SU/DSV
KTH/ICT

QUERYING XML DATA WITH XQUERY

v. 2.0

IV1351 - Data Storage Paradigms

Autumn Term 2010

Rafael Cordones Marcos
nikos dimitrakas

Table of contents

1 Introduction *3*
2 XQuisitor *3*
3 Example data *4*
4 XQuery *4*
4.1 Data Model **5**
4.2 Serialization and Deserialization of XML Documents **5**
4.3 XPath Expressions **6**
4.4 Iteration and Variable Declaration (`FLWOR` expressions) **7**
4.5 XQuery and XPath Functions and Operators **8**
5 Example Queries Explained *9*
5.1 Query 1: Basic XPath Expressions and Loops **10**
5.2 Query 2: Predicates in XPath expressions (1) **14**
5.3 Query 3: Predicates in XPath expressions (2) **14**
5.4 Query 4: Using functions in queries **14**
5.5 Query 5: Renaming Attribute Names in the Result **15**
5.6 Query 6: Subqueries and Variable Declaration **16**
5.7 Query 7: Adding Constraints with a WHERE Clause **19**
5.8 Query 8: Conditionals (if – then – else) **21**
5.9 Query 9: Attribute Creation with the `attribute` Keyword **25**
5.10 Query 10: Joining two structures **28**
5.11 Query 11: Queries from more than one XML Source **29**
5.12 Query12: Query Based on the Name of Nodes (Labels) **33**
6 Assignments ..... *Error! Bookmark not defined.*
7 Epilogue *35*
8 References *35*
9 Appendix I: jEdit's Advanced Text Editor (aXe) ..... *Error! Bookmark not defined.*

1 Introduction

This document is a brief introduction to XQuery, a query language for querying XML data. It is based on the article *XQuery: An XML query language* [1] but contains examples that will relate XQuery to other query languages you have seen in this course.

The main contents of this document are:

- An very brief introduction to XQuery
- An introduction to the tool we will use: XQuisitor
- Examples and exercises on using XQuery for querying XML data

It is **strongly recommended** that you **study** the article in reference [1] (especially pages 597 to 609), *"XQuery: An XML query language"*. Throughout this document we will make references and use concepts found in that article. You will therefore not be able to understand this document without having studied the article first. It is also recommended that you read through this entire document before starting to work with the exercises.

To understand the contents of this compendium you also need to have an understanding of XML and related concepts like: XML elements, XML attributes, data definition documents (DTDs), well-formed XML documents, valid XML documents, URLs, URIs, …

2 XQuisitor

XQuisitor is a simple GUI written in Java that allows the user to evaluate XQuery expressions. It's free software [4] so you can download a binary distribution as well as the source code of the application.

On the next page, XQuisitor's GUI is shown. We provide an explanation of the different parts of the GUI:

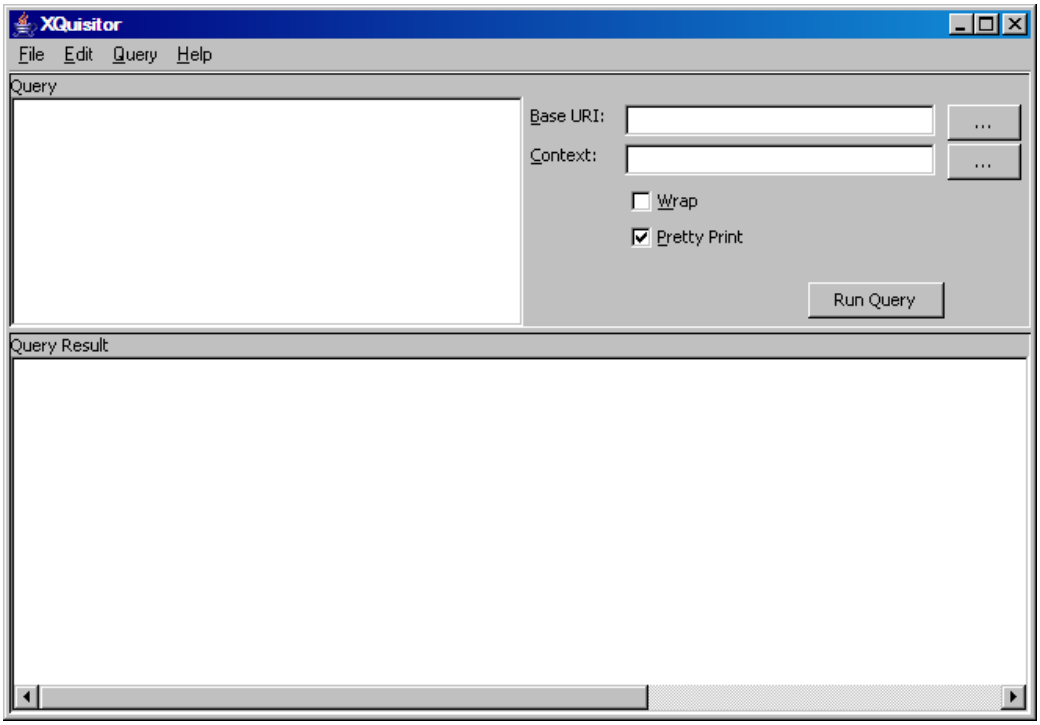
Menu (File, Edit, Query): provides the standard capabilities for loading, saving and printing queries, XML data files and query results.

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 more clear. 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 schema.



3 Example data

We provide you with four files `books.xml`, `books.dtd`, `publishers.xml` and `publishers.dtd` that we will us during the examples in this document. You can find the files in Bilda.

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 builds on the article *"XQuery: An XML query language"* which you can find in reference [1]. We will briefly highlight some of the most important aspects of XQuery but you will need to **study** the article in order to understand XQuery and the rest of this document.

XQuery is a language focused on information retrieval from XML data. The result of evaluating an XQuery expression 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's data model is explained thoroughly in pages 598 to 600 in the article. It is recommended, that before you go through the explanation of the example queries included in Chapter 5, you draw a representation of the file XML data in the file `books.xml`.

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

```
<?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>
```

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 [5] 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.

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 starts with the slash character "`/`" which indicates the root of the input XML document. Every slash in the expressions indicates a next step and **the result of each step is a sequence of nodes**. The character "@" (pronounced "at") is used to select an attribute.

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

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

XPath expressions are case-sensitive, so "`//Book/@Title`" will return the desired values 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 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`".

To navigate the nodes of an XML document, we 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 so if you try to evaluate it you will get an error because they are attribute nodes and not **element** nodes. An XML document is made up of element nodes 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 FLWR (from **`for`**-**`let`**-**`where`**-**`order`**-**`return`**, pronounced "flower" expression will do the trick! The `for` clause loops through **each** of the elements 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 elements 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 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 your own 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 Chapter 5. Refer to reference [3] 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)`	Removes duplicates from sequence `s`
`data()`	Evaluates to the contents of a text node.
`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 amount of elements in sequence `s`.
`min(), max(), sum(), avg()`	Functions that operate on sequences.
`not(exp)`	Boolean function that inverts the value of the
`concat(s1, s2)`	Function that concatenate two sequences or strings.
`empty(s)`	Function that returns true if `s` contains no elements.
`exists(s)`	Function that returns true if `s` is not empty.
`matches(s, regexp)`	Evaluates to true when the regular expression `regexp` matches the string `s`.

For the complete list of core functions and operators in XPath and XQuery see [3] or visit the quick reference at http://www.w3.org/TR/xpath-functions/#quickref.

5 Example Queries Explained

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 query is a sequence of XML nodes and when this sequence of nodes is serialized to XML, each node will be treated as an XML document. Thus, the query

```
//Book
```

will 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>
<?xml version="1.0" encoding="UTF-8"?>
<Book Title="The Fourth Star" OriginalLanguage="English"
      Genre="Science Fiction">
    …
</Book>
…
```

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

```
<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.

Some of the following example queries are the same examples you can find in the XML DB2 compendium. This way you can learn by comparing different ways of doing the same thing but bear in mind that in the XML DB2 laboratory assignment you are learning how to query XML data that is stored in XML documents stored in a **relational database** whereas now we are dealing with XML data stored natively. Learning new things by relating them to things you already know is one of the best-proven methods of learning.

Please, before reading the example queries and the explanation, study the previous chapters and study the article in reference [1] which you can find on the "Articles & Excerpts" compendium.

5.1 Query 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>
...
</result>
```

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

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

But we will get

```
<?xml version="1.0" encoding="UTF-8"?>
<result Title="Le chateau de mon pere"/>
```

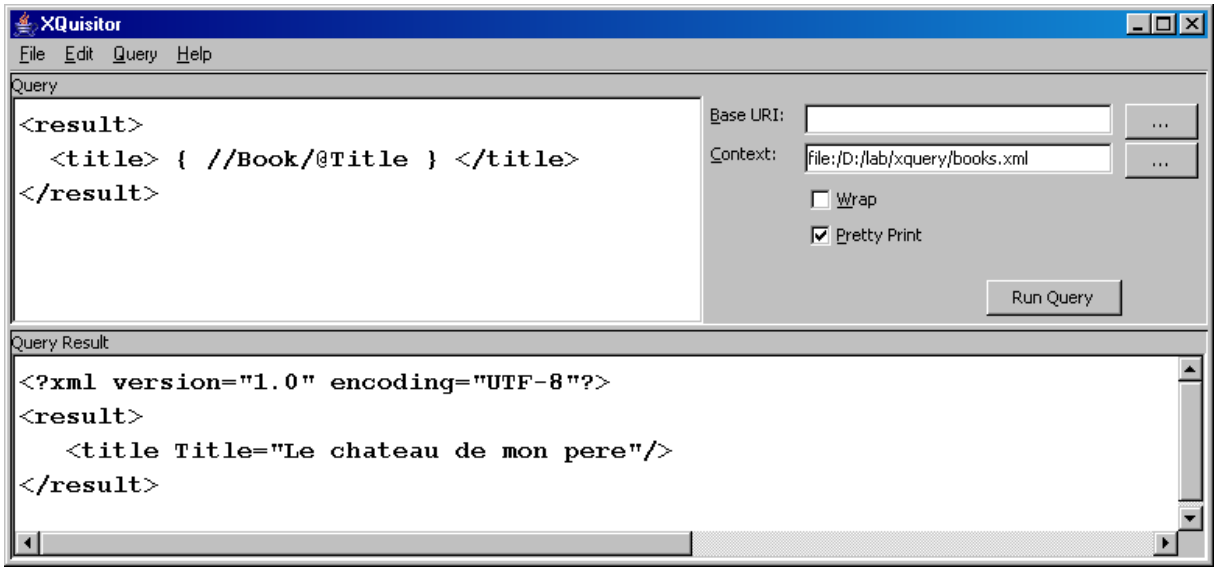
This happens because, as we mentioned earlier, XPath expressions can only select nodes or attributes from the input XML data and **cannot create new nodes**. Besides that, the result of an XQuery expression is always a sequence of nodes of type Element or Document. This means that we need to build a node out of the contents of the `Title` attribute:

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

With this query you will get only one result:

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
   <title Title="Le chateau de mon pere"/>
</result>
```

This is how it would look if you 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 nodes of type Attribute and all the nodes have the same attribute name, i.e. `Title`. Therefore, when serializing the XML document to produce the result, only the last value of the attribute is taken.

To solve this query we need to loop through all the `Book` elements and return the contents of the `Title` attribute. With the following query:

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

We obtain

```
<?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="Våren vid sjön"/>
   <title Title="Dödliga Data"/>
   <title Title="Music Now and Before"/>
   <title Title="Midsommar i Lund"/>
   <title Title="Encore une fois"/>
   <title Title="European History"/>
   <title Title="Musical Instruments"/>
   <title Title="Oceans on Earth"/>
   <title Title="The Beach House"/>
   <title Title="Le chateau de mon pere"/>
</result>
```

We are one step nearer our target but as you can see from the result, we have `title` elements with a `Title` attribute. This happens because in the `return` part of our query we have selected the attribute **but not the contents of the attribute**! We can do this 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 $b in //Book
order by data($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>
```

By default, the ordering is that of the English language but it is possible to order strings according to the order of the letters of other languages. In the Swedish alphabet, for instance, the letters “å”, “ä” and “ö” go after “z” and thus it might be necessary for you to specify a collation. *“A **collation** is a specification of the manner in which character strings are compared and, by extension, ordered”* [2]. This means that if we want to order the previous list using the Swedish collation we will need to indicate this in the `order by` clause using the `collation` keyword in the `order by` clause:

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

Which will yield the same results in this case.

5.2 Query 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.

5.3 Query 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. Remember that predicates in XPath expressions can only **remove** nodes from the result:

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

Results in:

```
<?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 Query 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 of nodes, returns the amount of nodes in the sequence.

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

Which gives the following result:

```
<?xml version="1.0" encoding="UTF-8"?>
<result>31</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 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 Query 5: Renaming Attribute Names 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 query

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

which gives:

```
<?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>
   <Book>
      <Title>Database Systems in Practice</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>
```

giving the 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 Title="Archeology in Egypt" Language="English"/>
   <Book Title="Contact" Language="English"/>
   <Book Title="Database Systems in Practice" Language="English"/>
   …
</result>
```

5.6 Query 6: Subqueries and Variable Declaration

How many books of each genre are there?
This is the first example in which we will use an aggregate function and the GROUP BY clause. Recall from SQL the function `count()`. We will use a function with the same name to solve this query. We want the result to be ordered by the amount of books and by genre.

First we will 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"/>
   <Genre Name="Educational"/>
   <Genre Name="Novel"/>
   <Genre Name="N/A"/>
</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>
```

Now we can build a sub-expression that will list the books of each genre by writing a sub-expression and by a `where` clause to connect them:

```
<result>
{
for $g in distinct-values(//Book/@Genre)
order by $g
return
  <Genre Name="{ $g }">
    {
      for $b in //Book
      where $b/@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"/>
      <Book Title="European History"/>
      <Book Title="Musical Instruments"/>
      <Book Title="Oceans on Earth"/>
   </Genre>
   <Genre Name="N/A">
      <Book Title="Encore une fois"/>
      <Book Title="Le chateau de mon pere"/>
   </Genre>
   <Genre Name="Novel">
      <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>
```

Now we only need to count them:

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

Result:

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
   <Genre AmountOfBooks="6" Name="Educational"/>
   <Genre AmountOfBooks="2" Name="N/A"/>
   <Genre AmountOfBooks="3" Name="Novel"/>
   <Genre AmountOfBooks="2" Name="Science Fiction"/>
   <Genre AmountOfBooks="2" Name="Thriller"/>
</result>
```

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

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

5.7 Query 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 books as result

```
<?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 ordered by name:

```
<result>
{
for $a in //Author
order by $a/@Name
return <Author>{ data($a/@Name) }</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`".

```
<result>
{
for $b in //Book, $a in $b/Author
where $b/@Genre = "Thriller" or
      $b/@Genre = "Science Fiction"
order by $a/@Name
return <Author>{ data($a/@Name) }</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. Note that that the `$a` variable is linked to the `$b` variable in the `for` clause, that is, which is similar to a join between two tables in a relational database.

We can also do this with just one variable and going back to the `Book` element from the `Author` element by using "`..`":

```
<result>
{
for $a in //Author
where $a/../@Genre = "Thriller" or
      $a/../@Genre = "Science Fiction"
order by $a/@Name
return <Author>{ data($a/@Name) }</Author>
}
</result>
```

We can also use an XPath expression with a predicate to do the same:

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

5.8 Query 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) of the `Book` element. As mentioned before, an XPath expression evaluates always to a sequence of nodes and cannot be used to delete **parts** of those nodes (attributes or sub-elements). Thus we cannot return parts of the `Book` element as a result using only an XPath expression.

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

Result:

```
<?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 the 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>
</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 do the same without using the if-then-else construct is by using a function in 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 not(empty($authors))
```

5.9 Query 9: Attribute Creation with the `attribute` Keyword

Show a list of all the authors born before 1940, the amount of book editions they have written and the amount 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, AmountOfEditions, AmountOfTranslations and AverageEditionPrice. The result shall be sorted by author name!

```
<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:

```
<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 { $a },
           attribute AmountOfEditions { count($editions) }
         }
       </Author>
}
</result>
```

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

Result:

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
   <Author Name="Andreas Shultz" AmountOfEditions="1"/>
   <Author Name="Carl George" AmountOfEditions="1"/>
   <Author Name="Carl Sagan" AmountOfEditions="1"/>
   <Author Name="Christina Ohlsen" AmountOfEditions="1"/>
   <Author Name="Franc Desteille" AmountOfEditions="1"/>
   <Author Name="Kostas Andrianos" AmountOfEditions="1"/>
   <Author Name="Lilian Carrera" AmountOfEditions="1"/>
   <Author Name="Marie Franksson" AmountOfEditions="3"/>
   <Author Name="Peter Feldon" AmountOfEditions="1"/>
   <Author Name="Sam Davis" AmountOfEditions="4"/>
</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 { $a },
           attribute AmountOfEditions { count($editions) },
           attribute AmountOfLanguages { 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` again.

Result:

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
   <Author Name="Andreas Shultz" AmountOfEditions="1" AmountOfLanguages="12"
           AverageEditionPrice="650"/>
   <Author Name="Carl George" AmountOfEditions="1" AmountOfLanguages="12"
           AverageEditionPrice="650"/>
   <Author Name="Carl Sagan" AmountOfEditions="1" AmountOfLanguages="3"
           AverageEditionPrice="140"/>
   <Author Name="Christina Ohlsen" AmountOfEditions="1" AmountOfLanguages="12"
           AverageEditionPrice="650"/>
   <Author Name="Franc Desteille" AmountOfEditions="1" AmountOfLanguages="5"
           AverageEditionPrice="65"/>
   <Author Name="Kostas Andrianos" AmountOfEditions="1" AmountOfLanguages="12"
           AverageEditionPrice="650"/>
   <Author Name="Lilian Carrera" AmountOfEditions="1" AmountOfLanguages="12"
           AverageEditionPrice="650"/>
   <Author Name="Marie Franksson" AmountOfEditions="3" AmountOfLanguages="1"
           AverageEditionPrice="56"/>
   <Author Name="Peter Feldon" AmountOfEditions="1" AmountOfLanguages="12"
           AverageEditionPrice="650"/>
   <Author Name="Sam Davis" AmountOfEditions="4" AmountOfLanguages="8"
           AverageEditionPrice="358.75"/>
</result>
```

5.10 Query 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.

5.11 Query 11: Queries from more than one XML Source

Show the publishers of books published in German (translation language) together with their country. Order the publishers by postal code.
To solve this query we will need to use two files, `books.xml` and `publishers.xml`. Up to know 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 written in 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 //Book//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` node to a `Book` 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 results in:

```
<?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>
```

Once we have the books with a translation in German we go to the second step, list publishers with their postal addresses ordered by postal code.
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) 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 the GUI point to the directory in which the file `publishers.xml` resides!

The previous query gives the following result:

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

Which we could also write in a more compact way, making use of a variable:

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

Which gives the result we wanted:

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
   <Publisher Name="Kingsly">
      <Country>Austria</Country>
   </Publisher>
   <Publisher Name="Benton Inc">
      <Country>England</Country>
   </Publisher>
   <Publisher Name="Addison">
      <Country>France</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>
   <Publisher Name="Aurora Publ.">
      <Country>Italy</Country>
   </Publisher>
   <Publisher Name="RP">
      <Country>Russia</Country>
   </Publisher>
   <Publisher Name="SCB">
      <Country>Sweden</Country>
   </Publisher>
</result>
```

But, if you look at the result “ABC International” is repeated three times! Once again we can use the `exists()` function to avoid the duplicates as we showed in a previous query:

```
<result>
{
for $p in doc("publishers.xml")//Publisher
let $c := $p/Address/Country
where exists(
      for $b in //Book
      let $t := $b//Translation
      where $t/@Language = "German" and $t/@Publisher = $p/@Name
      return 1
            )
order by $c
return <Publisher> { $p/@Name, $c } </Publisher>
}
</result>
```

5.12 Query12: 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:

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

6 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

7 References

[1] *XQuery: An XML query language*. D. Chamberlin.
IBM Systems Journal, Vol 41, No 4, 2002.
URL: http://www.research.ibm.com/journal/sj/414/chamberlin.html
[2] *XQuery 1.0: An XML Query Language*. W3C.
URL: http://www.w3.org/TR/xquery/
[3] *XQuery 1.0 and XPath 2.0 Functions and Operators*. W3C.
URL: http://www.w3.org/TR/xpath-functions/
[4] XQuisitor. Elliotte Rusty Harold.
URL: http://www.ibiblio.org/xml/xquisitor/
[5] Free Software. Wikipedia.
URL: http://en.wikipedia.org/wiki/Free_Software
[6] Serialization. Wikipedia.
URL: http://en.wikipedia.org/wiki/Serialization
[7] Uniform Resource Identifier (URI). Wikipedia.
URI: http://en.wikipedia.org/wiki/Uniform_Resource_Identifier