SU/DSV
KTH/EECS

QUERYING XML DATA WITH XQUERY

v. 3.0

SDXML (IB166N)
Models and Languages for Handling
Semi-structured Data and XML

January 2024

Rafael Cordones Marcos
nikos dimitrakas

Table of contents

1 Introduction

This compendium is a brief introduction to XQuery, a query language for querying XML data and two execution environments for XQuery: XQuisitor [1] and BaseX [2]. Functionality for XQuery 1 and XQuery 3 is covered through multiple examples. The reader should still be familiar with XML, XPath and XQuery before starting with the examples in the following chapters. For XQuery, the article *XQuery: An XML query language* [3] can be a suitable starting point, or perhaps any book introducing basic concepts of XML, XPath and XQuery, for example *Querying XML* [4]. It will be easier to understand what comes next if you understand concepts like XML nodes (elements, attributes, comments, etc), data definition documents (DTDs) and XML Schemas, well-formed XML documents, valid XML documents, etc. Also, any prior knowledge of basic programming concepts can be useful.

The main contents of this document are:

- A very brief introduction to XQuery
- An introduction to the tools XQuisitor and BaseX
- Examples and exercises on using XQuery for querying XML data

This introduction covers both XQuery 1 and XQuery 3. Chapter 5 is full of examples compatible with XQuery 1, while chapter 6 has examples requiring XQuery 3.

2 Execution environments

XQuery can be executed with many different tools, some with a user-friendly interface. It is also possible to execute XQuery with custom programs or from within SQL. The two execution environments presented here are XQuisitor and BaseX. There are differences between them both when it comes to what they support, but also in how they handle certain things like serialization of sequences, literals and nodes of different types.

2.1 XQuisitor

XQuisitor is a simple GUI written in Java that allows the user to evaluate XQuery 1 statements. It's free software so you can download a binary distribution as well as the source code of the application.

XQuisitor's GUI looks like this:

Here is an explanation of the different parts of the GUI:

Menu (File, Edit, Query): provides the standard capabilities for loading, saving and printing queries, XML data files and query results.

Base URI: here you can select the Universal Resource Identifier (URI) that will be added as a prefix to relative URIs you use in your queries. The URI you choose here will almost always be a directory in your local disk, but you could also use a URL. We can use an example to make this functionality clearer. If you write in a query the function `doc("books.xml")` (we will explain later what this function does), XQuisitor will prepend to `"books.xml"` the URI indicated in Base URI before trying to open the `books.xml` file. So, if the Base URI textbox contains `"file:/D:/labs/xquery/"` and you use the function `doc("books.xml")` then XQuery will actually use `doc("file:/D:/labs/xquery/books.xml")`.

Context: in this textbox you can indicate which XML file you want to use for your queries. You can think of the context of a query as the XML input data.

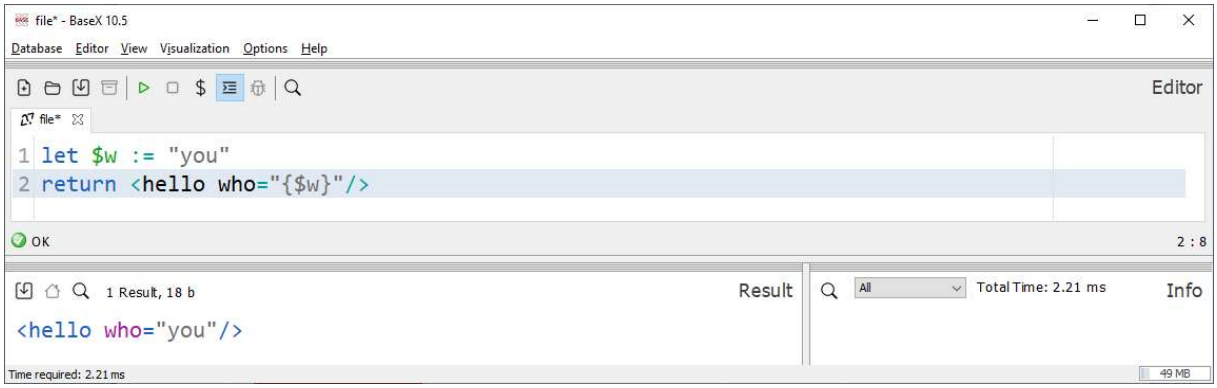
Query area: you write your queries in this textbox.

Query result: you get the results of the query execution in this textbox. By selecting **Pretty Print** you will get the result indented. If you select **Wrap**, the result will be wrapped in an XML document. This can be useful especially when the result of the query is something that XQuisitor cannot serialize, like a literal, a sequence of literals or an attribute node.

Running XQuisitor requires Java. Depending on which Java version your system has and how it is configured, XQuisitor may be started by simply double-clicking on XQuisitor.jar or by writing the command `javaw -jar XQuisitor.jar` when inside the appropriate folder.

2.2 BaseX

BaseX is a free open-source XQuery execution environment that supports XQuery. It also supports the XQuery Update Facility and XSLT. BaseX has a simple user interface that is similar to XQuisitor. Unlike XQuisitor, BaseX support the latest versions of XQuery and the XQuery Update Facility. There is a possibility to configure a default context (called a database, but it is an XML file or a folder with multiple XML files). There is one text area for the query and one text area for the result. Errors are shown below the query area and it is possible to show an extra pane with "Query info" where errors or execution details will appear. Multiple tabs can be opened, each one with each own query, but all sharing the same result pane. Here is BaseX with a simple query executed:



The selected database (default context), if any, is shown in the window title. Use the Database menu to open (or create) a database. It is also possible to open a database (that has already been created) with the BaseX XQuery function `db:open`. It is also possible to work without using a database by utilizing the function `doc()` to open an XML file.

3 Sample data

Four files with sample data are provided and will be used during the examples in later chapters. They are `books.xml`, `books.dtd`, `publishers.xml` and `publishers.dtd`.

Have a look at these files so you can see what kind of data they contain before proceeding with the example queries. For most of the examples we will only use the file `books.xml`.

4 XQuery

This chapter has a short introduction to some basic concepts of XML [5], XPath [6], XQuery [7], the accompanying functions [8] and the XQuery Update Facility [9].

XQuery is a language focused on information retrieval from XML data. The result of evaluating an XQuery statement is not always a well-formed XML document. In XQuery you can have a query like

```
let $a := 3, $b := 5
return $a + $b
```

Which, when evaluated, would return the value `8`.

XQuery keywords are case-sensitive which means that "`where`" is a correct XQuery keyword while "`WHERE`" is not.

4.1 Data Model

XQuery is also a model for representing literals, nodes and sequences. In later versions also functions, maps and arrays.

XML data is made up of nodes and each node can be of several kinds, of which, element, attribute and text are the ones that concern us most. Consider the following XML data:

```
<?xml version="1.0" encoding="UTF-8"?>
<Book Title="Database Systems in Practice" OriginalLanguage="English"
      Genre="Educational">
     <Author Name="Alan Griff" Email="ag@mit.edu" YearOfBirth="1972"
             Country="USA"/>
</Book>
```

It consists of two element nodes:

- `Book` with attribute nodes `Title`, `OriginalLanguage` and `Genre`
- `Author` with attribute nodes `Name`, `Email`, `YearOfBirth` and `Country`.

Each of the attribute nodes has a text node containing the contents of the attribute. For example, the attribute node `Genre` has a text node with the text `Educational` as its content.

4.2 Serialization and Deserialization of XML Documents

Serialization and deserialization is a very important concept in computer science and you have probably met it before with a different name. Serialization is the process by which a data structure residing in memory is stored in a persistent medium. Usually, in the domain of programming languages (like Java) this data structure is an object or a graph of objects. Serializing the object, means storing it on disk (usually in a file). In the XML domain, we serialize XML which resides in memory to an XML file. Deserialization is then the inverse process, taking an XML file and building a representation of it in memory, probably according to the XQuery model.

4.3 XPath Expressions

XQuery builds on XPath, which is a language for selecting parts of XML documents. You will almost always use an XPath expression in your queries. Their main purpose is to select nodes from the input XML data.

An XPath expression may start with the slash character "`/`" which indicates the root of the input XML document. Every slash in the expression indicates a next step and **the result of each step is a sequence of nodes (or literals)**. The character "`@`" (pronounced "at") is used to select an attribute node.

Examples (using the `books.xml` file as a context for the query):

- The expression "`//Book`" evaluates to a sequence of all the `Book` element nodes. We use two slashes because we want to *step over* the first element, which is `Books`.
- "`//Book/@Title`" evaluates to a sequence of attribute nodes `Title`.

XPath expressions are case-sensitive, so "`//Book/@Title`" will return the desired nodes while the expressions "`//Book/@title`", "`//book/@Title`" or "`//book/@title`" will not.

We use predicates in XPath expressions to select nodes from the input XML data. Predicates are written between square brackets "`[ ]`". So, to select the authors born after 1950 we would write the expression "`//Author[@YearOfBirth > 1950]`". And to select **only the name** of those authors we would write "`//Author[@YearOfBirth > 1950]/@Name`".

When navigating nodes in an XPath expression, we may use the characters "`..`" and "`.`" to indicate the parent node and current node respectively. Much as we do when navigating the file system in our computer. Continuing with the previous example, if we would like to find **the title of the books** with authors born after 1950 we would write the expression "`//Author[@YearOfBirth > 1950]/../@Title`".

4.4 Iteration and Variable Declaration (FLWOR expressions)

We just finished the last section with the expression "`//Author[@YearOfBirth > 1950]/../@Title`". This expression evaluates to a sequence of attribute nodes. Attribute nodes can be represented according to the XQuery model, but we normally want to have attribute nodes inside element nodes. Based on your execution environment attribute nodes may be serializable or not. An XML document is made up of element nodes (with one root element node) and thus the result of the previous query cannot be serialized into an XML document.

We would need to loop over the sequence of attribute nodes and convert each of them to an element node. A FLWOR (from **f**or-**l**et-**w**here-**o**rder-**r**eturn, pronounced "flower") statement will do the trick! The `for` clause loops through each of the items in a sequence:

```
for $t in //Author[@YearOfBirth > 1950]/../@Title
return <Book>{ $t }</Book>
```

The `let` clause assigns a value to a variable for each of the elements in the `for` clause:

```
for $b in //Author[@YearOfBirth > 1950]/..
let $t := $b/@Title
return <Book>{ $t }</Book>
```

and the `return` clause constructs the resulting element. We can optionally sort the results by title with the `order by` clause:

```
for $b in //Author[@YearOfBirth > 1950]/..
let $t := $b/@Title
order by $t
return <Book>{ $t }</Book>
```

The difference between `for` and `let` clauses is that while the `for` clause binds the variable to each of the items in the sequence, the `let` clause binds the variable only once. Using a different example, in which we want to retrieve the books and their translations

```
for $b in //Book
let $t := $b//Translation
return <Book> { $b/@Title , $t } </Book>
```

the variable `$b` is bound to **each one** of the books in the resulting sequence of evaluating the XPath expression `//Book`. For each of those books, the variable `$t` is bound to **all** the translations (a sequence of `Translation` element nodes) of the given book.

4.5 XQuery and XPath Functions and Operators

XQuery provides a fairly good amount of operations and functions. It is even possible to define custom functions. In this section we provide a brief summary of some of the most commonly used functions, some of which we will use in the examples in later chapters. Refer to reference [8] to find the complete list and explanation. Remember that XQuery and XPath are case-sensitive languages!

`doc(URI)`	Returns the XML data contained in the file or resource indicated by the URI.
`distinct-values(s)`	Returns a sequence of literals without duplicates by first evaluating the value of each node in sequence `s`.
`data(s)`	Converts a sequence of nodes to a sequence of their values.
`name()`	Evaluates to the name of a node.
`starts-with(s1,s2)`	String function. Evaluates to true if string `s1` starts with string `s2`.
`count(s)`	Sequence function. Evaluates to an integer that indicates the number of items in sequence `s`.
`min(s), max(s), sum(s), avg(s)`	Functions that operate on sequences.
`not(exp)`	Boolean function that inverts the value of the parameter.
`concat(s1, s2)`	Function that concatenates the string values of the parameters.
`empty(s)`	Function that returns true if sequence `s` contains no items.
`exists(s)`	Function that returns true if sequence `s` is not empty.
`matches(s, regexp)`	Evaluates to true when the regular expression `regexp` matches the string `s`.

5 XQuery 1 Examples

In this chapter we will go through a few examples that are compatible with XQuery 1. That means that they only use the original five FLWOR clauses and no advanced features introduced in XQuery 3. They will of course work in any environment supporting XQuery 3 since any XQuery 1 statement is a valid XQuery 3 statement.

During the following example queries we will always want one XML document as a result of our query. Remember that the result of an XQuery statement may be a sequence of XML nodes and when this sequence of nodes is serialized to XML, the serialization will depend on the execution environment and the settings. Thus, the query

```
//Book
```

may return the following result:

```
<?xml version="1.0" encoding="UTF-8"?>
<Book Title="Misty Nights" OriginalLanguage="English" Genre="Thriller">
     <Author Name="John Craft" Email="jc@jc.com" YearOfBirth="1948"
             Country="England"/>
     …
</Book>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<Book Title="Archeology in Egypt" OriginalLanguage="English"
      Genre="Educational">
     <Author Name="Arnie Bastoft" Email="bastoft@frei.at"
             YearOfBirth="1971" Country="Austria"/>
     …
</Book>
<?xml version="1.0" encoding="UTF-8"?>
<Book Title="Database Systems in Practice" OriginalLanguage="English"
      Genre="Educational">
     <Author Name="Alan Griff" Email="ag@mit.edu" YearOfBirth="1972"
             Country="USA"/>
     …
</Book>
<?xml version="1.0" encoding="UTF-8"?>
<Book Title="Contact" OriginalLanguage="English" Genre="Science Fiction">
     <Author Name="Carl Sagan" Email="carlsagan@nasa.gov"
             YearOfBirth="1913" Country="USA"/>
     …
</Book>
…
```

As you can see, we get an XML document for each book. In the following examples, we will want one XML document as the result of a query. Thus, we will surround each query with a `result` element. Like this:

```
<result>
{ //Book }
</result>
```

We also make use of the "…" symbol to indicate that there is more XML data that we do not show since it is not relevant to our discussion.

5.1 Basic XPath expressions and loops

What are the titles of all the books?

A first approach to solving this query would be to create an XPath expression that would return the desired values from the attribute `Title` of the element `Book`.

With the following expression

```
<result>
{ //Book }
</result>
```

we obtain a sequence of `Book` elements:

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
   <Book Title="Misty Nights" OriginalLanguage="English" Genre="Thriller">
      <Author Name="John Craft" Email=jc@jc.com
              YearOfBirth="1948" Country="England"/>
      <Edition Year="1987" Price="120">
         <Translation Language="German" Publisher="Kingsly"
                      Price="130"/>
         <Translation Language="French" Publisher="Addison"
                      Price="135"/>
         <Translation Language="Russian" Publisher="Addison"
                      Price="125"/>
      </Edition>
   </Book>
   <Book Title="Archeology in Egypt" OriginalLanguage="English"
         Genre="Educational">
      <Author Name="Arnie Bastoft" Email="bastoft@frei.at"
              YearOfBirth="1971" Country="Austria"/>
      <Author Name="Meg Gilmand" Email="megil@archeo.org"
              YearOfBirth="1968" Country="Australia"/>
      <Author Name="Chris Ryan" Email="chris@egypt.eg"
              YearOfBirth="1944" Country="France"/>
      <Edition Year="1992" Price="250">
         <Translation Language="Swedish" Price="340" Publisher="N/A"/>
         <Translation Language="French" Price="320" Publisher="N/A"/>
      </Edition>
   </Book>
   …
</result>
```

But we want only the contents of the attribute `Title`! We can try the following XPath expression:

```
<result>
{ //Book/@Title }
</result>
```

The XPath expression returns a sequence of attribute nodes (all with the same name), that we add to the `result` element. Depending on the execution environment we may get an error, or perhaps only one of the attributes survives, thus giving a result like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<result Title="Oceanography for Dummies"/>
```

Even if we try to add the title in its own element, with the following statement, we would still be adding a sequence of attribute nodes to one element node.

```
<result>
  <title> { //Book/@Title } </title>
</result>
```

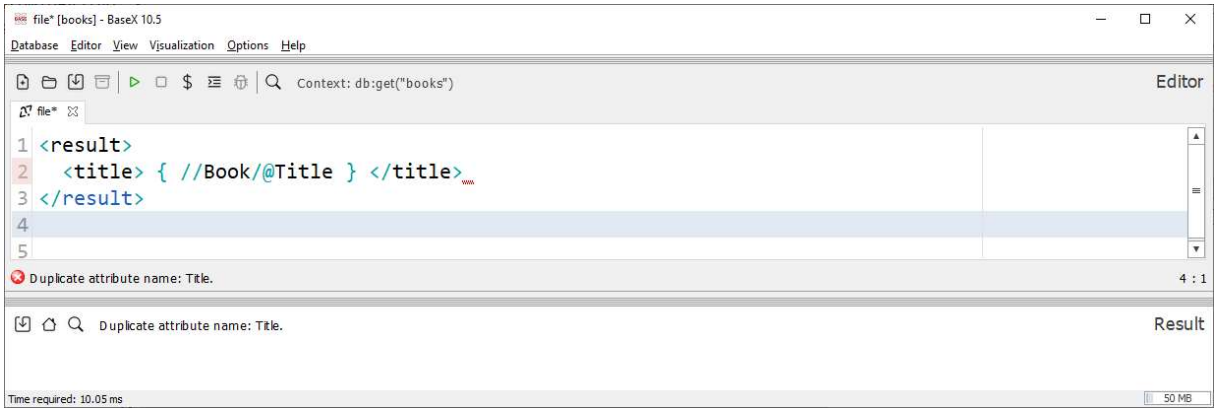
With this query we will get only one result:

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
   <title Title="Oceanography for Dummies"/>
</result>
```

This is how it would look if we run the query in XQuisitor:

If you look at the XML database (in the file `books.xml`) you will see that this title corresponds to the last book in the collection. This happens because the XPath expression `//Book/@Title` results in a sequence of attribute nodes and all of them have the same name, i.e. `Title`. Therefore, when adding the attribute nodes to the element `title`, every new one writes over the previous one, with only the last one not being replaced.

BaseX would instead reject this query with the error message "Duplicate attribute name: Title":



To solve this query, we need to loop through all the `Book` elements and return one title per iteration. With the following query:

```
<result>
{
for $b in //Book
return <title> { $b/@Title } </title>
}
</result>
```

We obtain the following result:

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
   <title Title="Misty Nights"/>
   <title Title="Archeology in Egypt"/>
   <title Title="Database Systems in Practice"/>
   <title Title="Contact"/>
   <title Title="The Fourth Star"/>
   <title Title="The Fifth Star"/>
   <title Title="Våren vid sjön"/>
   <title Title="Dödliga Data"/>
   <title Title="Music Now and Before"/>
   …
</result>
```

We are one step closer to our target but as you can see from the result, we have `title` elements with a `Title` attribute. This happens because in the `return` clause of our query we have selected the attribute and **not the contents of the attribute**! We can extract the value of the attribute node with the built-in `data()` function:

```
<result>
{
for $b in //Book
return <title> { data($b/@Title) } </title>
}
</result>
```

And we finally obtain:

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
   <title>Misty Nights</title>
   <title>Archeology in Egypt</title>
   <title>Database Systems in Practice</title>
   …
</result>
```

How would we sort the titles? Using the `order by` clause

```
<result>
{
for $b in //Book
order by $b/@Title
return <title> { data($b/@Title) } </title>
}
</result>
```

will yield:

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
   <title>Archeology in Egypt</title>
   <title>Contact</title>
   <title>Database Systems in Practice</title>
   …
</result>
```

The function `data()` is actually for sequences, but works even when we only supply one node. The function `string()` would be more suitable since we only have one node.

5.2 Predicates in XPath expressions (1)

Which are the authors of the second book in the database?

To solve this query we need to create an XPath expression with a predicate that will filter out all the `Book` element nodes except for the second:

```
<result>
{ //Book[2]/Author }
</result>
```

As you can see, a predicate containing only a number, evaluates to `true` when the predicate is evaluated with the node in the sequence that occupies the position represented by that number. In the aforementioned example, the predicate `[2]` evaluates to `true` when the second node is evaluated and thus it is the only one that is returned. The predicate `[2]` is an abbreviation of the predicate `[position() = 2]`.

5.3 Predicates in XPath expressions (2)

Which are the authors of the book with the title "Archeology in Egypt"?

To solve this query we need to create an XPath expression with a predicate that will filter out all the `Book` element nodes except for the one we are interested in. Remember that predicates in XPath expressions can only **exclude** nodes from the result:

```
<result>
{ //Book[@Title = "Archeology in Egypt"]/Author }
</result>
```

This produces the following result:

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
   <Author Name="Arnie Bastoft" Email="bastoft@frei.at"
           YearOfBirth="1971" Country="Austria"/>
   <Author Name="Meg Gilmand" Email="megil@archeo.org"
           YearOfBirth="1968" Country="Australia"/>
   <Author Name="Chris Ryan" Email="chris@egypt.eg"
           YearOfBirth="1944" Country="France"/>
</result>
```

5.4 Using functions in queries

How many authors are there in the database?

To solve this query, we will use a function that given a sequence, returns the number of items in the sequence.

```
<result>
{count(//Book/Author)}
</result>
```

Which gives the following result:

```
<?xml version="1.0" encoding="UTF-8"?>
<result>32</result>
```

But this would not solve the question since the same author can appear in many books. We need to remove duplicate values from the sequence of `Author` elements. We do it with the function `distinct-values()` and we need to use the `Name` attribute of the `Author` element since the actual content of the `Author` element (and thus its value) is empty:

```
<result>
{count(distinct-values(//Book/Author/@Name))}
</result>
```

Which gives the result:

```
<?xml version="1.0" encoding="UTF-8"?>
<result>29</result>
```

5.5 Renaming attributes in the result

List all the titles and original language for all the books! Sort the results by language and then by title!

We can return the result using only XML elements with the following query:

```
<result>
{
for $b in //Book
order by $b/@OriginalLanguage, $b/@Title
return <Book>
  <Title>{ string($b/@Title) }</Title>
  <Language>{ string($b/@OriginalLanguage) }</Language>
</Book>
}
</result>
```

Which gives the following result:

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
   <Book>
      <Title>Archeology in Egypt</Title>
      <Language>English</Language>
   </Book>
   <Book>
      <Title>Contact</Title>
      <Language>English</Language>
   </Book>
   …
</result>
```

Or only with attributes with the same name as in the input XML data (`OriginalLanguage` instead of `Language`):

```
<result>
{
for $b in //Book
order by $b/@OriginalLanguage, $b/@Title
return <Book> { $b/@Title, $b/@OriginalLanguage } </Book>
}
</result>
```

Producing the following result:

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
   <Book Title="Archeology in Egypt" OriginalLanguage="English"/>
   <Book Title="Contact" OriginalLanguage="English"/>
   <Book Title="Database Systems in Practice" OriginalLanguage="English"/>
   …
</result>
```

Or with attributes with a **different name** than in the input XML data:

```
<result>
{
for $b in //Book
order by $b/@Language, $b/@Title
return <Book Title = "{ $b/@Title }"
             Language = "{ $b/@OriginalLanguage }"/> }
</result>
```

Which gives:

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
   <Book Language="English" Title="Archeology in Egypt"/>
   <Book Language="English" Title="Contact"/>
   <Book Language="English" Title="Database Systems in Practice"/>
   …
</result>
```

As you can see, depending on the context, an attribute node will be used as an attribute node or as its value. XQuery does this automatically even during other operations as we saw in the previous examples (in the `order by` clause or in a predicate).

5.6 Subqueries and variable declaration

How many books of each genre are there?

First, we write a query that will list all the genres:

```
<result>
{
for $g in //Book/@Genre
return <Genre Name="{ $g }"/>
}
</result>
```

which yields the following result:

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
   <Genre Name="Thriller"/>
   <Genre Name="Educational"/>
   <Genre Name="Educational"/>
   <Genre Name="Science Fiction"/>
   <Genre Name="Science Fiction"/>
   <Genre Name="Novel"/>
   <Genre Name="Thriller"/>
   <Genre Name="Educational"/>
   <Genre Name="Novel"/>
   <Genre Name="N/A"/>
   <Genre Name="Educational"/>
   <Genre Name="Educational"/>
   …
</result>
```

But as we can see, we get each genre repeated as many times as there are books with it. We can use the function `distinct-values()` to remove repetitions from the result:

```
<result>
{
for $g in distinct-values(//Book/@Genre)
return <Genre Name="{ $g }"/>
}
</result>
```

obtaining the result:

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
   <Genre Name="Thriller"/>
   <Genre Name="Educational"/>
   <Genre Name="Science Fiction"/>
   <Genre Name="Novel"/>
   <Genre Name="N/A"/>
</result>
```

The last genre is actually based on the default value from the DTD, so it may not show up if the execution environment does not support DTDs or if the DTD is not available.

Now we can write a nested statement that will list the books of each genre using a predicate to connect the two statements (like a join in SQL):

```
<result>
{
for $g in distinct-values(//Book/@Genre)
order by $g
return <Genre Name="{ $g }">
       {
         for $b in //Book[@Genre = $g]
         return <Book Title="{ $b/@Title }"/>
       }
       </Genre>
}
</result>
```

Result:

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
   <Genre Name="Educational">
      <Book Title="Archeology in Egypt"/>
      <Book Title="Database Systems in Practice"/>
      <Book Title="Music Now and Before"/>
      …
   </Genre>
   <Genre Name="N/A">
      <Book Title="Encore une fois"/>
      <Book Title="Le chateau de mon pere"/>
   </Genre>
   <Genre Name="Novel">
      <Book Title="The Fifth Star"/>
      <Book Title="Våren vid sjön"/>
      <Book Title="Midsommar i Lund"/>
      <Book Title="The Beach House"/>
   </Genre>
```

```
   <Genre Name="Science Fiction">
      <Book Title="Contact"/>
      <Book Title="The Fourth Star"/>
   </Genre>
   <Genre Name="Thriller">
      <Book Title="Misty Nights"/>
      <Book Title="Dödliga Data"/>
   </Genre>
</result>
```

But this was not really what we wanted. We didn't want to see the books. We wanted to count them. And we can do this with a nested XPath expression that finds the books of the current genre:

```
<result>
{
for $g in distinct-values(//Book/@Genre)
order by $g
return <Genre Name="{ $g }"
              NumberOfBooks="{ count(//Book[@Genre = $g]) }"/>
}
</result>
```

Result:

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
   <Genre Name="Educational" NumberOfBooks="7"/>
   <Genre Name="N/A" NumberOfBooks="2"/>
   <Genre Name="Novel" NumberOfBooks="4"/>
   <Genre Name="Science Fiction" NumberOfBooks="2"/>
   <Genre Name="Thriller" NumberOfBooks="2"/>
</result>
```

And an even more condensed version of the previous query, using the `let` clause:

```
<result>
{
for $g in distinct-values(//Book/@Genre)
let $number := count(//Book[@Genre = $g])
order by $g
return <Genre Name="{ $g }" NumberOfBooks="{ $number }"/>
}
</result>
```

5.7 Adding constraints with a `where` clause

Which authors have written thrillers or science fiction?

To solve this question we start with the first step, i.e. list all the authors:

```
<result>
  { //Author }
</result>
```

which gives all the `Author` elements as result, and contains duplicates:

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
   <Author Name="John Craft" Email="jc@jc.com" YearOfBirth="1948"
           Country="England"/>
   <Author Name="Arnie Bastoft" Email="bastoft@frei.at" YearOfBirth="1971"
           Country="Austria"/>
   <Author Name="Meg Gilmand" Email="megil@archeo.org" YearOfBirth="1968"
           Country="Australia"/>
   <Author Name="Chris Ryan" Email="chris@egypt.eg" YearOfBirth="1944"
           Country="France"/>
   …
</result>
```

Now we can modify the previous query to return only the names of the authors with no duplicates and ordered by name:

```
<result>
{
for $a in distinct-values(//Author/@Name)
order by $a
return <Author>{ $a }</Author>
}
</result>
```

Result:

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
   <Author>Alan Griff</Author>
   <Author>Alicia Bing</Author>
   <Author>Andreas Shultz</Author>
  …
</result>
```

Now we need to select only the authors that have written a thriller or a science-fiction book, i.e. the attribute `Genre` of the `Book` element has to be "`Thriller`" or "`Science Fiction`" and one of the authors must be the current one in the loop.

```
<result>
{
for $a in distinct-values(//Author/@Name)
where //Book[@Genre = "Thriller"]/Author/@Name = $a
   or //Book[@Genre = "Science Fiction"]/Author/@Name = $a
order by $a
return <Author>{ $a }</Author>
}
</result>
```

Result:

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
   <Author>Carl Sagan</Author>
   <Author>Jakob Hanson</Author>
   <Author>John Craft</Author>
   <Author>Leslie Brenner</Author>
</result>
```

We have used the `where` clause to specify the required conditions on the books. We could optimize this by putting the right books in a variable:

```
<result>
{
for $a in distinct-values(//Author/@Name)
let $b := //Book[@Genre = ("Thriller", "Science Fiction")]
where $b/Author/@Name = $a
order by $a
return <Author>{ $a }</Author>
}
</result>
```

We could actually put the `let` clause before the `for` clause, which is allowed even thought the order would not be the same as the acronym FLWOR.

Another way would be to first find the books of the current author and then check the genre:

```
<result>
{
for $a in distinct-values(//Author/@Name)
let $b := //Book[Author/@Name = $a]
where $b/@Genre = ("Thriller", "Science Fiction")
order by $a
return <Author>{ $a }</Author>
}
</result>
```

Or we could find the right books before getting the authors (once again putting the `let` clause before the `for` clause):

```
<result>
{
let $books := //Book[@Genre = ("Thriller", "Science Fiction")]
for $a in distinct-values($books/Author/@Name)
order by $a
return <Author>{ $a }</Author>
}
</result>
```

5.8 Conditionals (if – then – else)

Make a list of all the educational books and the authors that have written each book! Show the book title and the authors' name and country! Show only authors that are born after 1950!

We begin by listing all the educational books with

```
<result>
{
for $b in //Book
where $b/@Genre = "Educational"
order by $b/@Title
return <Book>{ $b/@Title }</Book>
}
</result>
```

and we get:

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
   <Book Title="Archeology in Egypt"/>
   <Book Title="Database Systems in Practice"/>
   <Book Title="European History"/>
   <Book Title="Music Now and Before"/>
   <Book Title="Musical Instruments"/>
   <Book Title="Oceans on Earth"/>
</result>
```

We could also have done a similar thing using the following XPath expression:

```
<result>
{
//Book[@Genre = "Educational"]
}
</result>
```

But if you try this expression you will see that you get all the contents (attributes and subelements) of the `Book` elements. As mentioned before, an XPath expression can only be used to select existing nodes and cannot be used to exclude **parts** of those nodes (attributes or subelements). Thus, we cannot return parts of the `Book` element as a result using only an XPath expression. We have the same situation with the `Author` elements in the following expression:

```
<result>
{
for $b in //Book[@Genre = "Educational"]
return <Book Title="{ $b/@Title }"> { $b/Author } </Book>
}
</result>
```

The result has the entire `Author` element:

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
   <Book Title="Archeology in Egypt">
      <Author Name="Arnie Bastoft" Email="bastoft@frei.at"
              YearOfBirth="1971" Country="Austria"/>
      <Author Name="Meg Gilmand" Email="megil@archeo.org"
              YearOfBirth="1968" Country="Australia"/>
      <Author Name="Chris Ryan" Email="chris@egypt.eg"
              YearOfBirth="1944" Country="France"/>
   </Book>
   <Book Title="Database Systems in Practice">
      <Author Name="Alan Griff" Email="ag@mit.edu"
              YearOfBirth="1972" Country="USA"/>
      <Author Name="Marty Faust" Email="marty@nyu.edu"
              YearOfBirth="1970" Country="USA"/>
      <Author Name="Celine Biceau" Email="celine.biceau@tok.cn"
              YearOfBirth="1969" Country="Canada"/>
   </Book>
   …
</result>
```

Now we just need to select only the authors born after 1950 and return only each author's name and country:

```
<result>
{
for $b in //Book[@Genre = "Educational"]
return <Book Title="{ $b/@Title }">
       {
       for $a in $b/Author[@YearOfBirth > 1950]
       return <Author> { $a/@Name,  $a/@Country } </Author>
       }
       </Book>
}
</result>
```

Result:

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
   <Book Title="Archeology in Egypt">
      <Author Name="Arnie Bastoft" Country="Austria"/>
      <Author Name="Meg Gilmand" Country="Australia"/>
   </Book>
   <Book Title="Database Systems in Practice">
      <Author Name="Alan Griff" Country="USA"/>
      <Author Name="Marty Faust" Country="USA"/>
      <Author Name="Celine Biceau" Country="Canada"/>
   </Book>
   <Book Title="Music Now and Before">
      <Author Name="Mimi Pappas" Country="USA"/>
   </Book>
   <Book Title="European History"/>
   <Book Title="Musical Instruments">
      <Author Name="Alicia Bing" Country="Belgium"/>
   </Book>
   <Book Title="Oceans on Earth">
      <Author Name="Linda Evans" Country="USA"/>
      <Author Name="Chuck Morrisson" Country="England"/>
      <Author Name="Kay Morrisson" Country="England"/>
   </Book>
   <Book Title="Oceanography for Dummies">
      <Author Name="Linda Evans" Country="USA"/>
   </Book>
</result>
```

Note how we get the book entitled *"European History"* with no authors. If we wanted to discard this book from the result, we could use a conditional expression:

```
<result>
{
for $b in //Book[@Genre = "Educational"]
let $authors := $b/Author[@YearOfBirth > 1950]
return if (count($authors) > 0)
       then
          <Book Title="{ $b/@Title }">
          {
          for $a in $authors
          return <Author> { $a/@Name,  $a/@Country } </Author>
          }
          </Book>
       else ""
}
</result>
```

which would return the same results as before except for not containing the book entitled *"European History"*. We have also defined a variable `$authors` since we need it twice. Note that the `else` part of the `if-then-else` construct is compulsory, i.e. you cannot avoid it even if, like in this case, you do not need it.

Another way to achieve the same result without using the `if-then-else` construct is by using the `where` clause:

```
<result>
{
for $b in //Book[@Genre = "Educational"]
let $authors := $b/Author[@YearOfBirth > 1950]
where count($authors) > 0
return <Book Title="{ $b/@Title }">
          {
          for $a in $authors
          return <Author> { $a/@Name,  $a/@Country } </Author>
          }
       </Book>
}
</result>
```

Another equivalent condition could be this:

```
where exists($authors)
```

or

```
where not(empty($authors))
```

5.9 Attribute creation with the `attribute` keyword

Show a list of all the authors born before 1940, the number of book editions they have written and the number of different languages each author's books have been translated to! Also show the average price of the book editions for each author! The result shall have the element Author with the following attributes: Name, NumberOfEditions, NumberOfTranslations and AverageEditionPrice. The result shall be sorted by author name!

Let's just start by getting the names of the right authors:

```
<result>
{
for $a in //Author[@YearOfBirth < 1940]
order by $a/@Name
return <Author> { $a/@Name } </Author>
}
</result>
```

Result:

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
   <Author Name="Andreas Shultz"/>
   <Author Name="Carl George"/>
   <Author Name="Carl Sagan"/>
   <Author Name="Christina Ohlsen"/>
   <Author Name="Franc Desteille"/>
   <Author Name="Kostas Andrianos"/>
   <Author Name="Lilian Carrera"/>
   <Author Name="Marie Franksson"/>
   <Author Name="Marie Franksson"/>
   <Author Name="Peter Feldon"/>
   <Author Name="Sam Davis"/>
   <Author Name="Sam Davis"/>
</result>
```

Note that we get duplicate authors and this is because the same author can appear in several books. In order to remove the duplicates we can use the `distinct-values()` function and the `attribute` keyword to recreate the attribute whose value we got from the function:

```
<result>
{
for $a in distinct-values(//Author[@YearOfBirth < 1940]/@Name)
order by $a
return <Author> { attribute Name { $a } } </Author>
}
</result>
```

Result:

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
   <Author Name="Andreas Shultz"/>
   <Author Name="Carl George"/>
   <Author Name="Carl Sagan"/>
   <Author Name="Christina Ohlsen"/>
   <Author Name="Franc Desteille"/>
   <Author Name="Kostas Andrianos"/>
   <Author Name="Lilian Carrera"/>
   <Author Name="Marie Franksson"/>
   <Author Name="Peter Feldon"/>
   <Author Name="Sam Davis"/>
</result>
```

And now that we have eliminated the duplicates we can move on to count the number of editions:

```
<result>
{
for $a in distinct-values(//Author[@YearOfBirth < 1940]/@Name)
let $editions := //Author[@Name = $a]/../Edition
order by $a
return <Author>
         {
           attribute Name { $a },
           attribute NumberOfEditions { count($editions) }
         }
       </Author>
}
</result>
```

We bind the `$editions` variable to a sequence of all the editions of each author independent of which book they appear in.

Result:

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
   <Author Name="Andreas Shultz" NumberOfEditions="1"/>
   <Author Name="Carl George" NumberOfEditions="1"/>
   <Author Name="Carl Sagan" NumberOfEditions="1"/>
   <Author Name="Christina Ohlsen" NumberOfEditions="1"/>
   <Author Name="Franc Desteille" NumberOfEditions="1"/>
   <Author Name="Kostas Andrianos" NumberOfEditions="1"/>
   <Author Name="Lilian Carrera" NumberOfEditions="1"/>
   <Author Name="Marie Franksson" NumberOfEditions="3"/>
   <Author Name="Peter Feldon" NumberOfEditions="1"/>
   <Author Name="Sam Davis" NumberOfEditions="5"/>
</result>
```

We can now count the number of different languages and the average edition price:

```
<result>
{
for $a in distinct-values(//Author[@YearOfBirth < 1940]/@Name)
let $editions := //Author[@Name = $a]/../Edition,
    $languages := distinct-values($editions/Translation/@Language)
order by $a
return <Author>
         {
           attribute Name { $a },
           attribute NumberOfEditions { count($editions) },
           attribute NumberOfLanguages { count($languages) },
           attribute AverageEditionPrice { avg($editions/@Price) }
         }
       </Author>
}
</result>
```

Note that we can reach the languages and the prices by using the `$editions` variable, instead of starting from the `Author` elements again.

Result:

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
   <Author Name="Andreas Shultz" NumberOfEditions="1"
           NumberOfLanguages="12" AverageEditionPrice="650"/>
   <Author Name="Carl George" NumberOfEditions="1"
           NumberOfLanguages="12" AverageEditionPrice="650"/>
   <Author Name="Carl Sagan" NumberOfEditions="1"
           NumberOfLanguages="3" AverageEditionPrice="140"/>
   <Author Name="Christina Ohlsen" NumberOfEditions="1"
           NumberOfLanguages="12" AverageEditionPrice="650"/>
   <Author Name="Franc Desteille" NumberOfEditions="1"
           NumberOfLanguages="5" AverageEditionPrice="65"/>
   <Author Name="Kostas Andrianos" NumberOfEditions="1"
           NumberOfLanguages="12" AverageEditionPrice="650"/>
   <Author Name="Lilian Carrera" NumberOfEditions="1"
           NumberOfLanguages="12" AverageEditionPrice="650"/>
   <Author Name="Marie Franksson" NumberOfEditions="3"
           NumberOfLanguages="1" AverageEditionPrice="56"/>
   <Author Name="Peter Feldon" NumberOfEditions="1"
           NumberOfLanguages="12" AverageEditionPrice="650"/>
   <Author Name="Sam Davis" NumberOfEditions="5"
           NumberOfLanguages="8" AverageEditionPrice="350"/>
</result>
```

5.10 Joining two structures

Which book has at least two authors from the same country?

We can start by trying to retrieve all the books with at least two authors for the same country:

```
<result>
{
for $b in //Book, $a1 in $b/Author, $a2 in $b/Author
where $a1/@Name != $a2/@Name and $a1/@Country = $a2/@Country
order by $b/@Title
return <Book Title="{ $b/@Title }"/>
}
</result>
```

We bind the variable `$b` to each book, and then bind `$a1` and `$a2` to each of the authors of the current book. Then in the `where` clause we specify the conditions between the two authors. Note that all the authors will go through both variables, thus we need to make sure that we don't have the same author in both variables at the same time.

Result:

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
   <Book Title="Database Systems in Practice"/>
   <Book Title="Database Systems in Practice"/>
   <Book Title="Oceans on Earth"/>
   <Book Title="Oceans on Earth"/>
   <Book Title="The Beach House"/>
   <Book Title="The Beach House"/>
</result>
```

As we can see in the result the same book may come in the result several times (based on the amount of pairs of authors that satisfied the condition). To remove these duplicates we can wrap the result in a new query (similar to nesting an SQL SELECT statement in the FROM clause of another):

```
<result>
{
for $t in
  distinct-values(
  for $b in //Book, $a1 in $b/Author, $a2 in $b/Author
  where $a1/@Name != $a2/@Name and $a1/@Country = $a2/@Country
  order by $b/@Title
  return <Book Title="{ $b/@Title }"/>
  /@Title)
return <Book Title="{ $t }"/>
}
</result>
```

Or another way of nesting:

```
<result>
{
for $t in //@Title
let $books :=
  for $b in //Book, $a1 in $b/Author, $a2 in $b/Author
  where $a1/@Name != $a2/@Name and $a1/@Country = $a2/@Country
  return <Book> { $b/@Title } </Book>
where $books/@Title = $t
order by $t
return <Book> { $t } </Book>
}
</result>
```

In this case we assign the result of the nested query to the variable `$books` and get the distinct book titles once again from the original source.

Another way to avoid the duplicates is to use the `exist()` function as illustrated here:

```
<result>
{
for $b in //Book
where exists(
       for $a1 in $b/Author, $a2 in $b/Author
       where $a1/@Name != $a2/@Name
             and $a1/@Country = $a2/@Country
       return 1)
order by $b/@Title
return <Book Title="{ $b/@Title }"/>
}
</result>
```

In this case the nested query checks if there exists at least one pair of authors that qualifies and then returns a symbolic 1 so the function `exists()` will evaluate to `true`. In this way the outer `for` clause goes through each book only once. The same condition can be expressed with an XPath predicate instead of a condition in the `where` clause:

```
<result>
{
for $b in //Book
where exists(for $a in $b/Author
             return $b/Author[@Name != $a/@Name and
                              @Country = $a/@Country])
order by $b/@Title
return <Book Title="{ $b/@Title }"/>
}
</result>
```

5.11 Queries from more than one XML Source

Show the publishers (name and country) of books published in German (translation language)!

To solve this query we will need to use two files, `books.xml` and `publishers.xml`. Up to now we have used the GUI to specify the context (XML data for input) for the query but to solve this query we are going to use the `doc()` function.

As usual, we proceed in steps. First we will find the books translated into German. We can do this in different ways, but we will show, two ways: one using a predicate in the XPath expression and another using the `where` clause.

Using a predicate in the XPath expression:

```
<result>
{
for $book in //Translation[@Language = "German"]/../..
return
<Book> { $book/@Title } </Book>
}
</result>
```

Gives as a result:

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
   <Book Title="Misty Nights"/>
   <Book Title="Contact"/>
   <Book Title="Music Now and Before"/>
   <Book Title="Musical Instruments"/>
   <Book Title="Oceans on Earth"/>
   <Book Title="Le chateau de mon pere"/>
</result>
```

Please note how we have navigated from a `Translation` element node to a `Book` element node by terminating the XPath expression with "`/../..`".

Using a `where` clause:

```
<result>
{
for $book in //Book
where $book//Translation/@Language = "German"
return <Book> { $book/@Title } </Book>
}
</result>
```

Which gives us the same result.

Once we have the books with a translation in German, we go to the second step: list publishers with their name and country.

Note that up to now we have always written XPath expressions starting with "/", this is because, as we said before, we have made use of the GUI (XQuisitor or BaseX) to specify the **context** of our query. For all the previous examples, the context has been the file `books.xml` but now, we need to query another file: `publishers.xml`. Instead of using the GUI for changing the context, we will use the `doc()` function.

```
<result>
{
for $p in doc("publishers.xml")//Publisher
let $c := $p/Address/Country
return <Publisher> { $p/@Name, $c } </Publisher>
}
</result>
```

The `doc()` function allows us to select the input data from an XML file. **Please observe** that you should make the **Base URI** in XQuisitor's GUI point to the directory in which the file `publishers.xml` resides, otherwise we have to specify the full path to the file when calling the function `doc()`!

The previous query gives the following result:

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
   <Publisher Name="ABC International">
      <Country>Germany</Country>
   </Publisher>
   <Publisher Name="Addison">
      <Country>France</Country>
   </Publisher>
   <Publisher Name="Aurora Publ.">
      <Country>Italy</Country>
   </Publisher>
   …
</result>
```

Now we only need to connect the two queries!

```
<result>
{
for $b in //Book,
    $p in doc("publishers.xml")//Publisher
let $c := $p/Address/Country
where $b//Translation/@Language = "German" and
      $b//Translation/@Publisher = $p/@Name
order by $c
return <Publisher> { $p/@Name, $c } </Publisher>
}
</result>
```

But this is wrong! Well, depending of our interpretation of the question. We are finding one translation in German, but the publisher may be from another translation of the same book. We need to make an adjustment so that the both conditions are checked on the same translation. We can use the quantifier `some`:

```
<result>
{
for $b in //Book,
    $p in doc("publishers.xml")//Publisher
let $c := $p/Address/Country
where some $t in $b//Translation satisfies
          ($t/@Language = "German" and $t/@Publisher = $p/@Name)
order by $c
return <Publisher> { $p/@Name, $c } </Publisher>
}
</result>
```

Which gives us the result we wanted:

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
   <Publisher Name="Kingsly">
      <Country>Austria</Country>
   </Publisher>
   <Publisher Name="ABC International">
      <Country>Germany</Country>
   </Publisher>
   <Publisher Name="ABC International">
      <Country>Germany</Country>
   </Publisher>
   <Publisher Name="ABC International">
      <Country>Germany</Country>
   </Publisher>
</result>
```

But, if we look at the result, “ABC International” is repeated three times! We can restructure our solution to not evaluate each publisher once for every book:

```
<result>
{
for $p in doc("publishers.xml")//Publisher
let $c := $p/Address/Country
where some $t in //Translation satisfies
          ($t/@Language = "German" and $t/@Publisher = $p/@Name)
order by $c
return <Publisher> { $p/@Name, $c } </Publisher>
}
</result>
```

We could of course just use a predicate instead of `some`:

```
<result>
{
for $p in doc("publishers.xml")//Publisher
let $c := $p/Address/Country
where exists(//Translation[@Language = "German" and
                           @Publisher = $p/@Name])
order by $c
return <Publisher> { $p/@Name, $c } </Publisher>
}
</result>
```

5.12 Query based on the name of nodes (labels)

Show a list of all attribute names that contain the letter "i".

With this query, we want to show how to access the information about the types of nodes (element, attribute, text, …) and their names. First we write a query that will list all the attribute names:

```
<result>
{
for $a in //@*
order by name($a)
return <attribute> { name($a) } </attribute>
}
</result>
```

Result:

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
   <attribute>Country</attribute>
   …
   <attribute>Country</attribute>
   <attribute>Email</attribute>
   …
   <attribute>Email</attribute>
   <attribute>Genre</attribute>
   …
   <attribute>Genre</attribute>
   <attribute>Language</attribute>
   …
   <attribute>Language</attribute>
   <attribute>Name</attribute>
   …
   <attribute>Name</attribute>
   <attribute>OriginalLanguage</attribute>
   …
   <attribute>OriginalLanguage</attribute>
   <attribute>Price</attribute>
   …
   <attribute>Price</attribute>
   …
</result>
```

As you can see, we use the function `name()` to get the name of the node and the XPath expression "`//@*`" that evaluates to any attribute. We get many duplicates, which we can remove the same way we have done in previous examples, with the function `distinct-values()`:

```
<result>
{
  for $at in distinct-values(
      for $a in //@*
      return <attr> { name($a) } </attr>
    )
  order by $at
  return <attribute>{ $at }</attribute>
}
</result>
```

Result:

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
   <attribute>Country</attribute>
   <attribute>Email</attribute>
   <attribute>Genre</attribute>
   <attribute>Language</attribute>
   <attribute>Name</attribute>
   <attribute>OriginalLanguage</attribute>
   <attribute>Price</attribute>
   <attribute>Publisher</attribute>
   <attribute>Title</attribute>
   <attribute>Year</attribute>
   <attribute>YearOfBirth</attribute>
</result>
```

Now, we only need to add the constraint on the name of the attribute:

```
<result>
{
  for $at in distinct-values(
      for $a in //@*
      return <attr> { name($a) } </attr>
    )
  where matches($at, "i")
  order by $at
  return <attribute>{ $at } </attribute>
}
</result>
```

Result:

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
   <attribute>Email</attribute>
   <attribute>OriginalLanguage</attribute>
   <attribute>Price</attribute>
   <attribute>Publisher</attribute>
   <attribute>Title</attribute>
   <attribute>YearOfBirth</attribute>
</result>
```

If we instead use the function `name()` in the XPath expression, we can simplify the solution:

```
<result>
{
  for $at in distinct-values(//@*/name()[matches(., "i")])
  order by $at
  return <attribute>{ $at } </attribute>
}
</result>
```

5.13 Using computed constructors

In most of the examples in this chapter we have created the output by manually writing the element tags, attribute names, equal signs, quotes, etc. XQuery offers a set of computed constructors in order to create nodes in a simpler way. There are computed constructors for elements, attributes (the one we discussed in section 5.9), comments, etc. The following two expressions produce the same result. The first one does not use any computed constructors, while the second one only uses computed constructors:

```
<result>
{
  for $x in (3 to 5), $y in (2 to 4)
  return <Addition X="{$x}" Y="{$y}">
                  <Result>{$x+$y}</Result>
         </Addition>
}
</result>

element result
{
  for $x in (3 to 5), $y in (2 to 4)
  return element Addition {attribute X {$x},
                           attribute Y {$y},
                           element Result {$x+$y}}
}
```

Both expressions produce the following result:

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
   <Addition X="3" Y="2">
      <Result>5</Result>
   </Addition>
   <Addition X="3" Y="3">
      <Result>6</Result>
   </Addition>
   <Addition X="3" Y="4">
      <Result>7</Result>
   </Addition>
   <Addition X="4" Y="2">
      <Result>6</Result>
   </Addition>
   <Addition X="4" Y="3">
      <Result>7</Result>
   </Addition>
   <Addition X="4" Y="4">
      <Result>8</Result>
   </Addition>
   <Addition X="5" Y="2">
      <Result>7</Result>
   </Addition>
   <Addition X="5" Y="3">
      <Result>8</Result>
   </Addition>
   <Addition X="5" Y="4">
      <Result>9</Result>
   </Addition>
</result>
```

The node name may also be dynamically created when using computed constructors. Here is an example:

```
element result
{
  for $x in ("A","B","C")
  let $y := attribute {$x} {"attvalue"}
  return element {$x} {$y, "textcontent"}
}
```

And the result looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
   <A A="attvalue">textcontent</A>
   <B B="attvalue">textcontent</B>
   <C C="attvalue">textcontent</C>
</result>
```

6 XQuery 3 Examples

All the examples in the previous chapter are based on XQuery 1 and are possible to execute in XQuisitor that only supports XQuery 1. The examples in this chapter use XQuery 3 features and can therefore not be executed in XQuisitor. They can be executed in BaseX.

XQuery 3 adds many new possibilities, among other things a new `group by` clause, a `count` clause, more flexible FLWOR statements, a `switch` expression, dynamic and inline functions, more predefined functions, and higher-order functions.

6.1 Grouping

How many books of each genre are there?

Let's revisit this query that we solved with XQuery 1 in section 5.6. We can start with getting all the books and retrieving their genres:

```
<result>
{
for $g in //Book/@Genre
return <Genre Name="{ $g }"/>
}
</result>
```

Just like before, this will give us each genre once per book:

```
<result>
  <Genre Name="Thriller"/>
  <Genre Name="Educational"/>
  <Genre Name="Educational"/>
  <Genre Name="Science Fiction"/>
  <Genre Name="Science Fiction"/>
  <Genre Name="Novel"/>
  <Genre Name="Novel"/>
  <Genre Name="Thriller"/>
  <Genre Name="Educational"/>
  <Genre Name="Novel"/>
  <Genre Name="Educational"/>
  <Genre Name="Educational"/>
  <Genre Name="Educational"/>
  <Genre Name="Novel"/>
  <Genre Name="Educational"/>
</result>
```

But what if we group the books by genre? We can do this with the `group by` clause where we can define what should be the thing that everything in a group must have in common. And we can even put that common value in a variable:

```
<result>
{
for $b in //Book
group by $g := $b/@Genre
return <Genre Name="{ $g }"/>
}
</result>
```

This will give us a result that only has each genre once. We also get an empty genre if the default value from the DTD does not work. We could easily exclude it of convert it to "N/A" with techniques we saw in earlier examples.

```
<result>
  <Genre Name="Thriller"/>
  <Genre Name="Educational"/>
  <Genre Name="Science Fiction"/>
  <Genre Name="Novel"/>
  <Genre Name=""/>
</result>
```

One thing you may wonder, is "What happened to the variable `$b`?" That variable was one `Book` element per iteration. But as soon as we used the `group by` clause, that variable became a sequence of all the `Book` elements that ended up in the current group. We can therefore use that variable to count the relevant books for each genre:

```
<result>
{
for $b in //Book
group by $g := $b/@Genre
return <Genre Name="{ $g }" NumberOfBooks="{ count($b) }"/>
}
</result>
```

And this will give us the desired result:

```
<result>
  <Genre Name="Thriller" NumberOfBooks="2"/>
  <Genre Name="Educational" NumberOfBooks="7"/>
  <Genre Name="Science Fiction" NumberOfBooks="2"/>
  <Genre Name="Novel" NumberOfBooks="4"/>
  <Genre Name="" NumberOfBooks="2"/>
</result>
```

6.2 Numbering iterations

Number the books and the authors per book!

XQuery 3 introduces a new clause called `count`. It should not be confused with the function `count()`. The `count` clause keeps track of how many times it has been reached during an evaluation of an XQuery statement. We can see if we can number the books using this clause:

```
<result>
{
for $b in //Book
count $bc
return element Book {attribute BookNumber {$bc}, $b/@Title}
}
</result>
```

This produces the following result:

```
<result>
  <Book BookNumber="1" Title="Misty Nights"/>
  <Book BookNumber="2" Title="Archeology in Egypt"/>
  <Book BookNumber="3" Title="Database Systems in Practice"/>
  <Book BookNumber="4" Title="Contact"/>
  <Book BookNumber="5" Title="The Fourth Star"/>
  …
</result>
```

We can see that the variable $bc increments with each iteration. Let's add the authors to the result with a nested XQuery statement:

```
<result>
{
for $b in //Book
let $authors := for $a in $b/Author
                count $ac
                return element Author
                    {attribute AuthorNumber {$ac}, $a/@Name}
count $bc
return element Book {attribute BookNumber {$bc}, $b/@Title, $authors}
}
</result>
```

This gives the following result:

```
<result>
  <Book BookNumber="1" Title="Misty Nights">
    <Author AuthorNumber="1" Name="John Craft"/>
  </Book>
  <Book BookNumber="2" Title="Archeology in Egypt">
    <Author AuthorNumber="1" Name="Arnie Bastoft"/>
    <Author AuthorNumber="2" Name="Meg Gilmand"/>
    <Author AuthorNumber="3" Name="Chris Ryan"/>
  </Book>
  <Book BookNumber="3" Title="Database Systems in Practice">
    <Author AuthorNumber="1" Name="Alan Griff"/>
    <Author AuthorNumber="2" Name="Marty Faust"/>
    <Author AuthorNumber="3" Name="Celine Biceau"/>
  </Book>
```

```
  …
  <Book BookNumber="12" Title="European History">
    <Author AuthorNumber="1" Name="Carl George"/>
    <Author AuthorNumber="2" Name="Peter Feldon"/>
    <Author AuthorNumber="3" Name="Lilian Carrera"/>
    <Author AuthorNumber="4" Name="Auna Gonzales Perre"/>
    <Author AuthorNumber="5" Name="Kostas Andrianos"/>
    <Author AuthorNumber="6" Name="Andreas Shultz"/>
    <Author AuthorNumber="7" Name="Antje Liedderman"/>
    <Author AuthorNumber="8" Name="Christina Ohlsen"/>
  </Book>
  <Book BookNumber="13" Title="Musical Instruments">
    <Author AuthorNumber="1" Name="Sam Davis"/>
    <Author AuthorNumber="2" Name="Alicia Bing"/>
  </Book>
  …
</result>
```

We see that each `count` clause works independently. We could even have multiple `count` clauses in the same XQuery statement, for example to count before and after a condition. In this modified version of the previous solution, we only include the authors from USA, but number them both among all authors and among the USA authors:

```
<result>
{
for $b in //Book
let $authors := for $a in $b/Author
                count $ac
                where $a/@Country = "USA"
                count $usac
                return element Author
                    {attribute AuthorNumber {$ac},
                    attribute USAAuthorNumber {$usac}, $a/@Name}
count $bc
return element Book {attribute BookNumber {$bc}, $b/@Title, $authors}
}
</result>
```

In the result we can see the numbering where Mimi Papas is the second author of the book, but the first author from USA:

```
<result>
  …
  <Book BookNumber="9" Title="Music Now and Before">
    <Author AuthorNumber="2" USAAuthorNumber="1" Name="Mimi Pappas"/>
  </Book>
  …
</result>
```

6.3 Many alternative flows

Categorise the books based on the number of authors as NoAuthorBook, SingleAuthorBook, AuthorPairBook and MultiAuthorBook!

We could of course use an `if-then-else` inside another `if-then-else` to solve this, but let's try the `switch` feature of XQuery 3. We can start with a finding number of authors per book with the following statement:

```
<result>
{
for $b in //Book
let $na := count($b/Author)
return element Book {$b/@Title, attribute Authors {$na}}
}
</result>
```

The result shows the number of authors as an attribute:

```
<result>
   <Book Title="Misty Nights" Authors="1"/>
   <Book Title="Archeology in Egypt" Authors="3"/>
   <Book Title="Database Systems in Practice" Authors="3"/>
   <Book Title="Contact" Authors="1"/>
   <Book Title="The Fourth Star" Authors="1"/>
   <Book Title="The Fifth Star" Authors="1"/>
   <Book Title="Våren vid sjön" Authors="1"/>
   <Book Title="Dödliga Data" Authors="1"/>
   <Book Title="Music Now and Before" Authors="2"/>
   <Book Title="Midsommar i Lund" Authors="1"/>
   <Book Title="Encore une fois" Authors="1"/>
   <Book Title="European History" Authors="8"/>
   …
</result>
```

But we want to use that number to decide the element name instead of only having `Book` elements. Let's use the number of authors in a `switch` expression:

```
<result>
{
for $b in //Book
let $en := switch (count($b/Author))
           case 0 return "NoAuthorBook"
           case 1 return "SingleAuthorBook"
           case 2 return "AuthorPairBook"
           default return "MultiAuthorBook"
return element {$en} {$b/@Title}
}
</result>
```

The result of the `switch` expression is placed in the variable `$en` (short for element name). The result is what we expected:

```
<result>
   <SingleAuthorBook Title="Misty Nights"/>
   <MultiAuthorBook Title="Archeology in Egypt"/>
   <MultiAuthorBook Title="Database Systems in Practice"/>
   <SingleAuthorBook Title="Contact"/>
   <SingleAuthorBook Title="The Fourth Star"/>
   <SingleAuthorBook Title="The Fifth Star"/>
   <SingleAuthorBook Title="Våren vid sjön"/>
   <SingleAuthorBook Title="Dödliga Data"/>
   <AuthorPairBook Title="Music Now and Before"/>
   <SingleAuthorBook Title="Midsommar i Lund"/>
   <SingleAuthorBook Title="Encore une fois"/>
   <MultiAuthorBook Title="European History"/>
   <AuthorPairBook Title="Musical Instruments"/>
   <MultiAuthorBook Title="Oceans on Earth"/>
   …
</result>
```

6.4 Dynamic functions

Find all the books that contain "on", "in" or "for" in their title!

Functions in XQuery 3 can be placed in variables and then get called with the variable, instead for their name. All functions in XQuery can be referred to based on their signature, which is comprised of the name and the number of parameters they accept. We can therefore refer to the `contains` function as `contains#2` and create a variable to refer to it:

```
<result>
{
let $f := contains#2
for $word in ("on","in","for")
for $b in //Book[$f(@Title, $word)]
return element Book {$b/@Title}
}
</result>
```

We go through each word and find the books that have a title with that word and return them:

```
<result>
  <Book Title="Contact"/>
  <Book Title="Oceans on Earth"/>
  <Book Title="Le chateau de mon pere"/>
  <Book Title="Archeology in Egypt"/>
  <Book Title="Database Systems in Practice"/>
  <Book Title="Music Now and Before"/>
  <Book Title="Oceanography for Dummies"/>
</result>
```

We notice that `contains` does not care if the word is part of another word, for example "on" is part of "mon" and "for" is part of "before". We could take care of that and even handle uppercase characters by using other functions lite `tokenize()` and `lower-case()`. We'll do this in the next section.

6.5 Inline functions

XQuery 3 allows us to create anonymous functions inline. Building on the previous example, we could create a function for checking a title. The new function can be placed in a variable in order to be called later:

```
<result>
{
let $f := function($s) {
              let $words := ("on","in","for")
              return tokenize(lower-case($s)) = $words
          }
for $b in //Book[$f(@Title)]
return element Book {$b/@Title}
}
</result>
```

The function that we create and assign to `$f` takes one parameter (a string or something with a string value, like an attribute node), converts it to lower case, tokenizes it and checks if any of the tokens is equal to any of the words in `$words`. The function returns `true` or `false` and can therefore be used as a condition in a predicate. The result includes the right books:

```
<result>
  <Book Title="Archeology in Egypt"/>
  <Book Title="Database Systems in Practice"/>
  <Book Title="Oceans on Earth"/>
  <Book Title="Oceanography for Dummies"/>
</result>
```

6.6 Higher-order functions

In the previous example we used an inline function to find the correct titles. We could have used a higher-order function like `filter()` to achieve the same result:

```
<result>
{
let $f := function($book) {
  let $words := ("on","in","for")
  return tokenize(lower-case($book/@Title)) = $words
}
for $b in filter(//Book, $f)
return element Book {$b/@Title}
}
</result>
```

The function that we create takes a `Book` element and returns `true` or `false` based on the value of its `Title` attribute. But instead of calling the function once per book in a predicate, we use the function `filter()` that takes a sequence and returns a new sequence that contains only the items that caused the function `$f` to return `true`. We don't actually call the function with a parameter. The function `filter()` calls the function `$f` with each item as a parameter.

7 XQuery Update Facility

XQuery offers only functionality for writing queries and returning a result. There is also an extension to XQuery called XQuery Update Facility [9] that adds support for modifying existing XML nodes (inserting new nodes, deleting nodes, replacing nodes or values of nodes and renaming nodes). XQuisitor does not support the XQuery Update Facility, but BaseX does.

As discussed earlier, XPath can only select nodes and literals that already exist and XQuery can create new nodes, but not modify existing ones. With XQuery Update Facility, it is possible to modify existing nodes by adding, removing, replacing or renaming existing nodes without having to recreate everything else. The available operations are `insert`, `delete`, `replace` and `rename`. These operations can be performed with the corresponding expressions as part of a `copy-modify` statement (called `transform` statement in XQUF 1) or a `transform-with` statement (new in XQUF 3). Both types of statements make a copy and modify the copy, so the original remains unchanged. To persist the changes the function `put()` can be used.

The four modification expressions can also be called directly (without a `copy-modify` statement or a `transform-with` statement). In such case, the original is modified and nothing is returned (since the expression creates a "pending updates list"). In BaseX such expressions modify the database, not the corresponding files.

7.1 Transform (Copy-Modify)

Change the genre of the book "The Fifth Star" from Novel to Science Fiction!

In order to change an existing node, without having to recreate everything, we use a `replace` expression. We do this inside a `copy-modify` statement so that we can get a result:

```
copy $b := /Books
modify replace value of node
                     $b/Book[@Title = "The Fifth Star"]/@Genre
                     with "Science Fiction"
return $b
```

The result is the entire element `Books`, but with the specified modification performed:

```
<Books>
  …
  <Book Title="The Fifth Star" OriginalLanguage="English"
        Genre="Science Fiction">
    <Author Name="Leslie Brenner" Email="leslie@yahoo.com"
            YearOfBirth="1945" Country="USA"/>
    <Edition Year="2003" Price="250">
      <Translation Language="Swedish" Publisher="Bästa Bok"
                   Price="270"/>
    </Edition>
  </Book>
  …
</Books>
```

7.2 Transform-with

Change all the Author elements to Writer and remove all the Email attributes!

We can perform multiple changes at once. In a `copy-modify` statement, the `modify` clause can contain a sequence of expressions that become the pending updates list. We can do the required changes with the following statement:

```
copy $b := /Books
modify (for $a in $b//Author return rename node $a as "Writer",
        delete nodes $b//Author/@Email)
return $b
```

Nodes must be renamed one by one, while deletion can be done for a sequence of nodes. Important to note that the renaming does not occur until after the entire `modify` clause has been evaluated. We must therefore refer to the `Author` element nodes with their original name when trying to find the `Email` attribute nodes to delete.

The statement above can be rewritten with the more compact syntax of the `transform-with` statement:

```
/Books transform with {
  (for $a in .//Author return rename node $a as "Writer",
   delete nodes .//Author/@Email)
}
```

The result of the XPath expression `/Books` is modified according to the specified expressions and returned. Since we do not have a variable to refer to the copy being modified, our XPath expressions are relative to the `Books` element.

8 Epilogue

We hope you enjoyed working through the examples! Please, do not hesitate to send comments or questions to the authors so we can improve this document!

/Rafa & nikos

9 References

[1] *XQuisitor*
URL: http://www.ibiblio.org/xml/xquisitor
[2] *BaseX*
URL: http://BaseX.org
[3] *XQuery: An XML query language*. D. Chamberlin.
IBM Systems Journal, Vol 41, No 4, 2002.
[4] *Querying XML, XQuery, XPath, and SQL/XML in Context*. Jim Melton & Stephen Buxton. Morgan Kaufmann, 2006
[5] *XML, Specification*
URL: https://www.w3.org/TR/xml/all/
[6] *XPath, Specification*
URL: https://www.w3.org/TR/xpath/all/
[7] *XQuery, Specification*
URL: http://www.w3.org/TR/xquery/all
[8] *XQuery and XPath Functions and Operators, Specification*
URL: http://www.w3.org/TR/xpath-functions/all
[9] *XQuery Update Facility, Specification*
URL: https://www.w3.org/TR/xquery-update/all/