where appropriate, as already done with the doc() function. A short overview on XQuery's set of functions will be provided in 4. Built-in Functions. As an example for a path expression navigating within a filter, the following XQuery does the same as a previous one, i.e. selecting all pieces that contain an Allegro. This is also an example for nested filters.

```
doc('m1p.xml')//piece[.//movement[title = 'Allegro']]
```

- 15 -

A special kind of filtering is indexing. The following XQuery retrieves the first movement of a specific piece.

```
doc('m1p.xml')//piece[title='Sonata II']/movements/movement[1]
```

- 16 -

The index '*[1]*' in the above XQuery refers to the current node's position within a specific context. What exactly this context is is determined by the axis in use. Since all path expressions so far have used the so-called abbreviated syntax, we have not yet spoken about axes. I'll try to avoid that altogether and explain indexing just by examples. In the above example, the index predicate occurs with an element following a single slash (in non-abbreviated-speak, the child axis was last traversed). In this case, the index refers to the sibling's position, i.e. for each *movements* node the first *movement* child is retrieved. Within the next query the index predicate occurs with an element following a double slash.

```
doc('m1p.xml')//piece//movement[1]
```

- 17 -

The movement nodes found are those that are the first children of their respective parent [*POS]. In this case the query finds all the first movements of pieces since there is only one *movements* child per *piece* element. There are queries where this behavior is slightly surprising. Look at the following :

```
doc('m1p.xml')//movements//title[1]
```

- 18 -

Without the index, this query would find all movement's titles. With index the query still finds all these titles since the index is interpreted as a sibling position and each title is the first (and only) child of its *movement* parent. The query does not find all titles of first movements as might have been expected. The query that does that is :

```
doc('m1p.xml')//movements/movement[1]/title
```

- 19 -

Using indexes with attributes makes no sense, since the sequence of attributes within one element is unspecified.

There is still one separate case that is illustrated in the following query :

```
(doc('m1p.xml')//piece//movement)[1]
```

- 20 -

Without the bracketing the query would retrieve those *movement* elements that are the first child of their respective *movements* parent, i.e. the first movements for every piece. With the brackets the result is the first movement for the whole document. So the effect of the bracketing is a sort of flattening. The node-set, which was previously implicitly grouped in families, loses any such structuring information and is furthermore a flat list sorted in document order. The index thus takes the first movement in document order.

In general every predicate that evaluates to a numeric value is interpreted as an index or, more precisely, as if the predicate was '*position() = ...*'. position() is a context-sensitive function that takes no input and returns the current position. Another context-sensitive function is last() that returns the number of items in the sequence currently dealt with. Since in path expressions positioning starts with 1, this is also the position of the last item of this sequence. Thus the following query returns the last movement of every piece.

```
doc('m1p.xml')//piece//movement[last()]
```

- 21 -

We have already seen nested filters. The last example in this section deals with consequent filters. The next query retrieves the first 'Allegro'-containing movement of every piece, i.e. consequent filters are applied after one another from left to right and indexes pertain to the positions after the preceding filters have been applied.

```
doc('m1p.xml')//movement[contains(title,'Allegro')][1]
```

- 22 -

As mentioned above, XQuery supports path expressions in both the abbreviated and the non-abbreviated syntax. So far we have only dealt with the abbreviated syntax. This is ok since using the non-abbreviated syntax as allowed in XQuery adds only very little value to what is already achieved with the abbreviated syntax [*NAB].

2.2 Node Selection in Document Collections

Until now we have used the doc() function to access data to be queried. This is however not a typical usage of XQuery, since it is mainly supposed to work on top of XML databases. In an XML database, XML documents (or instances) are organized into so-called *collections*. The term *collection* is not exactly defined in either the XQuery draft or the data model, but it most generally means a set of nodes. I'll use that term to denote a set of document nodes representing a set of documents in an XML database that are assembled together. For subsequent queries I presume that my XML database contains (at least) two collections. The collection *music* that contains instances conforming to the schema musicp.xsd and the collection *composer* containing instances conforming to composer.xsd [*COE]. The following query uses the collection() function to obtain the *music/title* nodes of all documents contained in the collection *music*.

```
collection('music')/music/title
```

- 23 -

The next query scans the collection for a *music* document having a specific *id* attribute and returns its *title* element.

```
collection('music')/music[@id="m1"]/title
```

- 24 -

Besides the doc() and collection() functions there is one other function that initially retrieves XML instances from a datasource, namely the input() function. This function takes no arguments and, as the draft states *returns the input sequence*. What exactly this *input sequence* is, depends on the implementation. This means to say that an XQuery implementer is free to supply as a result of input() whatever he or she thinks best. There are however two main ideas what to do here. The first idea is to let the invocation of input() evaluate to the result of a query previously executed and thus allow query pipelining. The second idea is to deliver the documents contained in the *current collection*. The *current collection* is again a term not specified in the draft or, more precisely, it is not mentioned as being part of a query's static context. If the second idea is implemented, an implementation has to specify what the *current collection* is, how it is set, etc. The last query in this section repeats the previous one presuming that the current collection is *music*.

```
input()/music[@id="m1"]/title
```

- 25 -

The three functions doc(), collection(), and input() have their own section in the draft, see Input Functions.

3. Joins and Node Generation

Syntactically, expressing joins and constructing nodes are independent XQuery facilities. These two are however most powerful when used together. Thus they are introduced simultaneously within this section.

3.1 The FLWOR Expression

The music instance m1p.xml does not specify a composer for all its pieces, but rather an element *composer_id* that identifies another XML instance representing the composer. To select a music element together with the name of its composer an XQuery needs to select the composer instance that has the same id as referred to from the music instance's *composer_id* element. The following query exhibits the usage of an XQuery FLWOR expression. FLWOR, pronounced *flower*, stands for *for*, *let*, *where*, *order by* and *return* The following FLWOR expression, consisting of a *for*, a *let* and a *return* clause, loops over all *music* instances [*LOP], finds the respective composers, and returns result nodes containing the music's title and the composer's name for each piece. A sample composer document is jsb.xml.

```
for $music in collection('music')/music
let $composer := collection('composer')/composer[@id = $music/composer_id]
return
 <result>
  <title> { $music/title } </title>
  <composer>
   { string-join(($composer/name/first,$composer/name/last)," ") }
  </composer>
 </result>
```


The result of this XQuery will be a list of *result* elements containing one entry for each music instance, i.e. among others :

```
<result>
 <title>Drei Sonaten und drei Partiten für Violine solo</title>
 <composer>Johann Sebastian Bach</composer>
</result>
```


The *for* clause binds each item from the sequence expressed by *collection('music')/music* to the variable *$music*. The subsequent *let* clause retrieves the suitable composer and binds it to the *$composer* variable. The *return* uses the two variables to construct the desired result nodes. Each result item is a *result* element containing a *title* element for the music's title and a *composer* element for the composer's name. The latter is constructed by selecting the composer's first and last name and combining these into a single string using the built-in function string-join(). This function takes two parms, a sequence of strings and a string and returns the concatenation of the members of the sequence using the second parm (here the blank) as a separator. Apart from node generation discussed in the next section the above query also introduces the sequence constructor, i.e. using brackets and commata to construct a sequence as in *($composer/name/first,$composer/name/last)*, the first input parm to *string-join()*.

3.2 Node Generation

The expression contained in the former query's *return* clause is a first example for a generated node, i.e. an element constructor. In XQuery, an element node can be constructed by simply providing the element's start- and end-tag (or an empty tag). The element's content (if any) is either literally given between start- and end-tag, or provided as an enclosed expression, or is a mixture of both. Using an enclosed expression the generated element does not contain this expression literally, but the element's content consists of what the expression enclosed in curly braces evaluates to [*AVT]. In our case, the elements use enclosed expressions containing a path expression or a function call respectively. Literal element content can be mere text, but it can also contain more literally provided element nodes or various other XML constructs as comments, CDATA sections, processing instructions, etc.

The next XQuery contains a more elaborate *return* clause exhibiting the generation of an attribute node for a generated element. The value of the attribute to-be gets computed and thus again an enclosed expression is used.

```
for $music in collection('music')/music
let $composer := collection('composer')/composer[@id = $music/composer_id]
return
 <result>
  <title ismn = " { $music/@ismn } ">
   { $music/title }
  </title>
  <composer>
   { string-join(($composer/name/first,$composer/name/last)," ") }
  </composer>
 </result>
```


The node constructing method used so far is called direct constructor. There is another method called computed constructor. Using a computed constructor the name of a node to-be can be provided dynamically. The previous query is repeated below but using a computed constructor for the attribute though the attribute's name is not yet created dynamically.

```
for $music in collection('music')/music
let $composer := collection('composer')/composer[@id = $music/composer_id]
return
 <result>
  <title>
   { attribute ismn { $music/@ismn },
     $music/title
   }
  </title>
  <composer>
   { string-join(($composer/name/first,$composer/name/last)," ") }
  </composer>
 </result>
```


A computed constructor clause contains the kind of the node to be constructed (here *attribute*), the name (here *ismn*) and the expression from which the content of the node is obtained (or rather the attribute's value in this case). There is no literal content here. If the attribute's value is a constant text, it must be enclosed in quotes, e.g. *attribute ismn { "ismn1" }*. We used a computed constructor to generate an attribute node, but computed constructors can also be used to construct element nodes, document nodes [*DOC] and text nodes. When generating elements the computed constructor is used as for attributes except the keyword *element* occurs instead of *attribute*. Since document and text nodes have no names, the respective computed constructors leave out the QName parts.

Using a computed constructor to construct elements or attributes the QName is may be provided as an enclosed expression. This is exemplified in the following query that extensively uses computed constructors to create a list of dynamically named elements for all pieces from documents in the *music* collection.

```
for $music in collection('music')/music
for $piece in $music//piece
return
 element
  { translate($piece/title,' ','_') }
  { attribute ismn { $music/@ismn },
    element from { $music/title/text() }
  }
```


translate() is a built-in function that converts the string provided as its first parameter by replacing all occurrences of the second parameter (here a blank) by the third parameter (here an underscore). This is necessary, since a string containing a blank is not allowed as an XML generic identifier. The above query takes a string as the element to-be's name. This might also be a QName to allow the name to be in a namespace. The following query does the same but introduces the xs:QName() constructor to establish the element's name.

```
for $music in collection('music')/music
for $piece in $music//piece
return
 element
  { xs:QName(translate($piece/title,' ','_')) }
  { attribute ismn { $music/@ismn },
    element from { $music/title/text() }
  }
```


xs:QName() is a first example of a constructor. In the above case it takes a string and converts it into a QName. It might also be applied upon a qualified name, i.e. a string containing a colon to create a name in the namespace denoted by the URI bound to the prefix provided by the part preceding the colon. *xs:QName* is itself a qualified name, i.e. it uses a namespace prefix. We did not define the prefix within the query, it is predefined. We'll discuss both, built-in functions and predefined function prefixes in more detail in the chapter 4.Built-in Functions. The result of the above queries will look like:

```
<Sonata_I ismn="M-006-46489-0">
 <from>Drei Sonaten und drei Partiten für Violine solo</from>
<Sonata_I>
<Partita_I ismn="M-006-46489-0">
 <from>Drei Sonaten und drei Partiten für Violine solo</from>
<Partita_I>
etc.
```


3.3 Joins

Presume we have an XML instance representing a concert schedule. Such a schedule will be made up of musical entities to be performed. The following example of a *concert* XML instance schedules a concert that contains several entities occasionally separated by breaks. The first *c_entity* is the complete Partita I and the second *c_entity* is three selected movements from Partita III.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<concert title="..." date="..." location="..." etc.>
 <c_entity nr="1">
  <music_id>m1</music_id>
  <c_item>Partita I</c_item>
 </c_entity>
 <c_entity nr="2">
  <music_id>m1</music_id>
  <c_item>Partita III
   <c_item>Preludio</c_item>
   <c_item>Menuet I</c_item>
   <c_item>Gigue</c_item>
  </c_item>
 </c_entity>
 <break/>
 <c_entity>
  ...
 </c_entity>
 ...
</concert>
```


In preparation of the concert a short program might be advantageous containing an overview of the music that will be played. This is not a very difficult task for XQuery. The task of interest here is what is usually printed on the back of the program, i.e. a short introduction to every composer being featured during the concert. The following query does this and is supposed to be an example of a serious join operation.

```
<to_be_printed>
 The Composers :
 {
  for $entity in doc('concert.xml')//c_entity
  for $music in collection('music')/music
  for $composer in collection('composer')/composer
  where $entity/music_id = $music/@id
        and $music/composer_id = $composer/@id
  return
   <print_item>
    { string-join(($composer/name/first,$composer/name/last)," ") }
    was born
    { $composer/birth/date/text() }
    in
    { $composer/birth/loc/text() }.
    { $composer/essay }
   </print_item>
 }
</to_be_printed>
```


XQuery being a functional language, it is not really to the point to think of the above FLWOR statement as three nested loops, since an XQuery implementation is not required to process a query in any specific way. Rather imagine the FLWOR as a relational table operation that an XQuery implementation is free to evaluate in whatever optimized fashion that seems appropriate.

If the above query is applied upon the *concert* example provided above, the *print_item* for J.S.Bach occurs twice since two pieces from him are performed during the concert. To prevent this we reconstruct the query and use the distinct-nodes() function as follows:

```
<to_be_printed>
 The Composers :
 {
  let $composers :=
    for $entity in doc('concert.xml')//c_entity
    for $music in collection('music')/music
    for $composer in collection('composer')/composer
    where $entity/music_id = $music/@id
          and $music/composer_id = $composer/@id
    return $composer

  for $composer in distinct-nodes($composers)
  return
   <print_item>
    { string-join(($composer/name/first,$composer/name/last),
                  " ") }
    was born
    { $composer/birth/date/text() }
    in
    { $composer/birth/loc/text() }.
    { $composer/essay }
   </print_item>
 }
</to_be_printed>
```


In the following we'll however continue discussing the simple query and ignore the fact that its results might occasionally not be unique.

The query still makes one simplifying assumption about where a *music* instance may reference a composer that we'll deal with later when having learnt about user-defined functions (5.1.2 User-Defined Functions).

There are two additional concepts with *for* and *let* clauses in FLWOR expressions that I'd like to mention before going into sorting, i.e. position variables and type declaration. The type declaration facility allows to check each value bound to a variable in a *for* clause (or *the* value bound to a variable in a *let* clause) to be of a certain type. If there is no match, an error is raised. The position variable allows to declare an additional variable in a *for* clause that is bound to the current loop number [*PVAR]. An example for a position variable can be found when explaining the position() function in 4. Built-in Functions

3.4 Sorting

The only way XQuery supports sorting is by the *order by* clause within FLWOR expressions. This is, however, not a restriction, since any expression that needs to be sorted can be wrapped into a FLWOR expression as in the following query that sorts all titles of music documents alphabetically.

```
for $title in collection('music')/music/title
order by .
return $title
```

The last but one query of the previous section plus an additional *order by* clause returns the composer data sorted alphabetically by the composer's last name. If a last name occurs more than once (as e.g. '*Bach*') the composer's first name is also considered.

```
<to_be_printed>
 The Composers :
 {
  for $entity in doc('concert.xml')//c_entity
  for $music in collection('music')/music
  for $composer in collection('composer')/composer
  where $entity/music_id = $music/@id
        and $music/composer_id = $composer/@id
  order by $composer/name/last
           $composer/name/first
  return
   <print_item>
    { string-join(($composer/name/first,$composer/name/last)," ") }
    was born
    { $composer/birth/date/text() }
    in
    { $composer/birth/loc/text() }.
    { $composer/essay }
   </print_item>
 }
</to_be_printed>
```


Before going to the next query featuring a different sorting criteria it might be interesting to consider how the result of such a FLWOR expression is ordered in absence of any *order by* clause. The rules for that are what one would expect. The order in which the *return* clause returns the results is directed by the order of the right-hand expressions in the *for* clauses. This is illustrated by the following query.

```
for $first in (1,2,3)
for $second in (4,5,6)
return ($first,$second)
```


Result:

```
(1,4,1,5,1,6,2,4,2,5,2,6,3,4,3,5,3,6)
```


Apart from the ordering aspect this query exhibits two other points worth mentioning. The sequence provided as the right-hand side of a for needs not be a sequence of nodes, it might as well be a sequence of atomic values, and secondly, sequences may not be contained in other sequences (i.e. a sequence of sequences is a sequence).

Having understood how the order of a FLWOR result depends on the order of the input to *for* the next question is about how the usual input to *for*, the path expression, is generally ordered. The answer is that the results of path expressions appear in document order, i.e. the order that the start tags appear in a document. The ordering between documents is implementation defined. So within the node sequence yielded by the expression '*$entity in doc('concert.xml')//c_entity*' entities belonging to the same documents are in document order. The last query in this section orders the composers by their birthday. The first idea is to just replace the former *order by* clause by a new one referring to '*$composer/birth/date*', but the ordering will still remain alphabetically though this is definitely not what was intended. The reason is that for the XQuery data model the content of the date field is just characters. One would expect the ordering to correspond to the type of the field to be ordered by. This can be achieved in two ways. The easier idea is to use the constructor function xs:date() to convert the characters to an instance of the date type. Another idea is to inform the query about the fact that the *composer/birth/date* field in a composer instance is supposed to be of type date by importing the respective schema composer.xsd. The introduction to typing follows in 5.2 The XQuery Typesystem. Constructor functions as xs:date() are introduced in 4.1 Constructor Functions and the "xs:" bit is explained in 4.1.1 Namespaces. Importing schemata is described in 5.1.3 Schema Import. For the time being we choose the first method and explain the missing bits later.

```
declare namespace xs = 'http://www.w3.org/2001/XMLSchema-datatypes'
<to_be_printed>
 The Composers :
 {
  for $entity in doc('concert.xml')//c_entity
  for $music in collection('music')/music
  for $composer in collection('composer')/composer
  where $entity/music_id = $music/@id
        and $music/composer_id = $composer/@id
  order by xs:date($composer/birth/date)
  return
   <print_item>
    { string-join(($composer/name/first,$composer/name/last)," ") }
    was born
    { $composer/birth/date/text() }
    in
    { $composer/birth/loc/text() }.
    { $composer/essay }
   </print_item>
 }
</to_be_printed>
```


Having declared the expression to be ordered by having *date* type is sufficient to ensure that ordering is done according to this type and not, say, alphabetically. Though this seems somehow natural we'll go deeper into this in 5.3 Comparisons and Operators. For more details on sorting details, see Order By and Return Clauses.

- collations
- empty least/greatest
- stable
- unordered

4. Built-in Functions

XQuery offers more than 100 built-in functions that are listed in the separate Working Draft XQuery 1.0 and XPath 2.0 Functions and Operators. Most of these built-in functions are contained in the namespace defined by the URI *http://www.w3.org/2002/11/xquery-functions*. This means that the functions must be called using their QNames, i.e. a prefix bound to the above URI and the local name. The draft uses the prefix *fn* for this namespace. For convenience the built-in functions belonging to that namespace might also be called without a prefix as we have done in our examples (unless a default function namespace is declared). Those functions that are declared in the namespace denoted by *op* are not directly callable but are referred to by operators as for example the function op:numeric-add() is internally called if two numbers are added using the plus operator '+'.

As an introduction to the built-in functions we'll have a closer look at the functions we've seen already.

doc()

```
fn:doc($uri as xs:string?) as document?
```

The doc() function retrieves the resource identified by the URI. The data resulting from the retrieval action is parsed as an XML document and a tree is constructed in accordance with the XQuery 1.0 and XPath 2.0 Data Model. The questionmark in a function's input signature means that an empty sequence may be provided instead of what is otherwise required. With the doc() function, as in most cases, the output of the function if applied upon the empty sequence is also the empty sequence. Thus the output type of the doc() function is *document?* meaning that the result is either a document node or an empty sequence.

contains()

```
fn:contains($operand1 as string?,
        $operand2 as string?) as boolean?
fn:contains($operand1 as string?,
        $operand2 as string?,
        $collationLiteral as anyURI) as boolean?
```

The contains() function is an example for an overloaded function, i.e. it has more than one input signature. In XQuery a function is only allowed to have multiple input signatures if these signatures have different arities. Otherwise the sometimes very generous type conversion rules might cause problems when trying to find the appropriate function definition. In our example we have allways applied contains() upon two input parms, both strings. In this case contains() finds out whether the first string operand *$operand1* contains the string provided as *$operand2*. Again the questionmarks in the declaration mean that the respective operand may be the empty sequence. If contains() is applied upon three operands, the third operand is interpreted as a collation [*COA].

position()

```
fn:position() as xs:integer?
```

position() and last() are examples for context functions. These functions get information from the Evaluation Context. The functions position() and last() access the context position and context size respectively. I guess these are only defined in path expressions and thus the only place where these can be used is in filters. The position() function can not be used directly in paths since path expressions must be of type node-set and not integer. Thus an expression like

```
doc('m1p.xml')/music/piece[title="Partita I"]/position()
```

to find the position of a specific piece is invalid. This is done by the following query using the *for* clause and a position variable (see For and Let Clauses).

```
for $title at $pos in doc('m1p.xml')/music//piece/title
where $title = 'Partita I'
return $pos
```


For string-join() and translate() refer to the section on 4.2 String Functions. For distinct-nodes() see 4.3 Functions on Sequences. For xs:date() and xs:QName() read the next section, 4.1 Constructor Functions.

4.1 Constructor Functions

We have already encountered two examples for a constructor functions, i.e. *xs:date()* and *xs:QName()*. Since *xs:QName()* is somehow special we'll explain the concept with *xs:date()*. As will be explained in 5.2 The XQuery Typesystem XQuery has a very elaborate typesystem based on a set of 46 datatypes. The basic datatypes *string*, *integer*, *decimal* and *double* can be provided literally in an XQuery. We have seen that strings are written as *'value'* or *"value"* and integers are provided as a suitable sequence of digits like *123*. Examples for instances of type *decimal* and *double* are *12.3* and *1.2e3* (meaning 120) respectively. For all other datatypes instances can not be literally provided in a query [*URI]. Constructor functions provide a means of constructing instances of these types, as for example *xs:date('1999-05-31')* constructs the date May, 31st in 1999. The prefix '*xs*' is predefined in XQuery and is by definition bound to the URI for XML Schema datatypes *http://www.w3.org/2001/XMLSchema-datatypes*. In this example the input to the constructor function is of type *string*. This string must be a valid lexical representation for the result type, as specified in XML Schema Part 2: Datatypes. It is also possible to provide a non-string input as is the case with *xs:unsignedInt(12)*. A detailed description of the mechanisms used therein is provided in the section on Casting Functions that also contains a matrix describing which type conversions are allowed. Of course it is also possible to construct/cast to user-defined types. The respective constructor functions are implicitly available with the type being introduced to the XQuery in the prolog, see Constructor Functions for User-Defined Types. The following query selects all composers that are born before a certain day.

```
for $composer := collection('composer')/composer
where $composer/birth/day < xs:date("1900-01-01")
return
 <composer>
  { string-join(($composer/name/first,$composer/name/last)," ") }
 </composer>
```


The *xs:QName()* constructor converts a string containing at most one colon into a qualified name. The part following the colon forms the local part of the name and the part preceding the colon is interpreted as a namespace prefix. The prefix must be bound to a namespace URI and this denotes the namespace the constructed QName belongs to. How prefixes are bound to namespace URIs within a query is discussed in 5.1.1 Namespaces. As a first example we'll include a query from a previous chapter and generate a new element with a QName in a specific namespace.

```
declare prefix music = "http://jhb/music"
for $music in collection('music')/music
for $piece in $music//piece
return
 element
  { xs:QName(concat('music:',translate($piece/title,' ','_'))) }
  { attribute ismn { $music/@ismn },
    element from { $music/title/text() }
  }
```

For details on *xs:QName()* see Casting to xs:QName from the Functions and Operators draft. The concat() function is explained below.

4.2 String Functions

We have already talked about contains() at the beginning of this chapter (4.Built-in Functions). We'll now introduce some more string functions.

string-join()

```
fn:string-join($operand1 as xs:string*, $operand2 as xs:string) as xs:string
```

string-join() is applied upon a sequence of strings (*$operand1*) and a string (*$operand2*) and concatenates the strings from the sequence separating them with the string provided in *$operand2*. This makes probably most sense when *$operand2* is a blank (as in our previous examples) or a comma. The result of string-join() is *xs:string* and not *xs:string?* thus it is never the empty sequence. If the value of *$operand1* is the empty sequence, the zero-length string is returned.

translate()

```
fn:translate($srcval as xs:string?,
        $mapString as xs:string?,
        $transString as xs:string?) as xs:string?
```

This function translates a string (*$srcval*) in that it replaces each occurrence of a character in *$mapString* by the character at the same position in *$transString*. For example, providing the string of all lowercase letters as *$mapString* and the string of all uppercase letters as *$transString* translate() turns *$srcval* into uppercase. Thus *translate('aBcD','abcde...','ABCDE...')* returns '*ABCD*'.

concat()

```
fn:concat() as xs:string
fn:concat($op1 as xs:string?) as xs:string
fn:concat($op1 as xs:string?, $op2 as xs:string?, ...) as xs:string
```

concat() concatenates the strings provided as parms. It does the same thing as string-join() in case the second parm to string-join() is the empty string. So concat() is strictly speaking obsolete, but it is in for reason of XPath 1.0 backwards compatibility (see concat() in the XPath 1.0 draft). What is peculiar about this function is that it is the only member of the whole set of XPath 2.0/XQuery functions that has an infinite set of potential signatures. Depending on the type checking algorithm of choice this might afford special treatment for this function.

compare()

```
fn:compare($comparand1 as xs:string?,
        $comparand2 as xs:string?) as xs:integer?
fn:compare($comparand1  as xs:string?,
        $comparand2  as xs:string?,
        $collationLiteral  as xs:string) as xs:integer?
```

compare() compares two strings optionally considering a collation.

substring()

```
fn:substring($sourceString  as xs:string?,
        $startingLoc  as xs:double) as xs:string?
fn:substring($sourceString  as xs:string?,
        $startingLoc  as xs:double,
        $length  as xs:double) as xs:string?
```

substring() is probably self explanatory, but just in case the examples below (taken from the draft) explain everything about it.

- *fn:substring("motor car", 6)* returns "*car*".
- *fn:substring("metadata", 4, 3)* returns "*ada*".

There are many more useful string functions. For a complete overview I recommend the chapter Functions on Strings from the Functions and Operators draft.

There are two other topics that are connected to string handling. The first of these is text retrieval or, as the WG puts it, *FTS* standing for *Full Text Search*. This topic is probably very important for XQuery since XML makes most sense when text is involved and thus sophisticated text handling is essential. Text retrieval is nevertheless not yet considered in the normative drafts but handled by two separate documents, i.e.

- the W3C Working Draft XQuery and XPath Full-Text Requirements, 2 May 2003, and
- the W3C Working Draft XQuery and XPath Full-Text Use Cases, 14 February 2003.

Another thing in this context is the concept of regular expressions as known from Perl. This is supported in XQuery by the three functions matches(), replace(), and tokenize(). For details on this one best reads the draft's section on String Functions that Use Pattern Matching.

4.2 Functions on Sequences

See Functions and Operators on Sequences for a nice table listing all these functions including a short explanation. In the following I'll go discuss those of the sequence function I consider interesting in the sequence they appear in the table.

boolean()

```
fn:boolean($srcval as item*) as xs:boolean
```

The boolean() function computes the xs:boolean value of the sequence $srcval. The reason I thought this function interesting is not that I guess it will be frequently used, but in that it is the result of much labour the W3C XQuery WG undertook to ensure XPath 1.0 compatibility. To simply declare an empty sequence to evaluate to false and a non-empty sequence to true would not have been in accordance with what XPath 1.0 used to do. So the boolean() function tries to mimic XPath 1.0 behavior in that a non-empty sequence evaluates to false if it contains a single null-value, i.e. a boolean false, a numeric value equal to zero, or an empty string.

concatenate()

```
op:concatenate($seq1 as item*, $seq2 as item*) as item*
```

As the namespace prefix '*op*' shows concatenate() is not a callable function but backs up an XQuery operator, in this case the comma operator. We have already encountered this one but so-far only learned that the comma may be used to build up sequences from items. The type of concatenate() shows that it is also possible to build up sequences from subsequences. Whether these expressions must then also be enclosed in round brackets, I do not know. I am sure it won't hurt.

index-of()

```
fn:index-of($seqParam  as xs:anyAtomicType*,
        $srchParam as xs:anyAtomicType) as xs:integer*

fn:index-of($seqParam  as xs:anyAtomicType*,
        $srchParam as xs:anyAtomicType,
        $collationLiteral as xs:string) as xs:integer*
```

The index-of() function takes a sequence and an item and produces the numbers of those entries in the sequence that are equal to the item. Since this seems to be a very elaborate function I'll provide an example.

```
let $title := doc('m1p.xml')/music//movement/title
return index-at($title,'Adagio')
```

- 45 -

The above example would yield the sequence consisting of the numbers 3 and 20 (see picture 6 in 2.1.1 Navigation). The same thing would be achieved by

```
for $title at $pos in doc('m1p.xml')/music//movement/title
where $title = 'Adagio'
return $pos
```

- 46 -

The second solution has the additional benefit that it might be extended from a mere comparison to applying a condition like *contains($title,'Allegro')*. So I'd rate the index-of() function as not very important.

The next two functions however are probably very important:

distinct-nodes()

```
fn:distinct-nodes($srcval as node*) as node*
```

distinct-values()

```
fn:distinct-values($srcval as xs:anyAtomicType*) as xs:anyAtomicType*
fn:distinct-values($srcval  as xs:anyAtomicType*,
            $collationLiteral as xs:string) as xs:anyAtomicType*
```

Both functions are making a sequence unique. The distinct-nodes() function does that for a sequence of nodes using as comparison mechanism node identity. Node identity is different from whether two nodes have the same string value (or typed value). So in our set of movement's titles the two 'Adagio' nodes are not identical, but two variables both denoting the same 'Adagio' title node are. The distinct-values() function uses for equality check the respective equality function that has been defined for the sequence's items, i.e. op:numeric-equal, op:time-equal(), etc.

Three sequence functions that allow modifications of sequences are insert-before(), remove(), and subsequence().

unordered()

fn:unordered($sourceSeq as item*) as item*

To conclude the description of sequence handling devices there are four operators on sequences not yet mentioned. These are the range operator and the sequence combinators union, intersect and except. The range operator uses the keyword '*to*' to construct a sequence of integers as in *1 to 4* yielding *(1,2,3,4)*. The sequence combinators are probably self explanatory. They are backed up with the sequence functions op:union(), op:intersect() and op:except() respectively.

5. Advanced Concepts

This section is supposed to cover all the rest of XQuery. In addition, the main concepts, i.e. path expressions, FLWORs and node generation, are again discussed in more detail.

5.1 The XQuery Prolog

An XQuery can be preceded by a prolog containing namespace declarations, definitions of user-defined functions, schema imports, and other issues. These three points are considered important and thus described in the subsequent sections. Four of the 'other issues' are considered not too important and are thus just listed below:

- xmlspace declaration
- default namespace declaration
- default collation
- validation declaration

There are however three additional issues that I can't very well describe since I do not yet fully understand them. Never the less I would rate them as important just to be on the safe side. The version declaration and the module import introduce the concept of modules and module libraries into XQuery. variable definitions in the prolog allow to define variables usable in the query's body and these variables might be declared as *external* meaning that they are defined by the query's external environment, whatever this might be.

5.1.1 Namespaces

We already encountered a query with a prolog in the section about sorting that is repeated below.

```
declare namespace xs = 'http://www.w3.org/2001/XMLSchema-datatypes'
<to_be_printed>
 The Composers :
 {
  for $entity in doc('concert.xml')//c_entity
  for $music in collection('music')/music
  for $composer in collection('composer')/composer
  where $entity/music_id = $music/@id
        and $music/composer_id = $composer/@id
  order by xs:date($composer/birth/date)
  return
   <print_item>
    { string-join(($composer/name/first,$composer/name/last)," ") }
    was born
    { $composer/birth/date/text() }
    in
    { $composer/birth/loc/text() }.
    { $composer/essay }
   </print_item>
 }
</to_be_printed>
```


XQuery uses QNames for (among others) elements and functions and thus there are element- and function-namespaces to be defined in an XQuery prolog [*VART]. The above query exhibits a namespace declaration that is used to call a qualified function. This kind of namespace declaration just binds a prefix to a URI. Whenever the prefix is used within a QName this object is meant to be in the namespace denoted by the URI. Thus '*xs:date*' denotes the object (in this case the function) that has the local name '*date*' and belongs to the namespace denoted by '*http://www.w3.org/2001/XMLSchema-datatypes*' which is where the W3C XML Query Working Group decided to have its constructor functions defined in. As usual in XML the prefix is insignificant. The last XQuery draft has changed w.r.t. the '*xs*' prefix. From this draft onwards this prefix is implicitly bound to the above namespace URI, i.e. the first line in the above query can be omitted in which case the prefix, of course, becomes very significant.

XQuery also enables the declaration of default namespaces. These declarations however distinguish between element- and function-namespaces, i.e. a default function namespace implicitly pertains to all non-prefixed function calls in the query whereas a default element namespace pertains to non-prefixed element names. As known from XML, default namespaces never pertain to unprefixed attribute names [*TYPE]. We encountered function namespaces in the context of 4 Built-in Functions. We'll also use them for 5.1.2 User-Defined Functions.

To illustrate element namespaces I took a query from the draft (see Namespace Declarations).

```
declare namespace foo = "http://example.org"
<foo:bar> Lentils </foo:bar>
```

- 48 -

The result is something like :

```
<foo:bar xmlns:foo="http://example.org">Lentils</foo:bar>
```

- 49 -

It is not clear to me whether the above query yields the same result, in case I declared '*http://example.org*' as the default element namespace and specified the *bar* element without a prefix. To be on the safe side either use prefixes bound in the prolog as done previously or provide an *xmlns:* attribute in the generated element in which case the query to generate the *foo:bar* element would look exactly as the result given above. So providing *xmlns:* attributes in generated elements is another way of introducing namespace bindings into a query. Bindings defined within generated elements are available within the element's scope whereas bindings in the prolog hold throughout the whole query unless they are overwritten by other declarations rebinding the same prefix.

Another necessity for element namespaces arises when accessing data that uses non-local QNames. Presume we query data containing *bar* elements belonging to a namespace denoted by '*http://example.org*'. We need to define a suitable binding and use this in path expressions.

```
declare namespace foo = "http://example.org"
collection('foobar')//foo:bar
```

- 50 -

Whether a namespace binding provided within a generated element pertains to path expressions in the element's scope is not clear to me. It is probably better to define appropriate bindings in the prolog. The whole issue becomes even more complicated when a generated element defines a default namespace. Another open question is whether namespace binding provided within a generated element can be used as function namespaces.

To be on the safe side with namespace bindings in XQuery I am currently using the following rules of thumb :

- do not use a default element namespaces in the prolog,
- do not declare default namespaces in generated elements,
- for path expressions use prefixes defined in the prolog,
- if a generated element is in a namespace, declare an appropriate binding in the element,
- do not use a default function namespace, and
- do not use namespaces in generated elements as function namespaces.

About function namespaces in more is said in both 4. Built-in Functions and 5.1.2 User-Defined Functions.

5.1.2 User-Defined Functions

The query prolog is also the place to declare user-defined functions. The following example exhibits a function '*coname()*' and its usage within a well-known query.

```
define function coname($co)
 { string-join(($co/name/first,$co/name/last)," ") }
<to_be_printed>
 The Composers :
 {
  for $entity in doc('concert.xml')//c_entity
  for $music in collection('music')/music
  for $composer in collection('composer')/composer
  where $entity/music_id = $music/@id
        and $music/composer_id = $composer/@id
  return
   <print_item>
    { coname($composer) }
    was born
    { $composer/birth/date/text() }
    in
    { $composer/birth/loc/text() }.
    { $composer/essay }
   </print_item>
 }
</to_be_printed>
```


A function is defined using the keywords *define function* followed by a QName (we use a local name here), followed by a parameter list enclosed in round brackets, followed by an expression enclosed in curly brackets. When the function gets applied upon an expression the argument values provided in the function call expression are bound to the formal parameters of the function, and the function body is evaluated, see Function Calls.

As previously mentioned, the query above still makes one simplifying assumption, namely that every item of a musical library contains material from only a single composer (as our sample instance m1p.xml does). This is not necessarily the case. The schema music.xsd reflects that by allowing *composer_id* elements for both the whole musical item and the *piece* elements. In case of multiple composers per document, an author of a *music* document might decide to give a general composer and add *composer_id* to those pieces having a different composer, or to give no general composer and provide a *composer_id* with each piece. The enhanced query is as follows :

```
define function getCid($it,$mu)
 {
  let $cid := $mu/piece[title = $it/text()]/composer_id
  return
   if $cid then $cid
   else $mu/composer_id
 }
<to_be_printed>
 The Composers :
 {
  for $item in doc('concert.xml')/entity/item
  for $music in collection('music')/music
  for $composer in collection('composer')/composer
  where $item/../music_id = $music/@id
        and getCid($item,$music) = $composer/@id
  return
   ...
 }
</to_be_printed>
```

There is much to be said about this query. The first very obvious thing is that it silently introduces '*if then else*'. There is not much to be said about this, in doubt read, Conditional Expressions. Another aspect here is that XQuery machines sitting on top of databases and apt to deal with large quantities of data will probably implement some optimization algorithms to speed up joins. When looking at the previous query (the one without the getCid() function) several ideas of how to speed up such a join arise. Suitable optimization technique might be to use information about the number of documents in the *music* and *composer* collections or whether indices are defined for fields mentioned in the *where* clause. Now the getCid() function is added some optimizations might have more work to do to understand how to do things quicker or might just fail. Thus the last query can be considered as an example of how join optimization (if any) can be foiled by user defined function (a thing that can also be said of *if the else* clauses).

The effect of possibly irritating optimization techniques gets even worse if user-defined functions happen to be recursive. Nevertheless we provide one recursive user-defined function for illustration. The following query detects the *depth* of the music document in m1p.xml, i.e. how far the furthest child node is away from the document node.

```
define function depth($node)
{
  if (node-kind($node) = 'document')
  then 0
  else 1 + depth($node/..)
}
max(for $x in doc('m1p.xml')//* return depth($x))
```


The built-in function node-kind() is applied upon a node and yields the kind of the node as a string value, i.e. '*element*', '*attribute*', etc. max() is probably self-explaining.

Some XQuery implementations (for example Michael Kay's saxon) insist that user-defined functions are in a namespace. Thus the above query must be rewritten as follows:

```
declare namespace uf = 'http://jhb/userDefinedFunctions'
define function uf:depth($node)
{
  if (node-kind($node) = 'document')
  then 0
  else 1 + uf:depth($node/..)
}
max(for $x in doc('m1p.xml')//* return uf:depth($x))
```


We added a query prolog containing a namespace declaration and used the prefix in both the function definition and function calls.

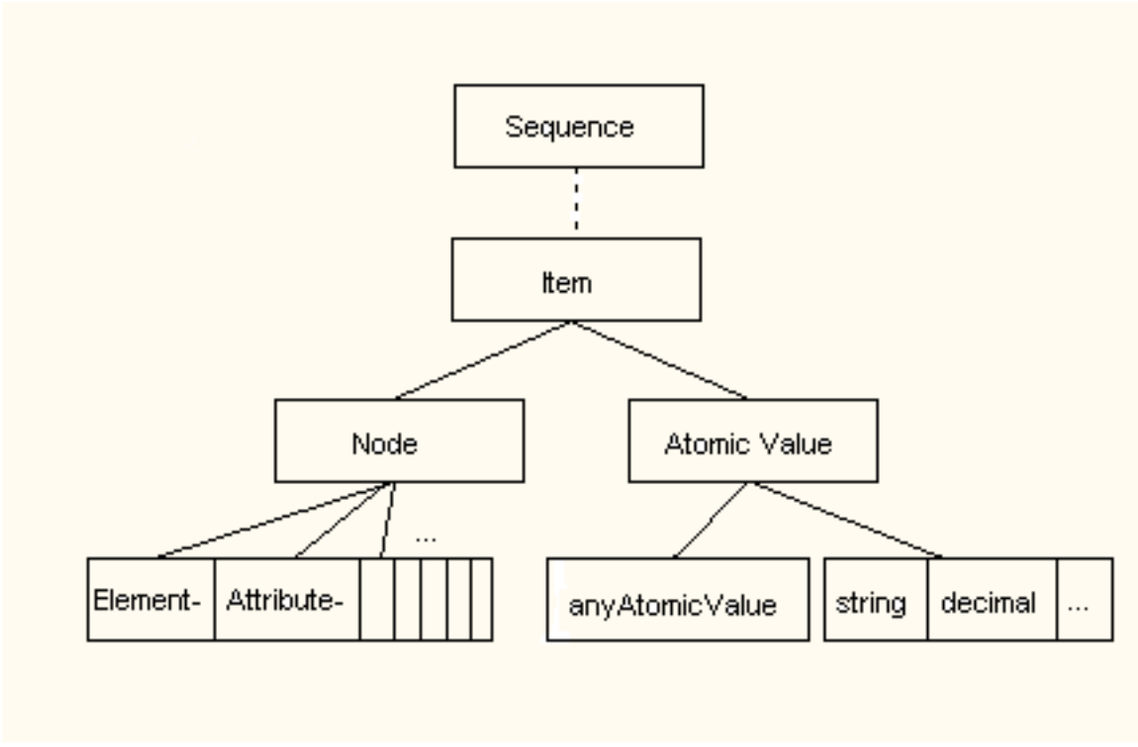
5.1.3 Schema Import

We will introduce type import by returning to a previous query that used the *xs:date()* function to sort by a date field. Instead of explicitly mentioning the datatype to be ordered by in the query we now want to sort by the datatype that the schema declares for the respective field. Thus we import musicp.xsd which will make the types the fields in each *music* instance adhere to known to the XQuery processor.

```
import schema namespace music="http://jhb/music"
<to_be_printed>
 The Composers :
 {
  for $entity in doc('concert.xml')//c_entity
  for $music in collection('music')/music:music
  for $composer in collection('composer')/composer
  where $entity/music_id = $music/@music:id
        and $music/music:composer_id = $composer/@id
  order by $composer/birth/date
  return
   <print_item>
    { string-join(($composer/name/first,$composer/name/last)," ") }
    was born
    { $composer/birth/date/text() }
    in
    { $composer/birth/loc/text() }.
    { $composer/essay }
   </print_item>
 }
</to_be_printed>
```


5.2 The XQuery Typesystem

Completely understanding XQuery typing is a slightly complicated issue since XQuery uses two type systems. These are, first, the one described in the data model draft and, second, the W3C XML Schema type system as described in the W3C Recommendation XML Schema Part 1: Structures. We will refer to these two systems as the XQuery type system and the WXS type system. The systems do intersect in that they are both based upon the same set of datatypes described in the W3C Recommendation XML Schema Part 2: Datatypes. We'll first deal with the XQuery type system that is shown in the following diagram.

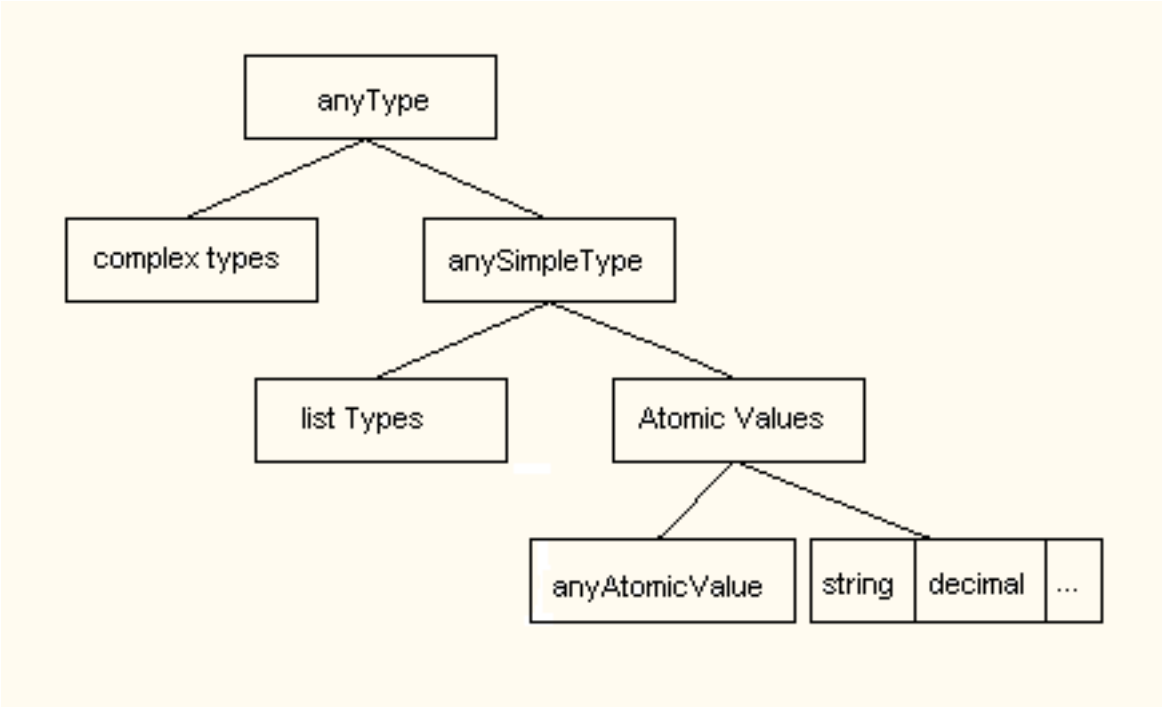


All objects in XQuery are *sequences*. This implies that to the XQuery type system a single value is the same as a sequence containing exactly one item (a so-called singleton sequence). We have already encountered the empty sequence denoted by '*()*'. There is no such thing as a sequence of sequences. The entries of sequences are called items and an item is either a node or an *atomic value*.

Nodes may be of seven different kinds. These are element, attribute, document, text, namespace, processing instruction, and comment.

The type of an atomic value can be a built-in type or a user-defined type defined by schema import The XQuery built-in atomic types are 41 of the 44 datatypes as defined in the W3C Recommendation XML Schema Part 2: Datatypes plus two additional types defined especially for XQuery, i.e. 43 built-in types. Three datatypes defined in XML Schema are singled out, since they are *list* datatypes and not *simple* [*ATO]. These are *NMTOKENS*, *IDREFS*, and *ENTITIES*. An overview of all 44 XML Schema datatypes is found in the chapter Built-in datatypes from *XML Schema Part 2: Datatypes*. The two additional XQuery datatypes are *yearMonthDuration* and *dayTimeDuration*. Since the necessity for these types arose when defining some XQuery built-in functions, these types are defined in the Functions and Operators document, see Two Totally Ordered Subtypes of Duration. Of the 43 built-in datatypes we have so far some are *primitive* (as.e.g. *decimal*) and some are *derived*, meaning they are defined as derivations of other types, as *integer* being a *decimal* without fraction digits. A user-defined atomic type must be derived by restriction from a built-in type. On top of all these atomic types XQuery uses the abstract type denotation *anyAtomicType* to denote the type of an atomic value for which no specific type is known. The system is designed to give a type to every object occurring within an XQuery [*CHC].

The content types of element and attribute nodes is where the other type system, the WXS types, comes in. This type system has especially been designed to denote those content models. The WXS type system is provided below.



anyType is the most general type for an element's content and *anySimpleType* is the most general type for an attribute's content. Thus an attribute may have list content as have those defined as being for example of type *IDREFS*. Whenever an element (or attribute) occurs in an XQuery and gets typed its type is *element node (content type)* and the *content type* is expressed by means of the WXS type system. But how are objects described by the WXS type system converted into the XQuery type system ? We have already seen the following query.

```
for $music in collection('music')/music
let $composer := collection('composer')/composer[@id = $music/composer_id]
return
 <result>
  <title>
   { attribute ismn {$music/@ismn},
     $music/title
   }
  </title>
  <composer>
   { string-join(($composer/name/first,$composer/name/last)," ") }
  </composer>
 </result>
```

Where does conversion from one type system into the other take place in this query ? It happens for the first time in the expression *attribute ismn {$music/@ismn}* wherein an attribute node gets converted into something that becomes the attribute's value. Another occasion is when providing parms for the string-join() function. In both cases the process applied for type conversion is the so-called atomization. Atomization is applied, for example, if a node is supplied as a function parameter, where a simple type is expected. In contradiction to what the term might imply the result of atomization is not an allways atomic value, but a sequence of atomic values. Atomization applied upon a node yields the so-called typed value of the node that is its string value or a value of a specific type if the type of the node's content can be obtained. For example atomizing *$composer/name/first* yields the composer's first name as a text string.

The XQuery typesystem usually just forms a basis upon which the processing of an XQuery is performed. It is however also possible to mention types directly in a query make the types of the objects more transparent. If the XQuery processor in use applies static typing (one of XQuery's optional features), the query might be rejected due to type mismatch before it gets executed. This saves time and allows a more precise error reporting. The next query is a copy from a query we encountered before adorned with additional type declarations.

```
declare namespace uf = 'http://jhb/userDefinedFunctions'
define function uf:depth($node as node) as xs:integer
{
  if (node-kind($node) = 'document')
  then 0
  else 1 + uf:depth($node/..)
}
max(for $x in doc('m1p.xml')//* return uf:depth($x))
```


- use-defined types

5.3 Comparisons and Operators

The last query contained an example for an XQuery operator, namely *'<'*. More precisely, this is a comparison operator. The exact meaning of the boolean expression *$composer/birth/day < xs:date("1900-01-01")* is: there is at least one node in the node-set the path expression on the left evaluates to, for which the property being less than the expression on the right holds. This comparison technique is called general comparison. The idea of interpreting a comparison operator involving path expressions is often referred to as *implicit any semantics* and was by some considered irritating. To clearer distinguish between comparison and set calculus, XQuery introduces another kind of comparison technique called value comparison. The value comparison operator that corresponds to the general comparison operator *'<'* is *'lt'*. The above query using value comparison instead of general comparison thus looks as follows.

```
declare namespace xs
   = "http://www.w3.org/2001/XMLSchema-datatypes"
for $composer := collection('composer')/composer
where xs:date($composer/birth/day) lt xs:date("1900-01-01")
return
 <composer>
  { string-join(($composer/name/first,$composer/name/last)," ") }
 </composer>
```


Different from general comparison this query expects the path expression *$composer/birth/day* to yield exactly one node. If there is more than one such node a type error is thrown. If there is no such node, the result of the comparison is the empty sequence. In the latter case the *where* clause's effective boolean value is *false* and the respective composer is skipped.

A topic that we have not dealt with yet is how a comparison operator is to be applied, i.e. how does the system know that some comparisons are to be done alphabetically and in others as in the one above *lt* is to be interpreted as *earlier*. The key to this question is the so-called operator mapping mechanism. If an operator is encountered, a table containing all operators is searched. If the table contains an entry for the respective operator and the two operand's types, the entry shows the function that is to be applied upon the operands to obtain the result of the operator's evaluation. If there is no such entry, there is no such function, i.e. an error is thrown. For our example above the table exhibits the following entry

A lt B	xs:date	xs:date	op:date-less-than(A, B)	xs:boolean

This tells us that it is allowed to use *'lt'* upon a pair of dates and the function op:date-less-than() gets applied of which the result is a boolean. The prefix '*op*' and its corresponding namespace URI '*http://www.w3.org/2002/11/xquery-operators for operators*' are reserved to denote functions that are applied when an operator is encountered.

Besides comparison operators, the operator table also contains entries for arithmetic operators. Arithmetic operators work the same way as comparison operators except for two additional concepts. The first thing is that when, for example, adding an integer to something that is not a number, the system tries to convert the latter to a number. (Even a node will be accepted due to called atomization.) The second thing is called type promotion and simply means that if two numbers are added that have a different numeric type (lets say *integer* and *double*) the operand having the less specific type is cast to the higher specific type. The operator table denotes this with the entry *numeric* as for example in :

A + B	numeric	numeric	op:numeric-add(A, B)	numeric

Read more about this in Arithmetic Expressions.

There are two other means of comparison. These are node comparison ("is" | "isnot") that introduces node identity and order comparison ("<<" | ">>") that works like the *precedes* and *following* axes (only defined in XPath 2.0).

5.4 Miscellaneous

- grouping / distinct-values()
- quantification
- whitespace
- modules & libraries
- optional features (static typing)

6. Footnotes

[*MOS] From my point of view it is sufficient to look at these three but the whole description of XQuery consists of five normative drafts plus six non-normative ones. See appendix 7.2 The XQuery Documents.

[*TXT] For a path expression returning a node-set consisting of text nodes or even of a text node singleton, the fact that this is a node-set is not allways obvious. Never the less the result of something like *doc('m1p.xml')//instrument[1]/text()* is still a text node containing the string-value *violin* and not a simple value (in XQuery referred to as atomic value).

[*LOC] What is referred to as steps in XQuery 1.0 is not the same thing that was called *location step* in XPath 1.0. A *step* might be either a location step (though this term is deprecated) or something called a primary expression. This among others allows function calls, variables, etc. to occur in path expressions.

[*SER] How the result returned by an XQuery implementation really looks like depends on two things. First the result being a sequence of items must be turned into an XML document and second, it must be serialized. There is a specific document on the latter, i.e. XSLT 2.0 and XQuery 1.0 Serialization, which I have not yet read. In this primer XQuery results are allways just straight forward ideas of how the data model representation of the result might be written down.

[*NSP] Actually, '*' selects all nodes of the current kind regardless of their local name and namespace. Wildcards specifying arbitrary local names in a specific namespace or specific local names in arbitrary namespaces are also possible, i.e. '*prefix:**' or '**:localname*' respectively.

[*UNI] The result sequence of that query contains each *piece* element only once even those that have more than one 'Allegro' movement, since the results of path expressions are automatically sorted in document order and made unique. .

[*POS] This is due to the fact that // is an abbreviation for */descendant-or-self::node()/* and thus the last axis traversed before the index is again the child axis.

[*NAB] The full-blown non-abbreviated syntax allows axes that are not contained in the abbreviated syntax as the *antecedents*, *preceding*, and *preceding-sibling* axes. However, the axes allowed in XQuery's non-abbreviated syntax are those that exist in an abbreviated version anyway. There is only one point in which XQuery's non-abbreviated syntax adds something. The abbreviated syntax contains no replacement for *descendant-or-self::node()*, since the double slash, being an abbreviation for */descendant-or-self::node()/*, never retrieves the current node. Use the extended expression *. union .//node()* instead.

[*COE] Within this primer we presume a collection to contain only document nodes. This does not necessarily mean that all documents in a collection conform to the same schema, as is the case with our music and composer collections.

[*LOP] To say that a *for* in a FLWOR statement initiates a loop is very informal, since it does not pay tribute to the fact that XQuery is a functional language, i.e. there is no such thing as looping. The proper wording as taken from the XQuery 1.0 draft is that the *for* generates a tuple-stream and the result of the FLWOR expression is an ordered sequence containing the concatenated results of the tuple's evaluations.

[*AVT] XQuery's *enclosed expression* is similar to XSLT's *attribute value templates*.

[*DOC] The computed document constructor will become especially interesting as soon as update facilities are introduced into XQuery. Creating an entire new document and making it persistent will probably require more parms to be added to this constructor.

[*PVAR] The concept of position variables is still very vague to me since, as I explained above a *for* is not exactly a loop. How does the usage of position variables affect optimization ? Is this the same as what the position() function would yield, if this would be allowed where the position variable is used ? How does the position variable work in the presence of an *order by* clause ?

[*COA] A collation is a pair consisting of a relation telling which characters are to be regarded as equal and a second relation that orders the available characters. The first relation is necessary to compare strings for equality, the second is necessary for ordering operators and sorting.

[*URI] An exception to this rule occurs in the XQuery prolog where a namespace URI can be provided as a string constant, i.e. simply surrounded by quotes or double quotes.

[*VART] Variables and types do also have QNames.

[*TYPE] Default element namespaces pertain to types.

[*ATO] The data model draft states: *An atomic type is a primitive simple type or a type derived by restriction from a primitive simple type. Types derived by list or union are not atomic* (from Atomic Values).

[*CHC] There is one additional type class occurring during XQuery typing, i.e. a choice type. The sequence *(1,'x')* for example has a type like *sequence(choice(integer,string))*.

7. Appendix

7.1 Sample Data

7.1.1 A *music* Instance m1p.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<music xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:noNamespaceSchemaLocation="musicp.xsd"
       id="m1"
       ismn="M-006-46489-0"
       type="concert">
<title>Drei Sonaten und drei Partiten für Violine solo</title>
<publisher>Bärenreiter</publisher>
<composer_id>c1</composer_id>
<opus>BWV 1001-1006</opus>
<remarks>
 <remark>Urtext der neuen Bach-Ausgabe</remark>
 <remark>Herausgegeben von Günter Haußwald</remark>
</remarks>
<instruments>
 <instrument>Violin</instrument>
</instruments>
<pieces>
 <piece>
  <title>Sonata I</title>
  <opus>BWV 1001</opus>
  <movements>
   <movement><title>Adagio</title></movement>
   <movement><title>Fuga Allegro</title></movement>
   <movement><title>Siciliano</title></movement>
   <movement><title>Presto</title></movement>
  </movements>
 </piece>
 <piece>
  <title>Partita I</title>
  <opus>BWV 1002</opus>
  <movements>
   <movement><title>Allemanda</title></movement>
   <movement><title>Double</title></movement>
   <movement><title>Corrente</title></movement>
   <movement><title>Double Presto</title></movement>
   <movement><title>Sarabande</title></movement>
   <movement><title>Double</title></movement>
   <movement><title>Tempo di Boreae</title></movement>
   <movement><title>Double</title></movement>
  </movements>
 </piece>
 <piece>
  <title>Sonata II</title>
  <opus>BWV 1003</opus>
  <movements>
   <movement><title>Grave</title></movement>
   <movement><title>Fuga</title></movement>
   <movement><title>Andante</title></movement>
   <movement><title>Allegro</title></movement>
  </movements>
 </piece>
 <piece>
  <title>Partita II</title>
  <opus>BWV 1004</opus>
  <movements>
   <movement><title>Allemanda</title></movement>
   <movement><title>Corrente</title></movement>
   <movement><title>Sarabanda</title></movement>
   <movement><title>Giga</title></movement>
   <movement><title>Ciaccona</title></movement>
  </movements>
 </piece>
 <piece>
  <title>Sonata III</title>
  <opus>BWV 1005</opus>
  <movements>
   <movement><title>Adagio</title></movement>
   <movement><title>Fuga</title></movement>
   <movement><title>Largo</title></movement>
  </movements>
 </piece>
 <piece>
  <title>Partita III</title>
  <opus>BWV 1006</opus>
  <movements>
   <movement><title>Preludio</title></movement>
   <movement><title>Loure</title></movement>
   <movement><title>Gavotte en Rondeau</title></movement>
   <movement><title>Menuet I</title></movement>
   <movement><title>Menuet II</title></movement>
   <movement><title>Bourée</title></movement>
   <movement><title>Gigue</title></movement>
  </movements>
 </piece>
</pieces>
</music>
```

7.1.2 The *music* Schema musicp.xsd

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

 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

<xs:element name='music'>
 <xs:complexType>
  <xs:sequence>
   <xs:element ref='title'/>
   <xs:element ref='publisher'/>
   <xs:element ref='composer_id' minOccurs='0' maxOccurs="1"/>
   <xs:element ref='opus' minOccurs='0' maxOccurs="1"/>
   <xs:element ref='remarks' minOccurs="0" maxOccurs="1"/>
   <xs:element ref='instruments' minOccurs="0" maxOccurs="1"/>
   <xs:element ref='pieces'/>
  </xs:sequence>
  <xs:attribute name="ismn" type="xs:string"/>
  <xs:attributeGroup ref="atts1"/>
 </xs:complexType>
</xs:element>

<xs:simpleType name='Mtype'>
 <xs:restriction base='xs:string'>
  <xs:enumeration value="study"/>
  <xs:enumeration value="concert"/>
 </xs:restriction>
</xs:simpleType>

<xs:attributeGroup name='atts1'>
 <xs:attribute name='type' type='Mtype'/>
 <xs:attribute name='manageable' type='xs:boolean'/>
</xs:attributeGroup>

 <xs:element name="title" type='xs:string'/>
 <xs:element name="publisher" type='xs:string'/>
 <xs:element name="composer_id" type='xs:integer'/>
 <xs:element name="opus" type='xs:string'/>

 <xs:element name='remarks'>
  <xs:complexType>
   <xs:sequence>
    <xs:element ref='remark' minOccurs='1' maxOccurs='unbounded'/>
   </xs:sequence>
  </xs:complexType>
 </xs:element>

 <xs:element name='remark' type='xs:string'/>

 <xs:element name='pieces'>
  <xs:complexType>
   <xs:sequence>
    <xs:element ref='piece' minOccurs='1' maxOccurs='unbounded'/>
   </xs:sequence>
  </xs:complexType>
 </xs:element>

 <xs:element name='piece'>
  <xs:complexType>
   <xs:sequence>
    <xs:element ref='title'/>
    <xs:element ref='composer_id' minOccurs='0' maxOccurs='1'/>
    <xs:element ref='opus' minOccurs='0' maxOccurs='1'/>
    <xs:element ref='instruments' minOccurs='0' maxOccurs='1'/>
    <xs:element ref='remarks' minOccurs='0' maxOccurs='1'/>
    <xs:element ref='movements' minOccurs='0' maxOccurs='1'/>
   </xs:sequence>
   <xs:attributeGroup ref="atts1"/>
  </xs:complexType>
 </xs:element>

 <xs:element name='instruments'>
  <xs:complexType>
   <xs:sequence>
    <xs:element ref='instrument'
                minOccurs='1' maxOccurs='unbounded'/>
   </xs:sequence>
  </xs:complexType>
 </xs:element>

 <xs:element name='instrument' type='xs:string'/>

 <xs:element name='movements'>
  <xs:complexType>
   <xs:sequence>
    <xs:element ref='movement'
                minOccurs='1' maxOccurs='unbounded'/>
   </xs:sequence>
  </xs:complexType>
 </xs:element>

 <xs:element name='movement'>
  <xs:complexType>
   <xs:sequence>
    <xs:element ref='title'/>
   </xs:sequence>
   <xs:attributeGroup ref="atts1"/>
  </xs:complexType>
 </xs:element>

</xs:schema>
```

7.1.3 The *composer* Instance jsb.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<composer
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:noNamespaceSchemaLocation="composer.xsd"
 id="c1">
 <name>
<first>Johann Sebastian</first>
<last>Bach</last>
</name>
<birth>
<date>1685-03-23</date>
<loc>Eisenach</loc>
</birth>
<death>
<date>1750-07-28</date>
<loc>Leipzig</loc>
</death>
<essay from="link3">
One of the greatest composers in history, Johann Sebastian Bach
was by far the most significant member of the Bach dynasty of
musicians ...
</essay>
<links>
<link nr="link1" href="http://www.jsbach.org/">
 J.S.Bach Homepage
</link>
<link nr="link2" href="http://www.jsbach.net/">
 Dave's J.S.Bach Page
</link>
<link
 nr="link3"
 href="http://www.classical.com/reference/composerbio.php?id=39">
Bach at Classical.Net
</link>
</links>
</composer>
```

7.1.4 The *composer* Schema composer.xml

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

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

 <xs:element name='composer'>
  <xs:complexType>
   <xs:sequence>
    <xs:element ref='name'/>
    <xs:element name='birth' type='dateLocType'/>
    <xs:element name='death' type='dateLocType'/>
    <xs:element ref='essay'/>
    <xs:element ref='links'/>
   </xs:sequence>
   <xs:attribute name="id" type="xs:string"/>
  </xs:complexType>
 </xs:element>

 <xs:element name='name'>
  <xs:complexType>
   <xs:sequence>
    <xs:element name='first' type="xs:string"/>
    <xs:element name='last' type="xs:string"/>
   </xs:sequence>
  </xs:complexType>
 </xs:element>

 <xs:element name='essay'>
  <xs:complexType mixed="true">
   <xs:attribute name="from" type="xs:string"/>
  </xs:complexType>
 </xs:element>

 <xs:element name='links'>
  <xs:complexType>
   <xs:sequence>
    <xs:element name='link' minOccurs='1' maxOccurs='unbounded'/>
   </xs:sequence>
  </xs:complexType>
 </xs:element>

 <xs:element name="link">
  <xs:complexType>
   <xs:attribute name="nr" type="xs:string"/>
   <xs:attribute name="href" type="xs:string"/>
  </xs:complexType>
 </xs:element>

 <xs:complexType name='dateLocType'>
  <xs:sequence>
   <xs:element name='date' type='xs:date'/>
   <xs:element name='loc' type='xs:string'/>
  </xs:sequence>
 </xs:complexType>

</xs:schema>
```

7.2 The XQuery Documents

Two non-normative documents that mark the beginning of the XQuery development :

- The W3C Working Draft XML Query Requirements, 2 May 2003,
- The W3C Working Draft XML Query Use Cases, 2 May 2003.

The complete normative description of XQuery:

- The W3C Working Draft XQuery 1.0: An XML Query Language, 2 May 2002,
- The W3C Working Draft XQuery 1.0 Formal Semantics, 2 May 2003,
- The W3C Working Draft (Last Call) XQuery 1.0 and XPath 2.0 Data Model, 2 May 2003,
- The W3C Working Draft (Last Call) XQuery 1.0 and XPath 2.0 Functions and Operators, 2 May 2003,
- The W3C Working Draft XSLT 2.0 and XQuery 1.0 Serialization, 2 May 2003.

Two non-normative specification:

- The W3C Working Draft XML Path Language (XPath) 2.0, 2 May 2003,
- The W3C Working Draft XML Syntax for XQuery 1.0 (XQueryX), 07 June 2001.

On behalf of XQuery's FTS activities see the two documents :

- The W3C Working Draft XQuery and XPath Full-Text Requirements, 2 May 2003,
- The W3C Working Draft XQuery and XPath Full-Text Use Cases, 14 February 2003.