

DEPARTMENT OF COMPUTER
AND SYSTEMS SCIENCES
SU / KTH

DB2 & XML

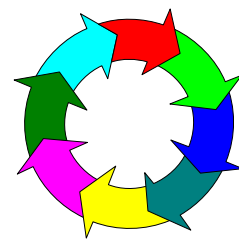
v. 4.0

IS4/2i1242/2i4042

Models and languages for object,
relational and web databases

Spring Term 2006

<http://L238.dsv.su.se/courses/IS4/>



nikos dimitrakas



Table of contents

1 Introduction	3
1.1 Homepage.....	3
1.2 The environment.....	3
1.3 Completed Assignment Requirements.....	3
2 XML & DB2.....	4
2.1 XML.....	4
2.2 XML in DB2.....	6
3 Databases	7
3.1 Books.....	7
3.2 Horse riding	10
4 Compulsory Exercises and Assignments	11
4.1 XML specific functions	11
4.1.1 Path.....	12
4.1.2 Extract functions.....	13
4.1.3 Update function	15
4.1.4 Functions for generating XML - XML/SQL functions.....	15
4.2 Queries against the XML column explained	16
4.2.1 Retrieving data	16
4.2.2 Manipulating data.....	33
4.3 Queries that produce XML explained	35
4.4 Assignments.....	44
5 Voluntary Exercises.....	44
5.1 Create the database	44
5.2 Enable the database for XML (as an XML collection) and compose XML documents.....	45
5.3 Extract XML documents into XML files.....	57
5.4 Store XML documents in an XML column.....	58
6 Internet Resources.....	66
7 Epilogue	66

Table of figures

Figure 1 XML and DTD.....	4
Figure 2 Main components of XML in DB2	7
Figure 3 XML structure for the Book XML files	8
Figure 4 Database model of horse riding database	10
Figure 5 Structure of elements and attributes for the XML documents	46
Figure 6 Entities of the XML structure	52

1 Introduction

This compendium contains the following:

- An introduction to XML
- An introduction to DB2's facilities for handling XML data
- Compulsory exercises on using DB2 for querying and manipulating XML data, as well as producing XML as the result of SQL statements
- Voluntary exercises on using DB2 to transform relational data to XML data

It is strongly recommended that you read through the entire compendium (except from chapter 5) before starting to work with the exercises.

1.1 Homepage

Information about this compendium can be found here:

<http://L238.dsv.su.se/courses/IS4>

The following can be found at this address:

- FAQ - Here there is a list of corrections and explanations that come after the course start.
- Links - Internet resources that can be helpful when working with the compendium.
- Files - The newest version of the compendium and all the files needed to complete the exercises in the compendium (not the solutions of assignments). These files are also available under \\DB-SRV-1\StudentCourseMaterial\IS4 spring 2006\DB2-XML

1.2 The environment

The following facilities will be used:

- IBM DB2 Universal Database version 8.2 fp9a, with XML extender
 - DB2 Command Window
 - DB2 Command Editor
 - DB2 Information Center
- Editor (of your choice)
- Web browser

More information on DB2 and its facilities can be found in the compendium "Introduction to IBM DB2 v.8.2 fp9a for Microsoft Windows XP Professional".

1.3 Completed Assignment Requirements

All the exercises in chapter 4 are compulsory. For the assignments in section 4.4 you have to send in electronically to the conference "MLDB Assignments" in FirstClass the following:

1. SQL statements for all the queries.
2. Execution results for the first 6 queries.

Don't forget to mention the group number and the names of all the group participants.

The deadline for this assignment is the 25th of March 2006.

2 XML & DB2

This chapter introduces XML and DB2's facilities for working with XML. This is not a complete reference of either XML or DB2's XML extender. The following sections only present the aspects of XML and DB2 that are needed to complete the exercises that follow.

2.1 XML

XML (eXtensible Markup Language) is a language with many uses. One of them is to transport data between different systems.

XML consists of two languages, one language for the actual XML documents and one language for specifying how the XML documents¹ should be structured, called DTD² (Document Type Definition). Not all XML documents are associated to DTDs. Here is an example of an XML document and its DTD:

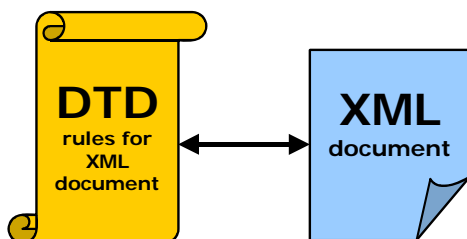
XML Document (saved in a file called "book.xml")

```
<?xml version="1.0"?>
<!DOCTYPE book SYSTEM "d:\dtd\book.dtd">
<book>
  <chapter id="1" date="07/01/1997">
    <section>This is a section in Chapter One.</section>
  </chapter>
  <chapter id="2" date="01/02/1997">
    <section>This is a section in Chapter Two.</section>
    <footnote>A footnote in Chapter Two is here.</footnote>
  </chapter>
  <price date="12/22/1998" time="11.12.13" timestamp="1998-12-22-11.12.13">
    38.281
  </price>
</book>
```

DTD (file "book.dtd")

```
<?xml encoding="US-ASCII"?>
<!ELEMENT book (author*,chapter*,price)>
<!ELEMENT author (#PCDATA)>
<!ELEMENT chapter (section*, footnote*)>
<!ATTLIST chapter id (1|2|3) #REQUIRED
  date CDATA #IMPLIED>
<!ELEMENT price (#PCDATA)>
<!ATTLIST price date CDATA #IMPLIED
  time CDATA #IMPLIED
  timestamp CDATA #IMPLIED>
<!ELEMENT section (#PCDATA)>
<!ELEMENT footnote (#PCDATA)>
```

❗ Both languages are case sensitive!



An XML document can refer to a DTD file.

A DTD file can be associated with many XML documents. When an XML document refers to a DTD file then the XML documents content is supposed to follow the rules defined in the DTD file.

Figure 1 XML and DTD

2.1.1 XML Explanation

Elements:

In the previous example **chapter** is an element. Everything from the **<chapter>** to the **</chapter>** constitutes an element **chapter**.

¹ The term XML document refers to a file with the extension .xml.

² DTD is the older language for defining XML structures. Another "newer" language is XMLSchema, which is somewhat more powerful than DTD.

Every XML document must have a root element, an element that has its start tag in the beginning of the XML document and its end tag at the end of the XML document. This element may appear only once in the XML document.

Attributes:

The element **chapter** has an attribute **id** and an attribute **date**. All attributes of an element appear within the starting tag of the element. Attributes have a value that is within double quotation marks ("").

Structure:

```
<element attribute1="value" attribute2="value2">  
    element content  
</element>
```

The element content can be empty, text or other elements.
If the element content is empty then the element can look like this:

```
<element attribute1="value" attribute2="value2"/>
```

If an end tag is used then no character are allowed between the starting tag and the end tag:

```
<element attribute1="value" attribute2="value2"></element>
```

XML declaration & DOCTYPE element

The first two lines of any XML document are always the XML declaration & the DOCTYPE declaration:

XML declaration:

```
<?xml version="1.0" standalone="no"?>
```

In the XML declaration we define the XML version and whether there is a DTD file with rules for the XML structure or not

DOCTYPE declaration:

```
<!DOCTYPE Book SYSTEM "d:\dtd\book.dtd">
```

The DOCTYPE defines the root element of the XML document and the SYSTEM points out the DTD file for the XML document.

2.1.2 DTD Explanation

The DTD file contains rules to be followed when constructing an XML document.

It defines the elements that can appear in the XML document:

```
<!ELEMENT element-name>
```

It defines the elements that can appear within an element:

```
<!ELEMENT element-name (element2-name)>
```

or the type of the element content:

<!ELEMENT element-name (#PCDATA)>

It also defines the attributes that an element can have, with the appropriate rules (the type of the attribute, whether it has to be there or not, a default value, etc.):

**<!ATTLIST element-name
attribute1-name CDATA #REQUIRED
attribute2-name CDATA #IMPLIED>**

For more help on how to construct an XML document visit one of the following tutorial sites (tutorials for both XML and DTD):

- <http://www.w3schools.com/xml/default.asp>
- <http://www.w3schools.com/dtd/default.asp>
- <http://www.spiderpro.com/bu/buxxmlm001.html>

2.2 XML in DB2

DB2 provides two ways for working with XML documents and XML data³:

- XML collection
- XML column

In addition to that, DB2 implements a large part of the XML functionality described in the SQL 2003 standard. Some of these functions can be used to transform ordinary relational data into XML documents. Some of this functionality overlaps with the functionality provided by an XML collection.

2.2.1 XML collection

When XML data is stored in a relational database, then this database is called an XML collection. DB2 XML extender provides functions for decomposing XML documents into relational data to be stored in the XML collection, and functions for composing XML documents from XML data stored in the XML collection.

Since XML documents are based on hierarchical models and relational databases are based on relational models, it is important to have a mapping between the two models. This mapping can then be used for transformations in both directions. The mapping is defined in DAD (Document Access Definition) files. A DAD file is an XML document that has the extension .dad and follows the rules defined in the file dad.dtd⁴. The DAD file is then used when enabling the XML collection. At that time DB2 verifies that the tables referred in the DAD file exist, otherwise they are created. In chapter 5 there is a more detailed description of how to do all this in practice.

³ With the term XML data we refer to the contents of XML documents, even when the data has been transformed. Data that is going to become the content of an XML document can also be referred to as XML data

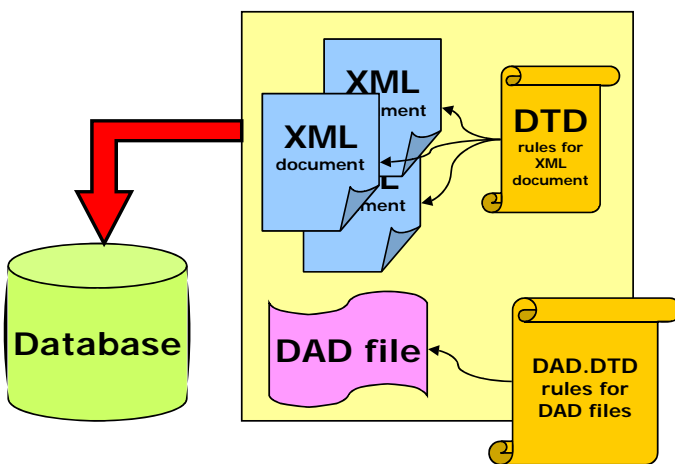
⁴ The file dad.dtd can be found in the following directory:

C:\Program Files\IBM\SQLLIB\samples\db2xml\dtd on all the prepared disks or any computer that has the DB2 XML extender installed.

2.2.2 XML column

XML column is a different approach than XML collection. XML column is an XML enabled database that contains intact XML documents. Those XML documents are stored in a certain table that has a column of one of these three types: XMLCLOB, XMLVARCHAR, XMLFile. That column has to be enabled and associated with a DAD file. In the DAD file there can be a reference to a DTD file for validating any inserted XML documents (XML documents that we insert to the database), and rules for creating side tables⁵ and storing XML data in them. The DTD file must have been registered in the DTD_REF table that is created when a database is being enabled for XML.

There are more details about this in chapter 5. In chapter 4 we will also use an XML column.



All the XML components are stored in the database. The XML documents, DTD files and DAD files are stored in user tables, while the DAD.DTD file is stored in the database manager.

The database can of course contain other non XML specific components too. Those components are not represented in Figure 2.

Figure 2 Main components of XML in DB2

3 Databases

As mentioned earlier this compendium contains some compulsory and some voluntary exercises/assignments. For the compulsory part (described in chapter 4) we will use a database about books and a database about horse riding. For the voluntary part (described in chapter 5) we will use the database about horse riding.

3.1 Books

This database is of the type XML column described in section 2.2.2. There are a number of commands that need to be executed in a certain sequence in order to create this database. We also need the XML data (stored as XML files). 15 XML files, 1 DTD file, 1 DAD file and a script for creating and populating the database can be found at the following network address:

<\\DB-SRV-1\\StudentCourseMaterial\\IS4 spring 2006\\DB2-XML\\Books>

So what do all the files do and what do the commands in the script do?

⁵ A side table is a table that contains data from the XML document. The side tables are used to improve performance when searching through the XML documents. Usually, only some of the XML data is placed in the side tables – the data that is used most frequently when searching.

15 XML files (book01.xml – book15.xml)

These files contain the actual data about the books. More precisely they contain the title, the genre, the original language, data about the authors, data on each edition and each translation and the price. The following figure shows the structure of the XML files.

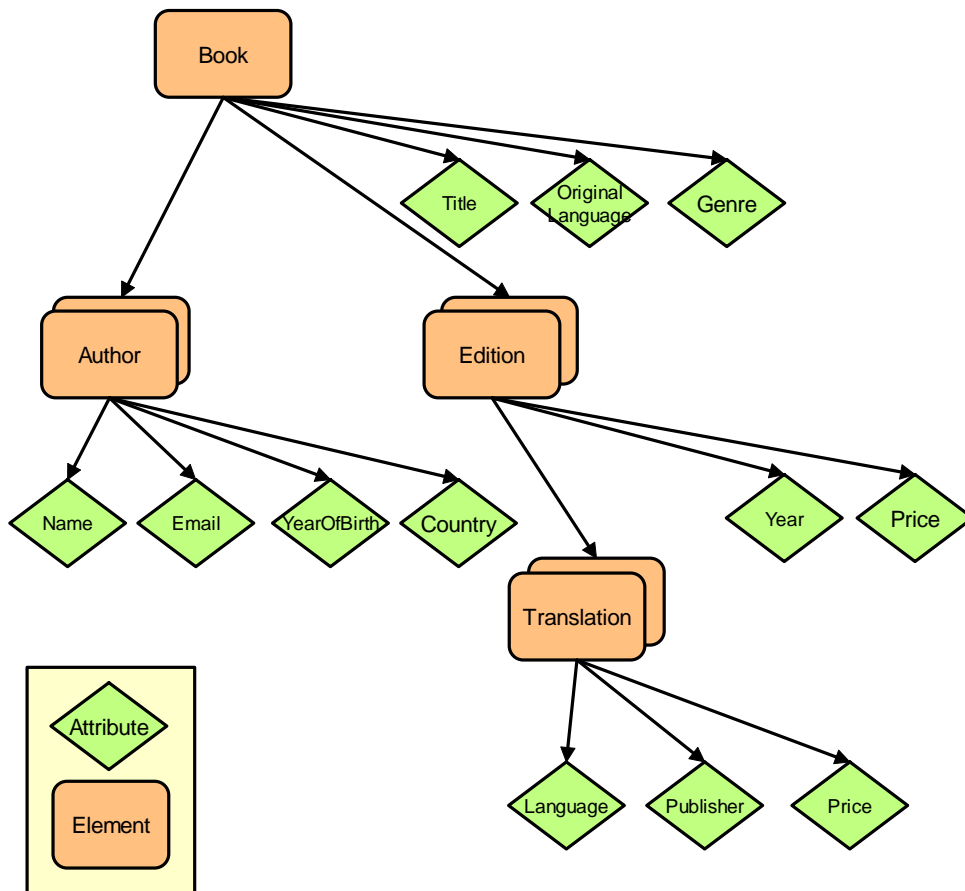


Figure 3 XML structure for the Book XML files

1 DTD file (Book.dtd)

This file contains the rules for the XML structure described in Figure 3.

1 DAD file (bookcolumn.dad)

This file contains the information required by DB2 for creating the XML column where the XML files will be stored. It also provides information about the DTD to be used for validating the inserted XML files.

1 script (bookxmldb.bat)

This script contains all the commands necessary for creating and populating the database (also called “XML column”). In detail the commands included in the script are:

1. **DB2 CREATE DATABASE book on D:**
This command creates a database called **book** on drive D.
2. **Dxxadm enable_db book**
This command tells DB2 that the database **book** will be used for XML data. DB2 creates some infrastructure for the XML data. This infrastructure includes some system tables and some XML specific data-types.
3. **DB2 CONNECT TO book**
Creates a connection to the database **book** that was just created.
4. **DB2 CREATE TABLE xmlcol (xmldoc DB2XML.XMLVARCHAR)**
Create a new table called **xmlcol** with one column called **xmldoc** of a special XML data type.
5. **DB2 INSERT INTO db2xml.DTD_REF VALUES ('D:\xmltemp\Book.dtd', db2xml.XMLClobFromFile('D:\xmltemp\Book.dtd'), 0, 'userX', 'userY', 'userZ')**
This command inserts the DTD file into the database, in the system table **DTD_REF**. This DTD file will be later used for controlling all the incoming XML files.
6. **Dxxadm enable_column book xmlcol xmldoc d:\xmltemp\bookcolumn.dad**
This command tells DB2 which column of what table will be used for inserting the XML files. It also specifies (in the DAD file) the DTD to be used for checking the incoming XML files.
7. **DB2 INSERT INTO xmlcol (xmldoc) VALUES (DB2XML.XMLVarcharFromFile('d:\xmltemp\book01.xml'))**
This is the first of 15 commands that insert the XML files into the database.
8. **DB2 DISCONNECT book**
Finally the scripts disconnects from the database.

In order to run the script you will need to first make sure that DB2 has been started and then copy all the files from the directory **books** into **d:\xmltemp** (if this directory doesn't exist you have to create it). To actually run the script you will need a DB2 Command Window. Go to **d:\xmltemp** (use the command **cd /d d:\xmltemp**). Run the script by using the command **bookxmldb**.

```
DB2 CLP
C:\Program Files\IBM\SQLLIB\BIN>cd /d d:\xmltemp
D:\xmltemp>bookxmldb
```

The script may take a few minutes to complete. When it has finished (the prompt has returned), the database is ready.

3.2 Horse riding

This database is necessary for some of the exercises in chapter 4 as well as for the voluntary exercises in chapter 5.

This database consists of five tables. The tables are connected with foreign keys as shown in Figure 4.

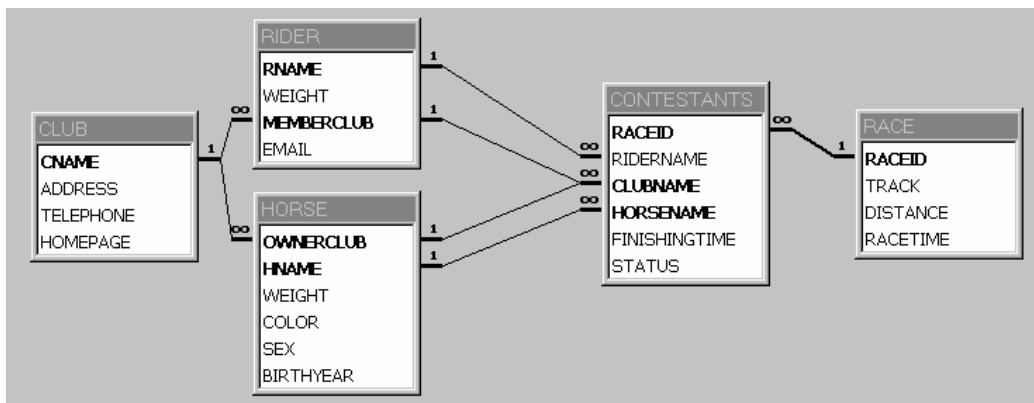


Figure 4 Database model of horse riding database

Scripts for creating and populating the database can be found here:

<\\DB-SRV-1\\StudentCourseMaterial\\IS4 spring 2006\\DB2-XML\\Horse riding>

Simply run the two scripts (first the riding.tables.script and then the riding.insert.script) in the DB2 Command Editor! Pay attention to the statement termination character!

4 Compulsory Exercises and Assignments

This chapter contains a number of exercises that are compulsory for completing the course assignment. For these exercises we will use the two databases that we created in sections 3.1 and 3.2. In the section that follows (section 4.1) you will find a description of some functions that we will use for querying and manipulating data in the XML column and for producing XML from relational data. After that we will go through a few queries that use these functions (sections 4.2 and 4.3). Finally, in section 4.4 you will be given some questions to solve.

4.1 XML specific functions

In this section we will look at the most common functions that DB2 provides for querying and manipulating data in an XML column, as well as functions for composing XML as the result of an SQL statement.

An XML column consists of XML documents stored in a column of a relational table. So, to extract a specific part of the XML documents we need to specify where in the XML structure the desired data is located. We call this the **path** (also known as the *location path*).

There are two groups of functions:

1. **Extract** functions that are used to retrieve values from XML documents.
There are 20 different **extract** functions, grouped in two groups. We will look at some functions from each group. The only difference between the functions of each group is the data type they return (there are 10 data types). The one group of functions returns atomic values, the other returns multiple values.
2. The **update** function, which is used for changing parts of XML documents⁶.
This function can be used to alter attribute and element values of an XML document and returns the altered version of the XML document.

The path is an important parameter for both the **extract** functions and the **update** function. All these function "belong" to the DB2XML schema. This means that when using the functions we must always qualify them with the schema name (we will see how this is done later). Before we look at the functions, we will take a quick look at the path and its syntax (in section 4.1.1).

For data stored in a usual relational database, we have a set of functions for creating XML. These functions can be incorporated in the SELECT clause of an SQL statement in order to transform the result into a user defined XML structure. There are several functions, but we will only look at the following:

1. XMLELEMENT, XMLFOREST and XMLATTRIBUTES.
With these functions we can create elements and attributes.
2. XMLAGG.
With this function we can aggregate several values.

⁶ This function can also be used to delete a part of an XML document. If you wish to delete the entire XML document, then you can simply delete the row where the XML document is stored (with a standard SQL DELETE statement).

3. XMLCONCAT.

With this function we can concatenate several elements.

4.1.1 Path

A path can have the following form⁷:

/element/element/@attribute

There may be one or more elements and there can be an attribute at the end (we denote that it is an attribute with the at-sign (@)). For the structure of the Book XML documents the following are valid paths:

/Book/@Title
/Book/Author/@Name
/Book/Edition
/Book/Edition/Translation/@Language
/Book

This kind of paths is in most cases sufficient. Sometimes, on the other hand it may be necessary (or just quicker) to use the advanced path syntax. This syntax requires the following extras:

- Filtering (only attribute values)
For example the following path finds only Names of Authors from Austria:
/Book/Author[@Country="Austria"]/@Name
- Use of wildcards
The following example finds an attribute Year at any sub-element (denoted by a *) of the element Book
/Book/*/Year
- Support for recursion
This is supported according to the documentation, but not by the actual DB2.

These can of course be combined in creating more complex paths. Here is an example that represents the price on any English book from year 2002 that has been translated into Swedish:

/Book[@OriginalLanguage="English"]/Edition[Year="2002"]/*[Language="Swedish"]/@Price

More information on the path syntax and use can be found in the "XML Extender Administration and Programming" document (pages 144-145) that can be found at the following addresses:

- \\DB-SRV-1\StudentCourseMaterial\IS4 spring 2006\DB2-XML\XML Extender Administration and Programming v8 (c2712340).pdf
- <http://publibfp.boulder.ibm.com/epubs/pdf/c2712340.pdf>

⁷ This is actually the syntax of the simple location path. We will see later that there is an advanced version of the path syntax.

4.1.2 Extract functions

As mentioned earlier there are two groups of extract functions. They all follow the same syntax and take the following two parameters:

1. XML document This is the column name where the XML document is stored
2. Path This is the XML path that will be extracted

The first group contains 10 functions for extracting atomic values from an XML document. The functions are:

<code>extractInteger()</code>	It returns an integer value of the extracted path.
<code>extractSmallint()</code>	It returns a smallint value of the extracted path.
<code>extractDouble()</code>	It returns a double value of the extracted path.
<code>extractReal()</code>	It returns a real value of the extracted path.
<code>extractChar()</code>	It returns a char value of the extracted path.
<code>extractVarchar()</code>	It returns a varchar value of the extracted path.
<code>extractDate()</code>	It returns a date value of the extracted path.
<code>extractTime()</code>	It returns a time value of the extracted path.
<code>extractTimestamp()</code>	It returns a timestamp value of the extracted path.
<code>extractCLOB()</code>	It creates a new XML document that has as its root element the last element that appears in the path parameter. The new XML document is returned as a CLOB . The path sent to this method cannot have an attribute at the end.

The second group contains 10 functions for extracting multiple values from an XML document. This means that the same path can appear more than once in the XML documents. In the XML structure for the Book XML documents the following are examples of paths that may have multiple values:

`/Book/Author/@Name`
`/Book/Edition/Translation`
`/Book/Edition/@Price`

The functions are:

<code>ExtractIntegers()</code>	It returns integer values of the extracted path.
<code>ExtractSmallints()</code>	It returns smallint values of the extracted path.

<code>ExtractDoubles()</code>	It returns double values of the extracted path.
<code>ExtractReals()</code>	It returns real values of the extracted path.
<code>ExtractChars()</code>	It returns char values of the extracted path.
<code>ExtractVarchars()</code>	It returns varchar values of the extracted path.
<code>ExtractDates()</code>	It returns date values of the extracted path.
<code>ExtractTimes()</code>	It returns time values of the extracted path.
<code>ExtractTimestamps()</code>	It returns timestamp values of the extracted path.
<code>ExtractCLOBs()</code>	It creates new XML documents that has as their root element the last element that appears in the path parameter. The new XML documents are returned as CLOBs . The path sent to this method cannot have an attribute at the end.

These functions are most useful together with the **table** function. The **table** function takes one parameter and makes a table out of it. The following example makes a table of all author names in the XML document:

```
table(extractVarchars(xmlDoc, '/Book/Author/@Name')
```

This would of course need to be in a context where `xmlDoc` is defined.

When using one the **extract** functions with the **table** function, then a table with one column is created. This column is named differently depending on the **extract** function used. The column is always named according to the following convention:

"returned" + data type

So in the example above the column of the created table would be named `returnedVarchar`.

All the 20 functions can at times return warnings and errors. These can depend on many reasons. The most common are:

- A path was not found
- A value of a path was incompatible with the type to be extracted
- A path appeared more than once (when using the first group of functions).

The full description of the functions and their associated error and warning codes can be found in the "XML Extender Administration and Programming" document that can be found under Start > Programs > Databases > IBM DB2 > DB2 XML Extender > XML Extender Admin and Programming.

4.1.3 Update function

The **update** function receives three parameters and returns an XML document. The update function works with one XML document at a time. The three parameters are:

- | | |
|-----------------|---|
| 1. XML document | The column name where the XML document is stored |
| 2. Path | This is the path within the XML document that will be updated |
| 3. New value | This is the value that the element or attribute at the defined path will be updated to. |

The **update** function does not affect directly the XML documents stored in the XML column. It merely reads them and creates copies of them. Those copies must replace the original XML documents in the XML column if the changes are to be saved. That has to be done with a standard SQL UPDATE statement. We will see examples of that in section 4.2.2.

It is important to know that the **update** function will update all the occurrences of the defined path to the new value. The following example would change the country of all the authors to "India":
`Update(xmldoc, '/Book/Author/@Country', 'India')`

This would again need to be in a context where `xmlDoc` is defined.

4.1.4 Functions for generating XML - XML/SQL functions

In this section we look at the functions DB2 provides for transforming relational data into XML documents directly in SQL SELECT statements⁸.

Here is a more detailed description of the functions used to compose XML as the output of SQL.

XMLELEMENT (name, content/attributes/sub-elements)	Creates an element of the given name and populates it with the given content, attributes and sub-elements.
---	--

XMLFOREST (arg1 [AS label1], arg2 [AS label2], ..., argN [AS labelN])	Creates a forest of elements from the given list of arguments with the labels as element names. If a label is omitted then the argument's name will be used as the element name.
--	--

XMLATTRIBUTES (arg1 [AS label1], arg2 [AS label2], ..., argN [AS labelN])	Similar to XMLFOREST, but the result is a list of attributes. This function can be used as a parameter to the XMLELEMENT function.
--	--

XMLAGG (element)	Aggregates all the elements generated by the parameter so that they can become sub-elements of another element. XMLAGG is regularly a parameter to an XMLELEMENT and it has an XMLELEMENT as its parameter. XMLAGG can also be combined with the GROUP BY clause of the SQL statement.
-------------------------	--

XMLCONCAT (element1, element2, ..., elementN)	Concatenates the elements of the parameters. Similar to concatenating strings.
--	--

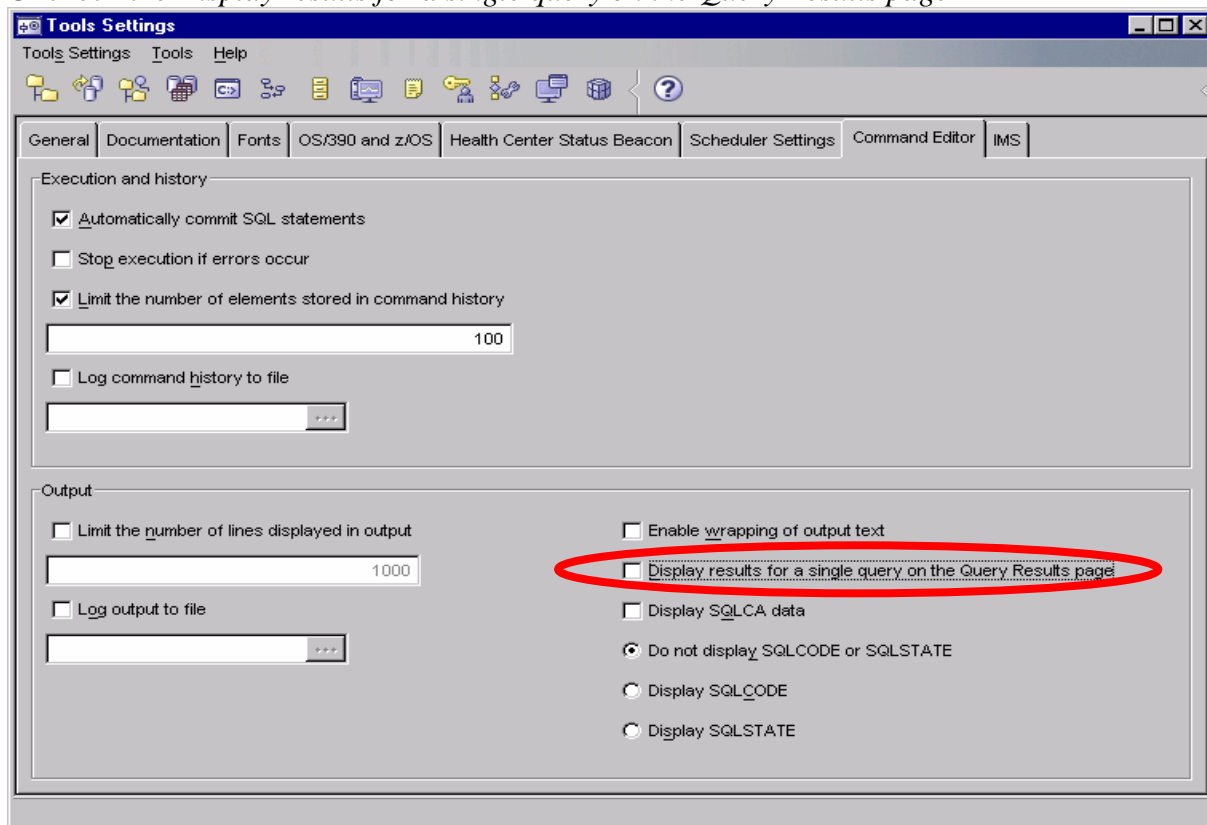
⁸ These functions are actually defined in SQL 2003.

4.2 Queries against the XML column explained

All the functions mentioned in the previous sections can be used in SQL statements. In the sections that follow we will look at some examples that require the use of **extract** and **update** functions.

All the commands in this section can be executed in the DB2 Command Editor. Don't forget to adjust the command termination character since many of the commands will be several rows long. It is also recommended that you change the way the output/result is shown in the DB2 Command Editor. In the options select the following (in the menu Tools > Tools Settings > Command Editor):

Uncheck the *Display results for a single query on the Query Results page*



A collection of all the SQL statements from the following sections exists at:

\\Db-srv-1\\StudentCourseMaterial\\IS4 spring 2006\\DB2-XML\\SQL commands.txt

4.2.1 Retrieving data

In this section we will look at ways to extract data from the XML documents in the XML column. We will first look at some simple examples that only use the first group of extract functions. Then we will look at some examples that use the second group of the extract functions. Finally we will look at some more advanced examples that use the **extractCLOBs** function to perform more complicated queries.

Let's start with the following question:

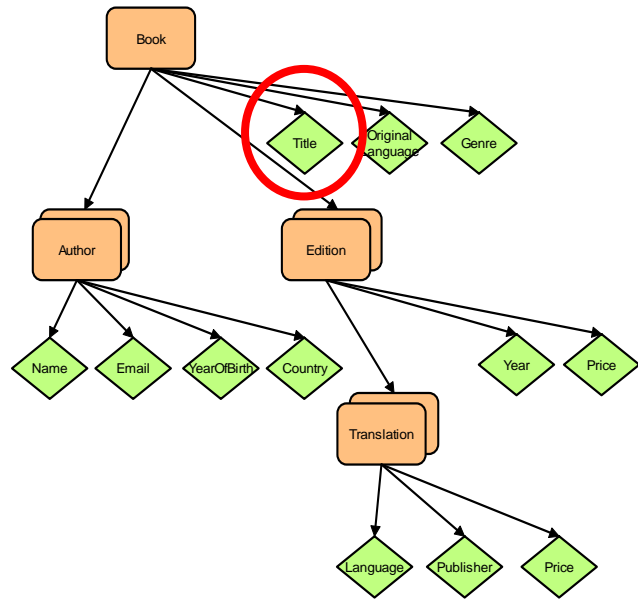
What are the titles of all the books?

To answer this question we have to extract the value of the attribute Title of the element Book.

Since the title is a string value, we will use the function `extractVarchar`. Here is a simple SQL SELECT statement:

```
SELECT  
DB2XML.extractVarchar(xmlcol,'/Book/  
@Title') FROM xmlcol
```

- Run this SQL statement in the DB2 Command Editor! (You will first need to connect to the database book. You do that with the command `connect to book.`)



You should see the following result:

```
Command Editor 1
Command Editor  Selected  Edit  View  Tools  Help
[Icons]
Commands  Query Results  Access Plan
Target: BOOK
SELECT DB2XML.extractVarchar(xmlcol,'/Book/@Title') FROM xmlcol

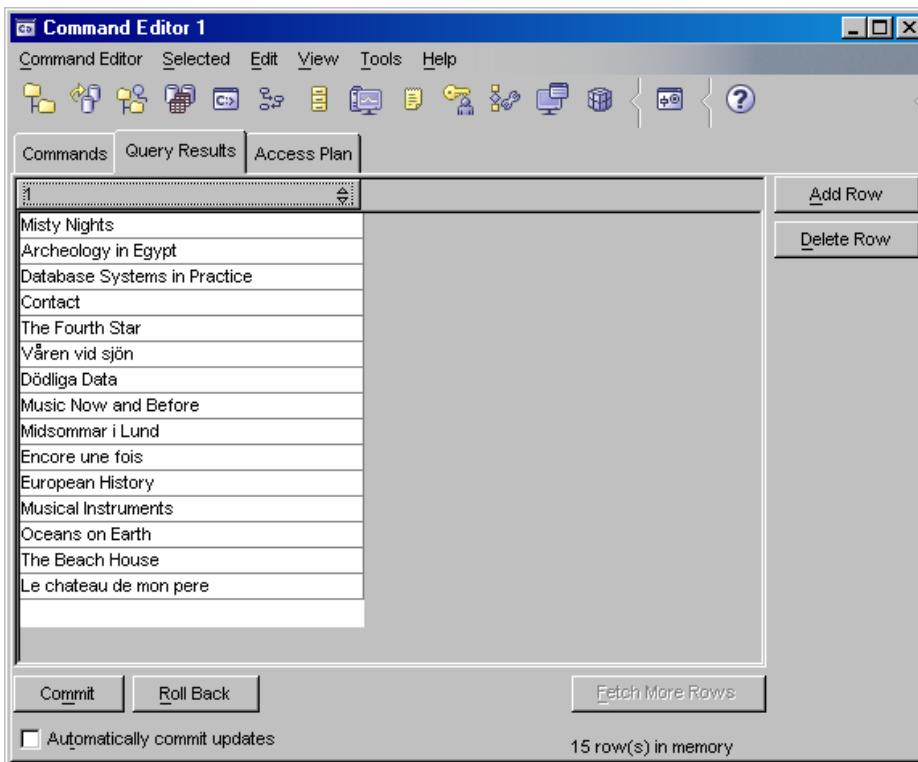
SELECT DB2XML.extractVarchar(xmlcol,'/Book/@Title') FROM xmlcol;
-----
SELECT DB2XML.extractVarchar(xmlcol,'/Book/@Title') FROM xmlcol

1
-----
Misty Nights
Archeology in Egypt
Database Systems in Practice
Contact
The Fourth Star
Våren vid sjön
Dödliga Data
Music Now and Before
Midsommar i Lund
Encore une fois
European History
Musical Instruments
Oceans on Earth
The Beach House
Le chateau de mon pere

15 record(s) selected.

Statement termination character: ;
```

Or (if you haven't configured the DB2 Command Editor according to section 4.2):

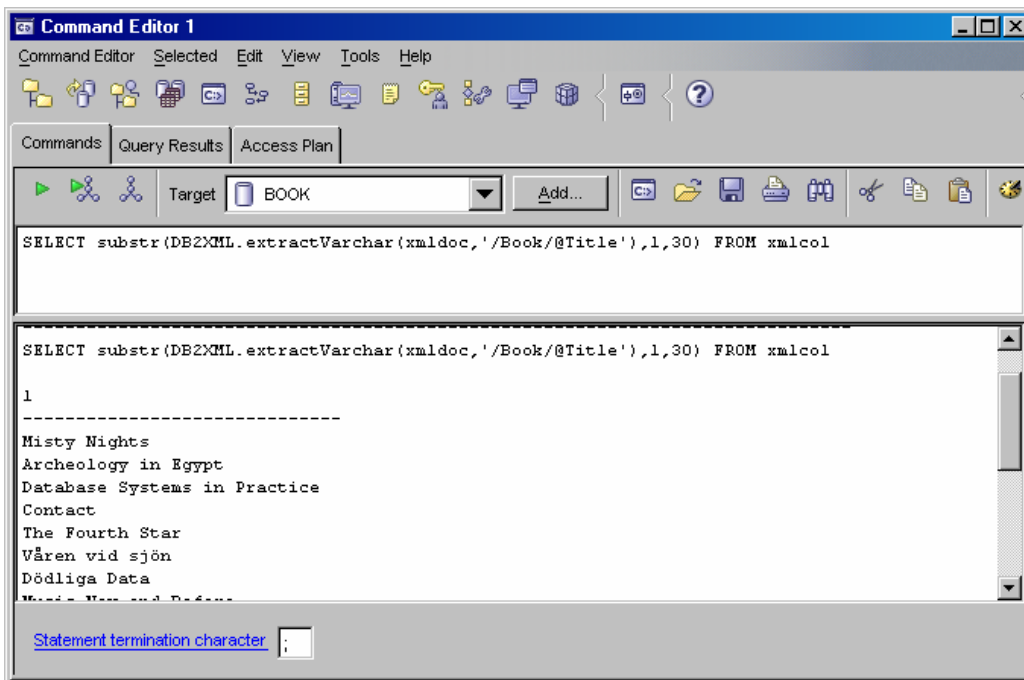


If you scroll right you can see that the column of the result is quite wide. This is because the `extractVarchar` function always returns a 4000-characters long string. To avoid this we can use the function `substr`. This function takes a string and returns a sub-string of a specified length. Here is the same SQL statement as before, but with the `substr` function:

```
SELECT substr(DB2XML.extractVarchar(xmlcol,'/Book/@Title'),1,30) FROM xmlcol
```

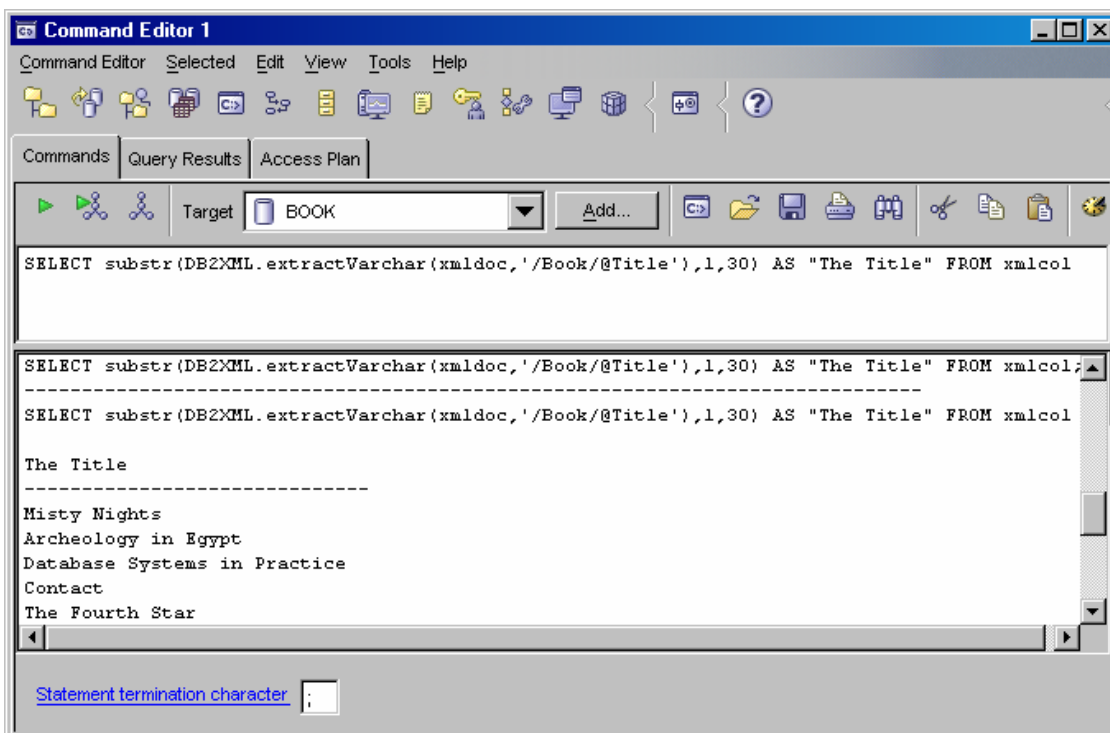
This will create a sub-string 30 characters long starting from the 1st character.

Here is the result:



You can notice in the result that the returned column doesn't have a name. It is therefore automatically called "1" since it is the 1st column. We can assign a name for the column by using the keyword AS. This is how the SQL statement and the result would look then:

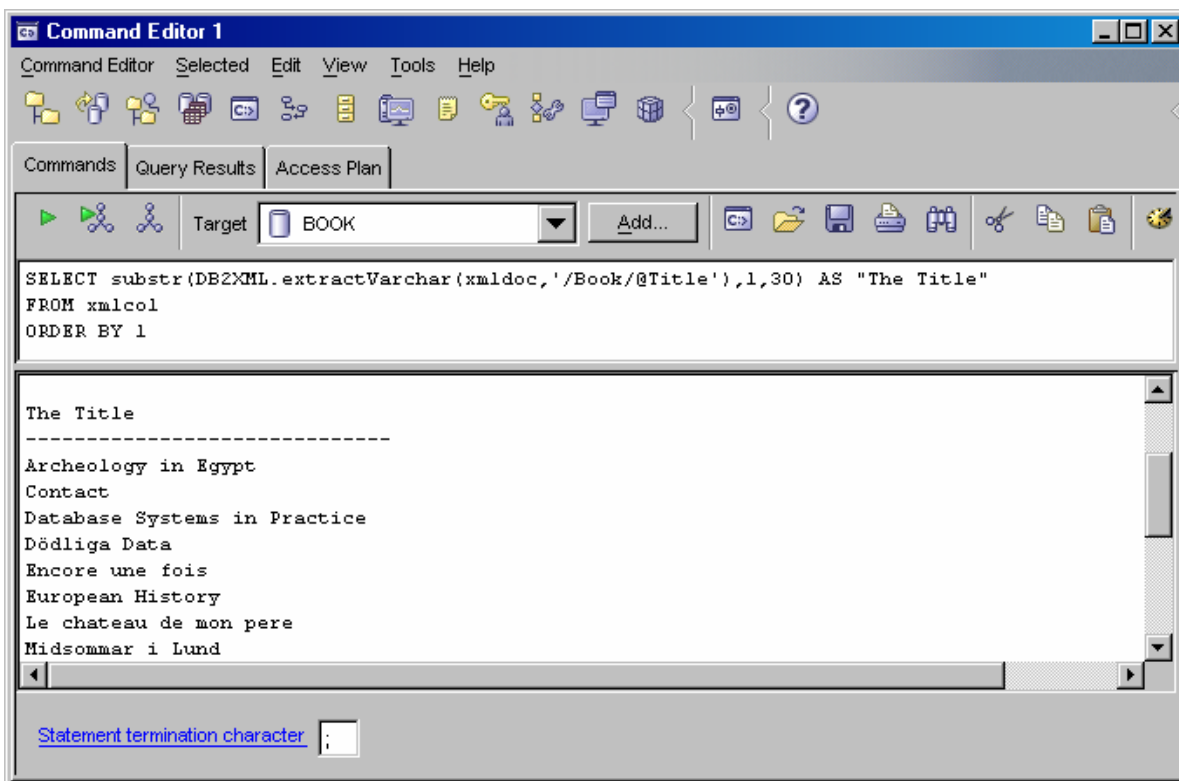
```
SELECT substr(DB2XML.extractVarchar(xml doc, '/Book/@Title'),1,30) AS "The Title" FROM xmlcol
```



Finally we may want to order the results alphabetically. We can then add an ORDER BY clause⁹ to the SQL statement (Observe that the column namea that we define with the keyword AS are not available in the ORDER BY clause.):

```
SELECT substr(DB2XML.extractVarchar(xmlcol,'/Book/@Title'),1,30) AS "The Title"
FROM xmlcol ORDER BY 1
```

And now the result is ordered:



We can look now at something more complicated. The following question for example:

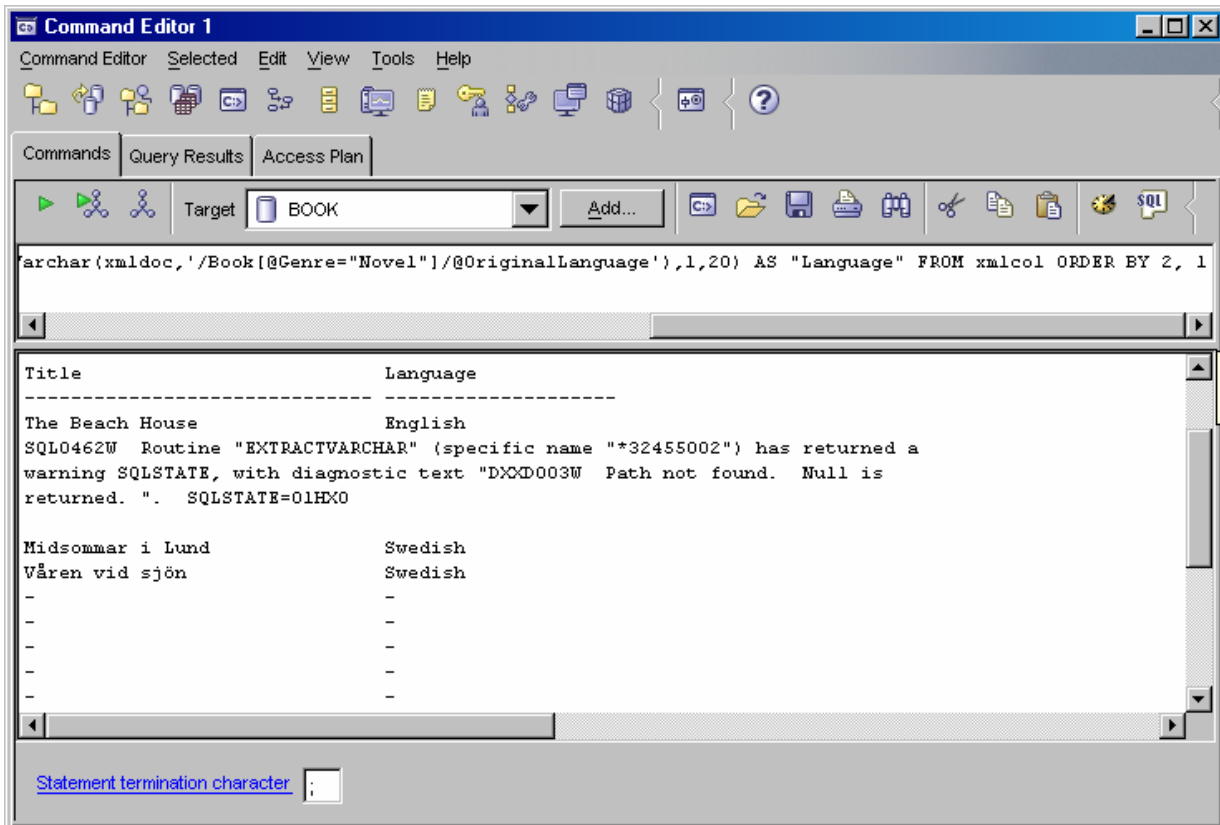
List all the titles and original language for all the novels! Sort the results by language and then by title!

In this case we will have two columns in our result and we also have one condition. Both our columns contain string values, so we will have to use the extractVarchar function. We will look at two ways of representing the condition. We start first with having the condition in the path:

⁹ In order to use a column in the ORDER BY clause, the column has to be 255 character or less (if it is a string). All other types (real, integer, time, etc) can also be used in the ORDER BY clause. The same rule applies to the use of DISTINCT.

```
SELECT substr(DB2XML.extractVarchar(xml doc,'/Book[@Genre="Novel"]/@Title'),1,30)
AS "Title",
substr(DB2XML.extractVarchar(xml doc,'/Book[@Genre="Novel"]/@OriginalLanguage'),1,20) AS "Language" FROM xmlcol ORDER BY 2, 1
```

The result of this SQL statement returns one row for each XML document, even if the condition was not fulfilled:

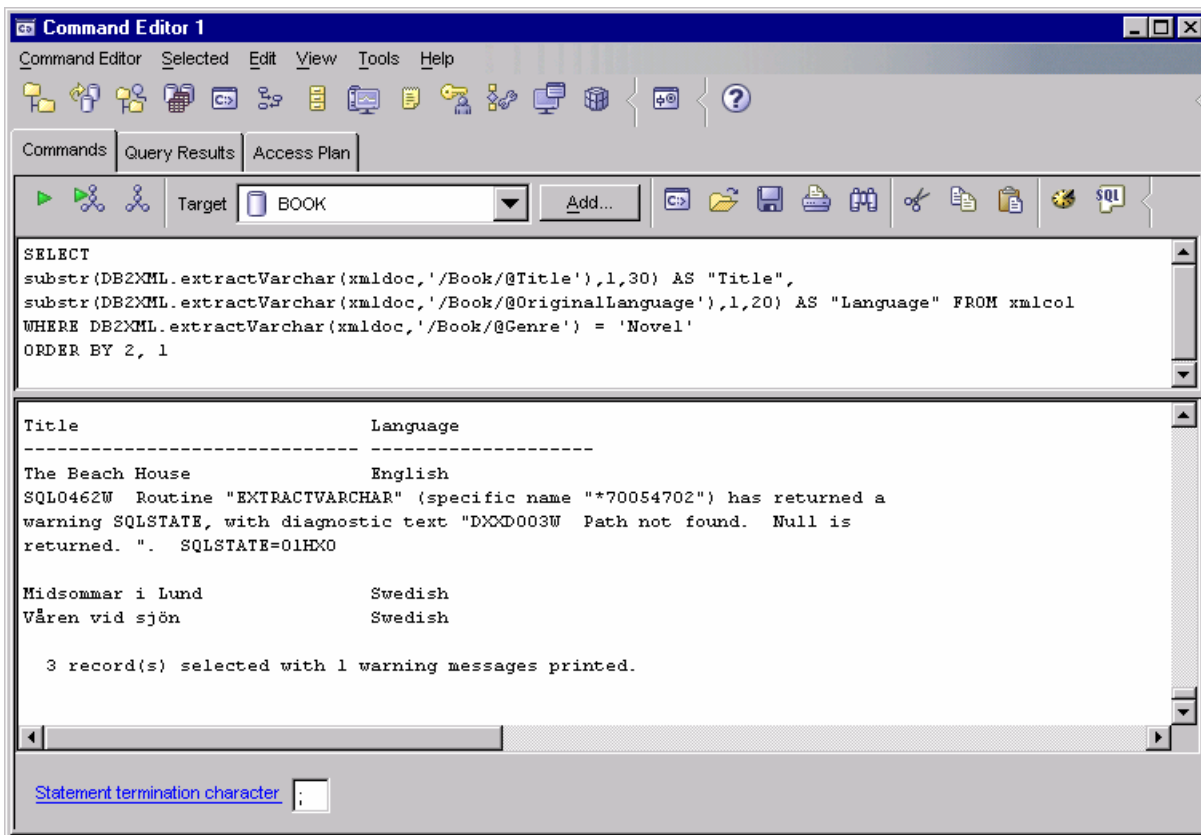


This is the disadvantage of using conditions in the path. Here we can also see that DB2 returns a warning when a path was not found. This is just a limitation of DB2. The correct behaviour would be to use the default value for the missing attribute, which is specified in the DTD. For now we will just ignore this warning.

We will now look at another way of using a condition. We can use an extract function in the WHERE clause of the SQL statement:

```
SELECT
substr(DB2XML.extractVarchar(xml doc,'/Book/@Title'),1,30) AS "Title",
substr(DB2XML.extractVarchar(xml doc,'/Book/@OriginalLanguage'),1,20) AS "Language" FROM
xmlcol
WHERE DB2XML.extractVarchar(xml doc,'/Book/@Genre') = 'Novel'
ORDER BY 2, 1
```

This version returns only three rows (one for each XML document that fulfilled the condition):



We can of course use aggregate functions to answer questions like this one:

How many books of each genre are there?

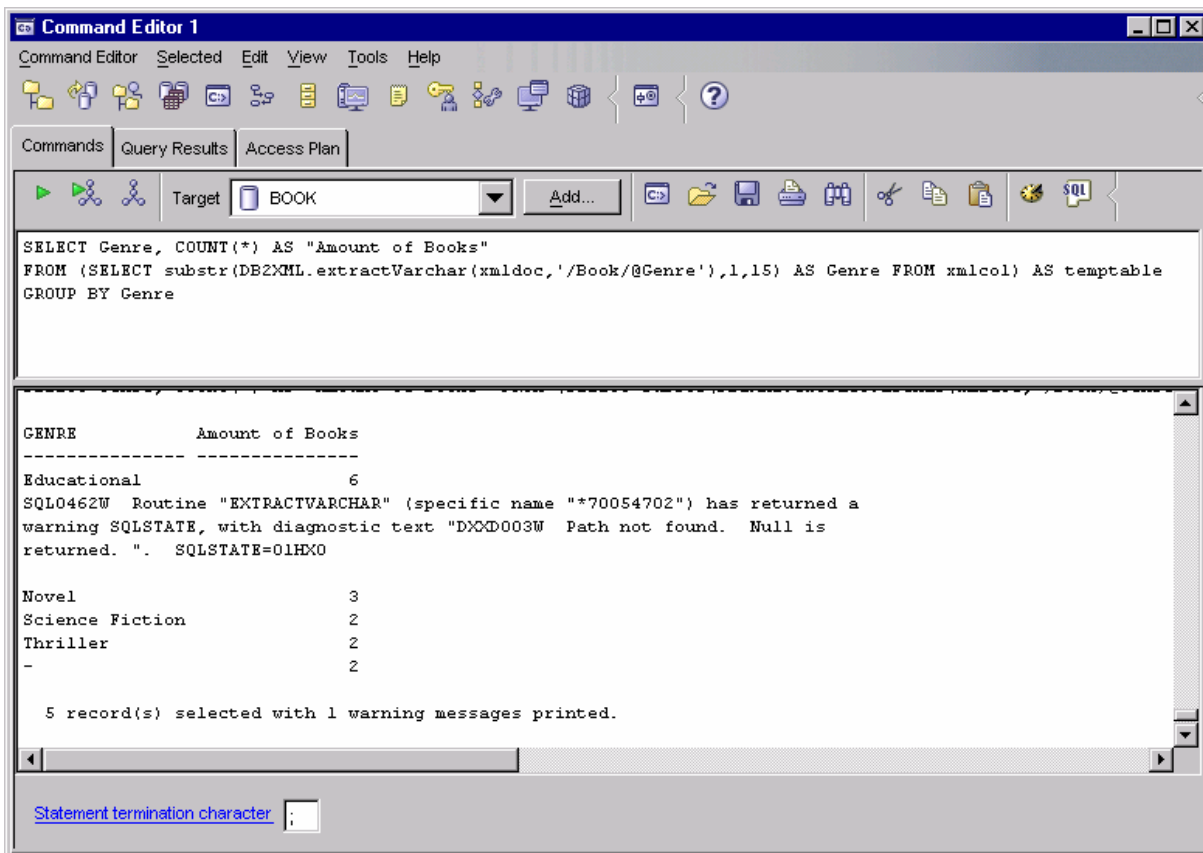
We can then use the COUNT function and the GROUP BY clause to solve this. The only problem is that the column we want to use for grouping doesn't exist from the beginning. We must therefore break the query into two. First we must create a table with all the genres from all the XML documents and then work with that table. This is the first part:

```
SELECT substr(DB2XML.extractVarchar(xmlcol,'/Book/@Genre'),1,15) AS Genre FROM
xmlcol
```

We can now use this part in the FROM clause of a new SELECT statement:

```
SELECT Genre, COUNT(*) AS "Amount of Books"
FROM (SELECT substr(DB2XML.extractVarchar(xmlcol,'/Book/@Genre'),1,15) AS Genre
FROM xmlcol) AS temptable
GROUP BY Genre
```

This will produce the following result:



So far we have only used paths that appeared only once in each XML document. The following question requires data from paths with multiple values:

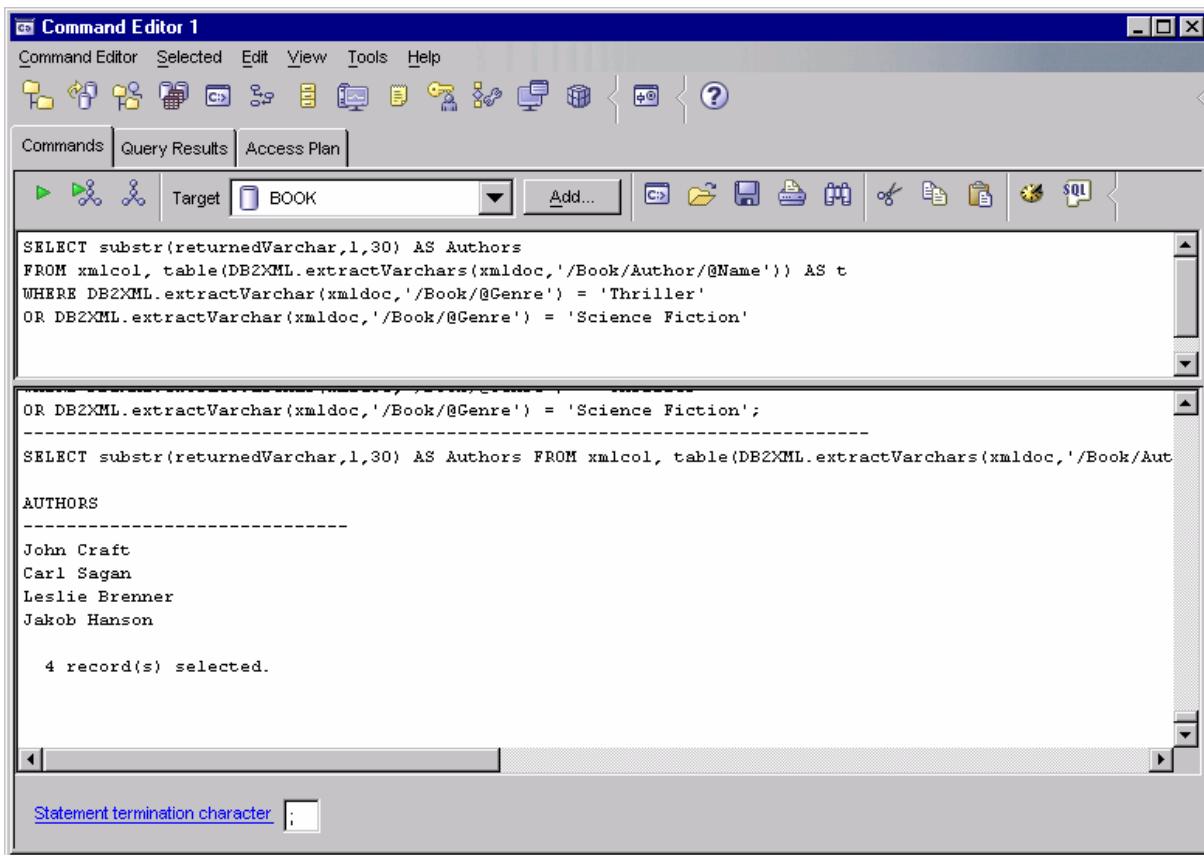
Which authors have written thrillers or science fiction?

To solve this we will need to use the `extractVarchars` function. We will also use the `extractVarchar` function for checking the conditions:

```
SELECT substr(returnedVarchar,1,30) AS Authors
FROM xmlcol, table(DB2XML.extractVarchars(xml doc, '/Book/Author/@Name')) AS t
WHERE DB2XML.extractVarchar(xml doc, '/Book/@Genre') = 'Thriller'
OR DB2XML.extractVarchar(xml doc, '/Book/@Genre') = 'Science Fiction'
```

Since we have to use one of plural `extract` functions, we also have to use the `table` function to capture the result. In the `FROM` clause we must have first the table `xmlcol` and then the `table` function, otherwise the `extractVarchars` function in the `table` function will not know where the `xml doc` comes from. The result of the `table` function is also given a name (t) with the keyword `AS`.

The result of this SQL statement is the following:



The screenshot shows a window titled "Command Editor 1" with a menu bar (Command Editor, Selected, Edit, View, Tools, Help) and a toolbar. Below the toolbar are tabs for "Commands", "Query Results", and "Access Plan". The "Query Results" tab is active, showing a target table of "BOOK". The SQL query in the editor is:

```
SELECT substr(returnedVarchar,1,30) AS Authors
FROM xmlcol, table(DB2XML.extractVarchars(xmldoc,'/Book/Author/@Name')) AS t
WHERE DB2XML.extractVarchar(xmldoc,'/Book/@Genre') = 'Thriller'
OR DB2XML.extractVarchar(xmldoc,'/Book/@Genre') = 'Science Fiction'
```

The results pane shows the output of the query:

```
OR DB2XML.extractVarchar(xmldoc,'/Book/@Genre') = 'Science Fiction';
-----
SELECT substr(returnedVarchar,1,30) AS Authors FROM xmlcol, table(DB2XML.extractVarchars(xmldoc,'/Book/Aut
AUTHORS
-----
John Craft
Carl Sagan
Leslie Brenner
Jakob Hanson

4 record(s) selected.
```

At the bottom, there is a "Statement termination character" field with a semicolon (;) entered.

We could of course return all the details of the authors instead of just the name, but if we would try to do this with three **table** functions, we would risk getting invalid results. The following query for example would not work:

```
SELECT substr(t1.returnedVarchar,1,30) AS Author,
returnedInteger as Year,
substr(t3.returnedVarchar,1,15) AS Country
FROM xmlcol,
table(DB2XML.extractVarchars(xmldoc,'/Book/Author/@Name')) AS t1,
table(DB2XML.extractIntegers(xmldoc,'/Book/Author/@YearOfBirth')) AS t2,
table(DB2XML.extractVarchars(xmldoc,'/Book/Author/@Country')) AS t3
WHERE DB2XML.extractVarchar(xmldoc,'/Book/@Genre') = 'Thriller'
OR DB2XML.extractVarchar(xmldoc,'/Book/@Genre') = 'Science Fiction'
```

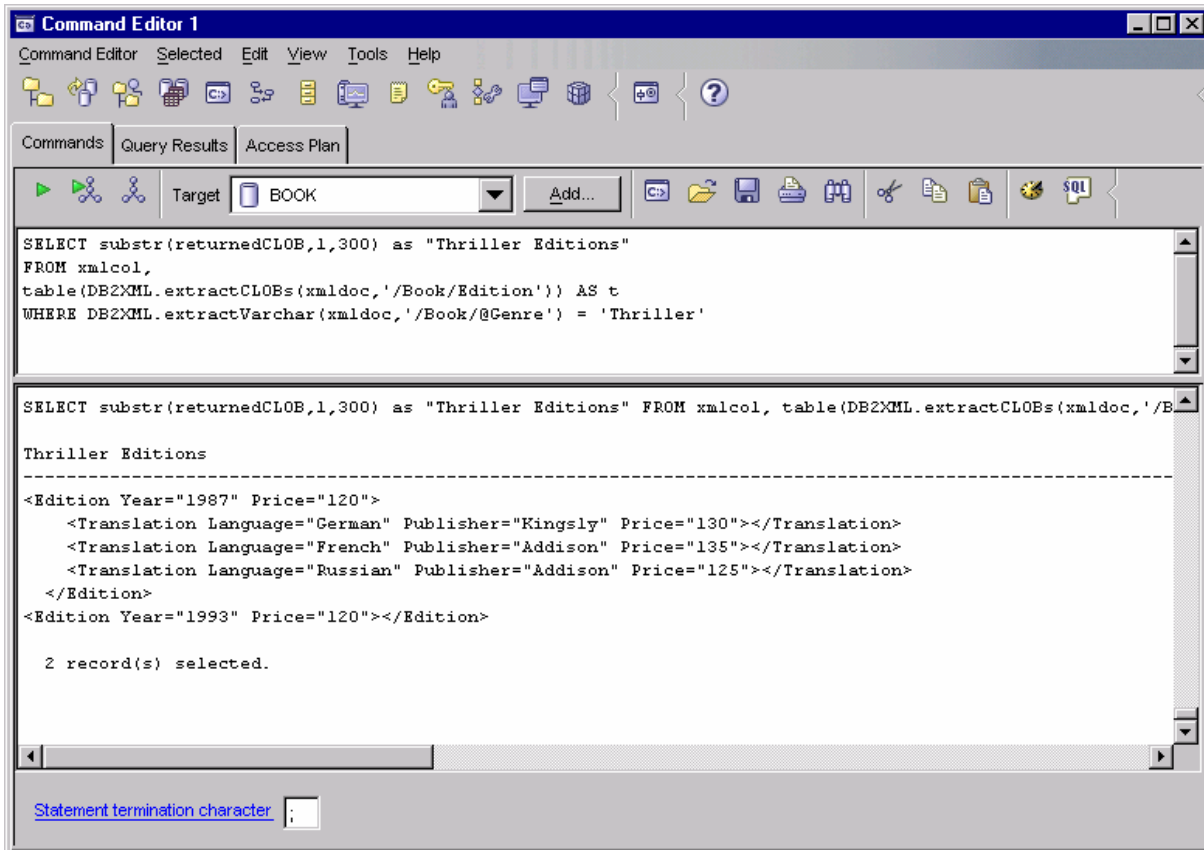


The reason is that the three table functions would be joined without any condition (like a Cartesian product), so if a book has 2 authors we would get 8 ($2*2*2$) combinations of the two names with the two years of birth and the two countries. Similarly if a book would have five authors there would be 125 combinations. To avoid this, we have to use the **extractCLOBs** function instead!

But first, let's see what the **extractCLOBs** function does. If we want to extract a part of an XML document as a smaller XML document we can use the **extractCLOBs** function. In the example that follows we extract the **Edition** elements of all the thrillers as new XML documents:

```
SELECT substr(returnedCLOB,1,300) as "Thriller Editions"  
FROM xmlcol, table(DB2XML.extractCLOBs(xmldoc,'/Book/Edition')) AS t  
WHERE DB2XML.extractVarchar(xmldoc,'/Book/@Genre') = 'Thriller'
```

The function `substr` is only used to make the result smaller, since we know that the new XML documents are not that big. The path used in the `extractCLOBs` function does not have an attribute at the end. It has instead the element that is to be the root element of the new XML documents:

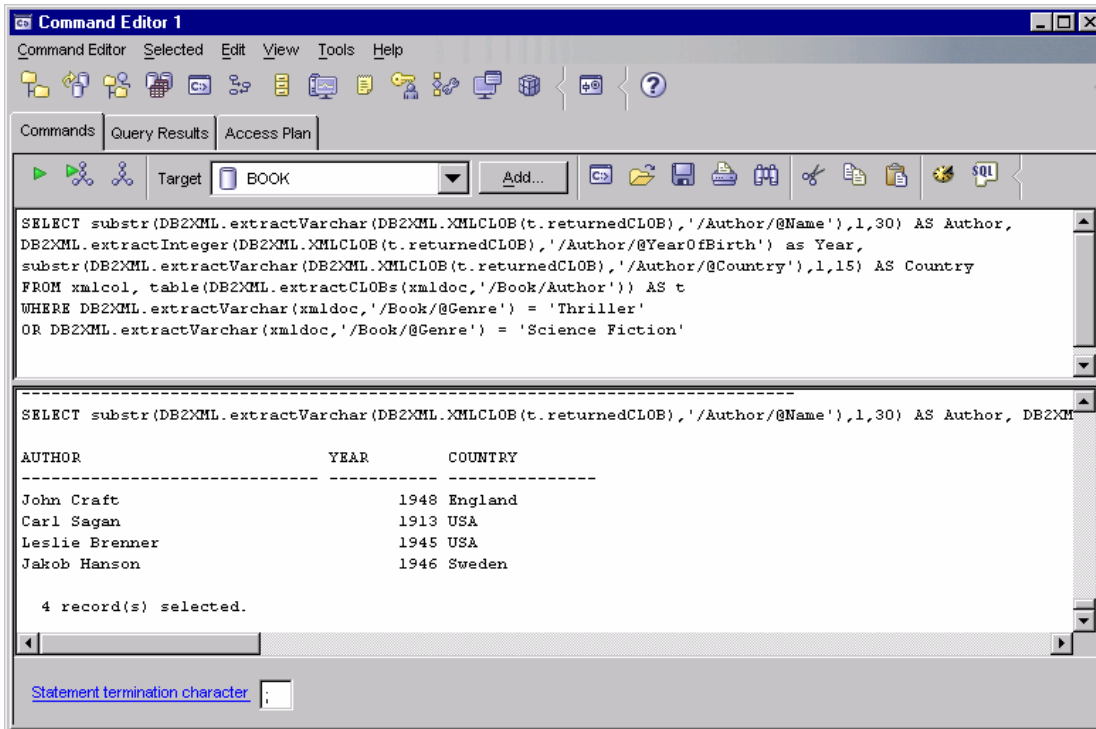


The way to solve the previous question with the `extractCLOBs` function instead of the three table functions (that did not work) would be the following:

First we extract CLOBs for all the **Author** elements of books that match the condition criteria and then we can use the simple extract functions to retrieve the wanted data from the new XML documents (the CLOBs):

```
SELECT  
substr(DB2XML.extractVarchar(DB2XML.XMLCLOB(t.returnedCLOB),'/Author/@Name'),1,30) AS  
Author,  
DB2XML.extractInteger(DB2XML.XMLCLOB(t.returnedCLOB),'/Author/@YearOfBirth') as Year,  
substr(DB2XML.extractVarchar(DB2XML.XMLCLOB(t.returnedCLOB),'/Author/@Country'),1,15)  
AS Country  
FROM xmlcol, table(DB2XML.extractCLOBs(xmldoc,'/Book/Author')) AS t  
WHERE DB2XML.extractVarchar(xmldoc,'/Book/@Genre') = 'Thriller'  
OR DB2XML.extractVarchar(xmldoc,'/Book/@Genre') = 'Science Fiction'
```

The function `XMLCLOB` of the schema `DB2XML` is also used here. This is a casting function that takes a `CLOB` value and returns it as an `XMLCLOB` value. This is required because the extract functions expect a variable of XML data type (such as `XMLCLOB` or `XMLVARCHAR`). This SQL statement returns all the information on authors that have written thrillers or science fiction:



Sometimes it may be necessary to combine in the result, data from different levels of the XML structure. The following questing asks as to do exactly that:

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

To solve this we will need to have conditions on two levels and also retrieve information from two levels. When solving a problem like this, we always start at the higher level of the XML structure (the Book element) and move step by step through the sub-elements. The first thing to do is to check the genre of the books and retrieve the title and the authors (as CLOBs). When we have done that we can start working with the contents of the author CLOBs. The first part can be done with the following SELECT statement:

```
SELECT substr(DB2XML.extractVarchar(xmldoc, '/Book/@Title'),1,30) AS Title,  
DB2XML.XMLCLOB(t.returnedCLOB) AS AuthorXML  
FROM xmlcol, table(DB2XML.extractCLOBs(xmldoc, '/Book/Author')) AS t  
WHERE DB2XML.extractVarchar(xmldoc, '/Book/@Genre') = 'Educational'
```

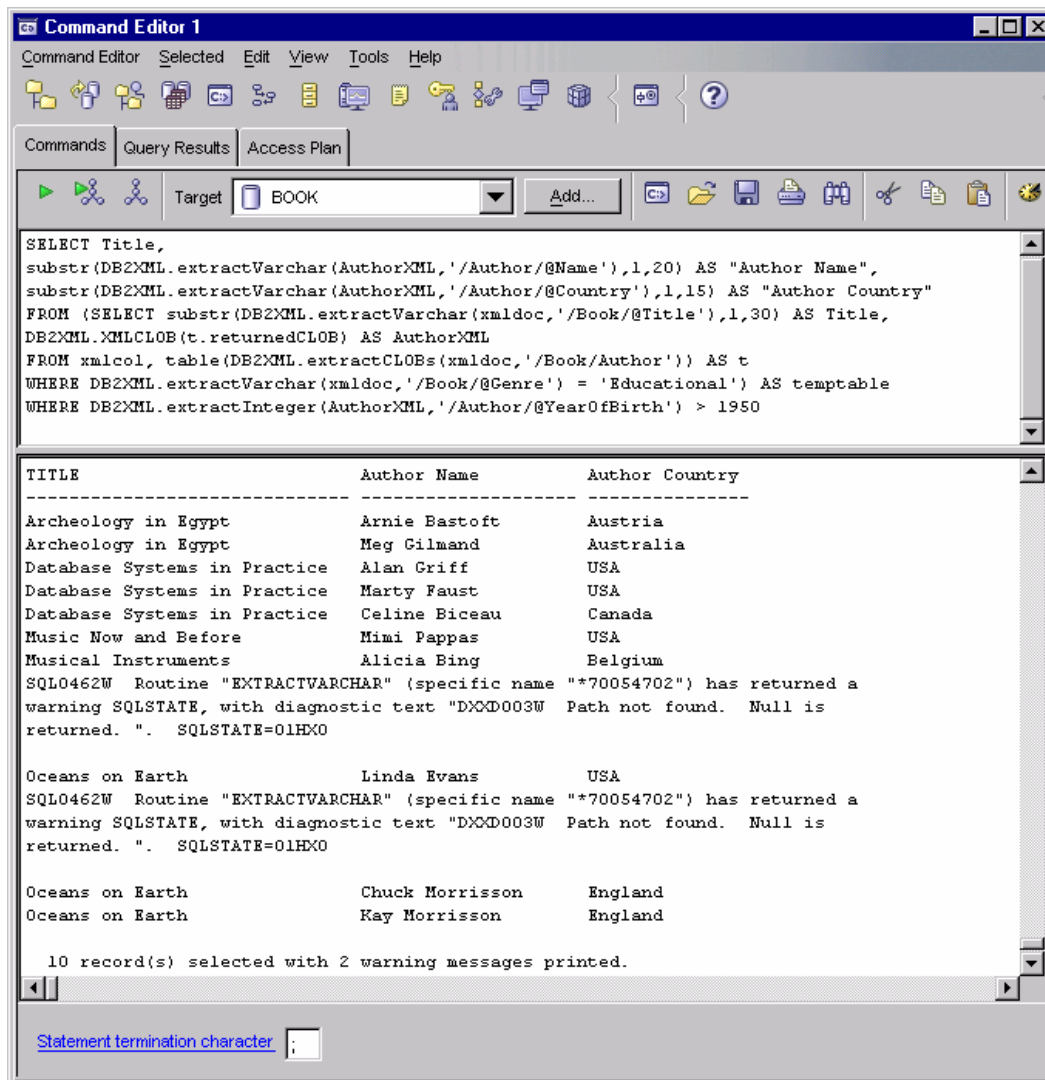
This will create a table with two columns (the book title and the author CLOB) and one row for each author of each educational book. We also cast the returned CLOB into an `XMLCLOB`, so that we don't have to do it later.

We can now use this SELECT statement as the source for an outer SELECT statement. This means that we assign a name to the result of this SELECT statement, which will be considered by the new SELECT statement as a table with two columns (Title and AuthorXML).

The new SELECT statement will then retrieve the name and country of the authors and control the year of birth:

```
SELECT Title,  
substr(DB2XML.extractVarchar(AuthorXML,'/Author/@Name'),1,20) AS "Author Name",  
substr(DB2XML.extractVarchar(AuthorXML,'/Author/@Country'),1,15) AS "Author Country"  
FROM (SELECT substr(DB2XML.extractVarchar(xmlcol,'/Book/@Title'),1,30) AS Title,  
DB2XML.XMLCLOB(t.returnedCLOB) AS AuthorXML  
FROM xmlcol, table(DB2XML.extractCLOBs(xmlcol,'/Book/Author')) AS t  
WHERE DB2XML.extractVarchar(xmlcol,'/Book/@Genre') = 'Educational') AS temptable  
WHERE DB2XML.extractInteger(AuthorXML,'/Author/@YearOfBirth') > 1950
```

And the result is the following:



Command Editor 1

Command Editor Selected Edit View Tools Help

Commands Query Results Access Plan

Target BOOK Add...

```
SELECT Title,  
substr(DB2XML.extractVarchar(AuthorXML,'/Author/@Name'),1,20) AS "Author Name",  
substr(DB2XML.extractVarchar(AuthorXML,'/Author/@Country'),1,15) AS "Author Country"  
FROM (SELECT substr(DB2XML.extractVarchar(xmlcol,'/Book/@Title'),1,30) AS Title,  
DB2XML.XMLCLOB(t.returnedCLOB) AS AuthorXML  
FROM xmlcol, table(DB2XML.extractCLOBs(xmlcol,'/Book/Author')) AS t  
WHERE DB2XML.extractVarchar(xmlcol,'/Book/@Genre') = 'Educational') AS temptable  
WHERE DB2XML.extractInteger(AuthorXML,'/Author/@YearOfBirth') > 1950
```

TITLE	Author Name	Author Country
Archeology in Egypt	Arnie Bastoft	Austria
Archeology in Egypt	Meg Gilmand	Australia
Database Systems in Practice	Alan Griff	USA
Database Systems in Practice	Marty Faust	USA
Database Systems in Practice	Celine Biceau	Canada
Music Now and Before	Mimi Pappas	USA
Musical Instruments	Alicia Bing	Belgium
SQL0462W Routine "EXTRACTVARCHAR" (specific name "*70054702") has returned a warning SQLSTATE, with diagnostic text "DXXD003W Path not found. Null is returned. ". SQLSTATE=01HX0		
Oceans on Earth	Linda Evans	USA
SQL0462W Routine "EXTRACTVARCHAR" (specific name "*70054702") has returned a warning SQLSTATE, with diagnostic text "DXXD003W Path not found. Null is returned. ". SQLSTATE=01HX0		
Oceans on Earth	Chuck Morrisson	England
Oceans on Earth	Kay Morrisson	England

10 record(s) selected with 2 warning messages printed.

Statement termination character :

Now we are ready to look at really complex examples. The following qualifies as such:

Show a list of all the authors born after 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 following columns: Author Name, Author Country, Amount of editions, Amount of translation languages, Average price. The result shall be sorted by author name!

To solve this we will need to work in many steps. First we need to extract the editions and the authors:

```
SELECT DB2XML.XMLCLOB(t1.returnedCLOB) AS AuthorXML,  
DB2XML.XMLCLOB(t2.returnedCLOB) AS EditionXML  
FROM xmlcol, table(DB2XML.extractCLOBs(xmldoc,'/Book/Author')) AS t1,  
table(DB2XML.extractCLOBs(xmldoc,'/Book/Edition')) AS t2
```

This will return all valid combinations of authors and editions (55 such).

Next thing we have to do is to extract the name and country of the authors and also get rid of the authors that were born 1940 or earlier. At the same time we can also extract the edition price:

```
SELECT substr(DB2XML.extractVarchar(AuthorXML,'/Author/@Name'),1,20) AS Name,  
substr(DB2XML.extractVarchar(AuthorXML,'/Author/@Country'),1,20) AS Country,  
DB2XML.extractInteger(EditionXML,'/Edition/@Price') AS Price  
FROM (SELECT DB2XML.XMLCLOB(t1.returnedCLOB) AS AuthorXML,  
DB2XML.XMLCLOB(t2.returnedCLOB) AS EditionXML  
FROM xmlcol, table(DB2XML.extractCLOBs(xmldoc,'/Book/Author')) AS t1,  
table(DB2XML.extractCLOBs(xmldoc,'/Book/Edition')) AS t2) AS temptable1  
WHERE DB2XML.extractInteger(AuthorXML,'/Author/@YearOfBirth') > 1940
```

We can now use this SQL statement in the FROM clause of the next SELECT statement. Now we have enough information to count the amount of editions and even calculate the average edition price:

```
SELECT Name, Country, COUNT(*) AS "Amount of editions", AVG(Price) "Average edition price"  
FROM (SELECT substr(DB2XML.extractVarchar(AuthorXML,'/Author/@Name'),1,20) AS Name,  
substr(DB2XML.extractVarchar(AuthorXML,'/Author/@Country'),1,20) AS Country,  
DB2XML.extractInteger(EditionXML,'/Edition/@Price') AS Price  
FROM (SELECT DB2XML.XMLCLOB(t1.returnedCLOB) AS AuthorXML,  
DB2XML.XMLCLOB(t2.returnedCLOB) AS EditionXML  
FROM xmlcol, table(DB2XML.extractCLOBs(xmldoc,'/Book/Author')) AS t1,  
table(DB2XML.extractCLOBs(xmldoc,'/Book/Edition')) AS t2) AS temptable1  
WHERE DB2XML.extractInteger(AuthorXML,'/Author/@YearOfBirth') > 1940) AS temptable2  
GROUP BY Name, Country
```

This statement has now four columns. These are four of the five that we need to have in the final result.

Command Editor 1

Command Editor Selected Edit View Tools Help

Commands Query Results Access Plan

Target: BOOK

```

SELECT Name, Country, COUNT(*) AS "Amount of editions", AVG(Price) "Average edition price"
FROM (SELECT substr(DB2XML.extractVarchar(AuthorXML,'/Author/@Name'),1,20) AS Name,
      substr(DB2XML.extractVarchar(AuthorXML,'/Author/@Country'),1,20) AS Country,
      DB2XML.extractInteger(EditionXML,'/Edition/@Price') AS Price
FROM (SELECT DB2XML.XMLCLOB(t1.returnedCLOB) AS AuthorXML,
      DB2XML.XMLCLOB(t2.returnedCLOB) AS EditionXML
FROM xmlcol, table(DB2XML.extractCLOBs(xmlcol,'/Book/Author')) AS t1,
      table(DB2XML.extractCLOBs(xmlcol,'/Book/Edition')) AS t2) AS temptable1
WHERE DB2XML.extractInteger(AuthorXML,'/Author/@YearOfBirth') > 1940) AS temptable2
GROUP BY Name, Country
  
```

Author Name	Country	Amount of editions	Average edition price
Antje Biederman	Germany	1	830
Arnie Bastoft	Austria	3	270
Auna Gonzales Perre	Portugal	1	650
Celine Biceau	Canada	2	435
Chris Ryan	France	3	270
Chuck Morrisson	England	4	445
Jakob Hanson	Sweden	1	120
James Patterson	USA	1	180
John Craft	England	1	120
Kay Morrisson	England	4	445
Leslie Brenner	USA	1	210
Linda Evans	USA	4	445
Marty Faust	USA	2	435
Meg Gilmand	Australia	3	270
Mimi Pappas	USA	2	317
Peter de Jonge	USA	1	180
Pierre Zargone	Belgium	2	135

19 record(s) selected.

Statement termination character: ;

The last (missing) column is the amount of different languages every author has been translated into. To get that, we have to start from the beginning again. This is especially important since the translation element may not appear for all editions. This means that we will have half of the results in one SQL statement and the other half in another. We will simply need to join the two results at the end. And in order to guaranty that all the authors are in the result we will have to use an outer join. Remember that the result above had 19 rows. We should have 19 rows in the final result as well. But first things first.

In order to retrieve the different languages we start from the xmlcol as we did before, but this time we can extract directly the author names and the translation languages (all the valid combinations). Since we are going to join the result of this part with the result from before, we need not care about the conditions (The invalid authors will automatically get filtered out when we join with the list of the valid ones that we created before.):

```

SELECT substr(t1.returnedVarchar,1,20) AS Name,
       substr(t2.returnedVarchar,1,20) AS Language
FROM xmlcol,
      table(DB2XML.extractVarchars(xmlcol,'/Book/Author/@Name')) AS t1,
      table(DB2XML.extractVarchars(xmlcol,'/Book/Edition/Translation/@Language')) AS t2
  
```

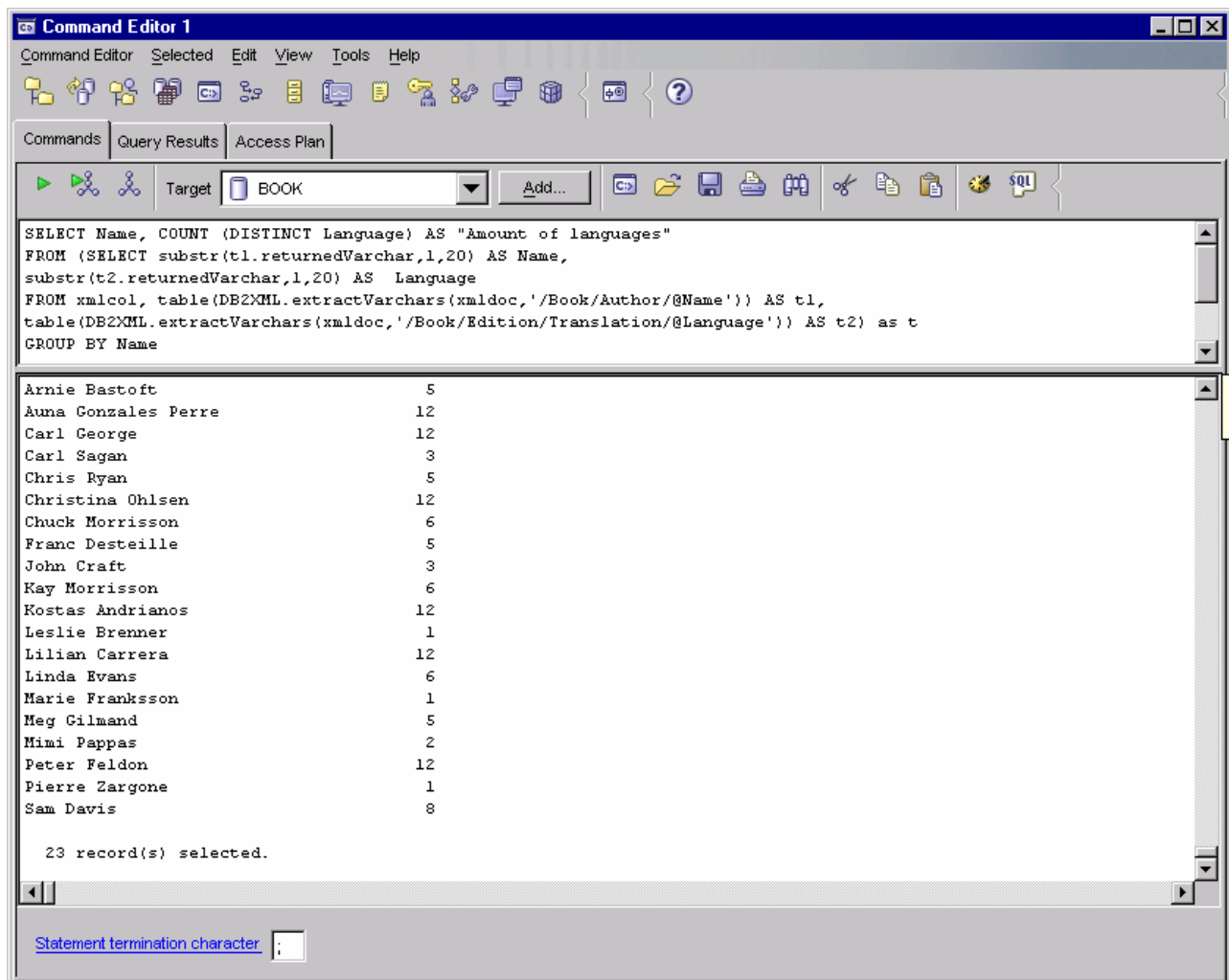
Notice that here it is okay to use two table functions together because we do want all the combinations of languages and author names!

Now we can use this result to count the different languages every author has been translated into:

```
SELECT Name, COUNT (DISTINCT Language) AS "Amount of languages"
FROM (SELECT substr(t1.returnedVarchar,1,20) AS Name,
      substr(t2.returnedVarchar,1,20) AS Language
      FROM xmlcol, table(DB2XML.extractVarchars(xmldoc,'/Book/Author/@Name')) AS t1,
      table(DB2XML.extractVarchars(xmldoc,'/Book/Edition/Translation/@Language')) AS t2) as t
GROUP BY Name
```

Here we used the **DISTINCT** keyword in the **COUNT** function in order to count each language once. We could just as easily have used **DISTINCT** in the **SELECT** clause of the nested statement.

Either way, this returns two columns: the author name and the amount of different languages:



The screenshot shows a 'Command Editor 1' window with a menu bar (Command Editor, Selected, Edit, View, Tools, Help) and a toolbar. Below the toolbar are tabs for 'Commands', 'Query Results', and 'Access Plan'. The 'Query Results' tab is active, displaying the results of the SQL query entered in the 'Commands' tab. The query is a SELECT statement that counts the number of distinct languages for each author. The results are shown in a table with two columns: 'Name' and 'Amount of languages'. There are 23 records selected. At the bottom, there is a 'Statement termination character' field with a semicolon (;) entered.

Name	Amount of languages
Arnie Bastoft	5
Auna Gonzales Perre	12
Carl George	12
Carl Sagan	3
Chris Ryan	5
Christina Ohlsen	12
Chuck Morrisson	6
Franc Desteille	5
John Craft	3
Kay Morrisson	6
Kostas Andrianos	12
Leslie Brenner	1
Lillian Carrera	12
Linda Evans	6
Marie Franksson	1
Meg Gilmand	5
Mimi Pappas	2
Peter Feldon	12
Pierre Zargone	1
Sam Davis	8

23 record(s) selected.

Statement termination character: ;

The author name is the column that this result and the previous result have in common, and it is the one we need in order to join the two results. Note that this result has 23 rows (more than 19). But this does not automatically mean that all the 19 that we want are included in the 23.

Now to join the two parts. We can simply construct a new SELECT statement and place the two parts as two tables in the FROM clause. Then we simply use as a join condition, checking that the author names are equal. We start with an inner join in order to prove that this could fail to catch all the 19 authors:

```
SELECT part1.Name AS "Author name", Country AS "Author Country",  
"Amount of editions", "Average edition price", "Amount of languages"  
FROM  
(SELECT Name, Country, COUNT(*) AS "Amount of editions",  
AVG(Price) "Average edition price"  
FROM (SELECT substr(DB2XML.extractVarchar(AuthorXML,'/Author/@Name'),1,20) AS Name,  
substr(DB2XML.extractVarchar(AuthorXML,'/Author/@Country'),1,20) AS Country,  
DB2XML.extractInteger(EditionXML,'/Edition/@Price') AS Price  
FROM (SELECT DB2XML.XMLCLOB(t1.returnedCLOB) AS AuthorXML,  
DB2XML.XMLCLOB(t2.returnedCLOB) AS EditionXML  
FROM xmlcol, table(DB2XML.extractCLOBs(xmlcol,'/Book/Author')) AS t1,  
table(DB2XML.extractCLOBs(xmlcol,'/Book/Edition')) AS t2) AS temptable1  
WHERE DB2XML.extractInteger(AuthorXML,'/Author/@YearOfBirth') > 1940) AS temptable2  
GROUP BY Name, Country) AS part1,  
(SELECT Name, COUNT (DISTINCT Language) AS "Amount of languages"  
FROM (SELECT substr(t1.returnedVarchar,1,20) AS Name,  
substr(t2.returnedVarchar,1,20) AS Language  
FROM xmlcol, table(DB2XML.extractVarchars(xmlcol,'/Book/Author/@Name')) AS t1,  
table(DB2XML.extractVarchars(xmlcol,'/Book/Edition/Translation/@Language')) AS t2) as t  
GROUP BY Name) AS part2  
WHERE part1.Name = part2.Name  
ORDER BY 1
```

And the result will look nicely like this:

Command Editor 1

Command Editor Selected Edit View Tools Help

Commands Query Results Access Plan

Target BOOK Add...

```
SELECT part1.Name AS "Author name", Country AS "Author Country",
"Amount of editions", "Average edition price", "Amount of languages"
FROM
(SELECT Name, Country,
COUNT(*) AS "Amount of editions",
AVG(Price) "Average edition price"
FROM (SELECT substr(DB2XML.extractVarchar(AuthorXML,'/Author/@Name'),1,20) AS Name,
```

```
SELECT part1.Name AS "Author name", Country AS "Author Country", "Amount of editions", "Average edition price", "Amount of languages"
FROM (SELECT substr(DB2XML.extractVarchar(AuthorXML,'/Author/@Name'),1,20) AS Name,
substr(DB2XML.extractVarchar(AuthorXML,'/Author/@Country'),1,20) AS Country,
DB2XML.extractInteger(EditionXML,'/Edition/@Price') AS Price
FROM (SELECT DB2XML.XMLCLOB(t1.returnedCLOB) AS AuthorXML,
DB2XML.XMLCLOB(t2.returnedCLOB) AS EditionXML
FROM xmlcol, table(DB2XML.extractCLOBs(xmldoc,'/Book/Author')) AS t1,
table(DB2XML.extractCLOBs(xmldoc,'/Book/Edition')) AS t2) AS temptable1
WHERE DB2XML.extractInteger(AuthorXML,'/Author/@YearOfBirth') > 1940) AS temptable2
GROUP BY Name, Country) AS part1 LEFT OUTER JOIN
(SELECT Name, COUNT (DISTINCT Language) AS "Amount of languages"
FROM (SELECT substr(t1.returnedVarchar,1,20) AS Name,
substr(t2.returnedVarchar,1,20) AS Language
FROM xmlcol, table(DB2XML.extractVarchars(xmldoc,'/Book/Author/@Name')) AS t1,
table(DB2XML.extractVarchars(xmldoc,'/Book/Edition/Translation/@Language')) AS t2) AS t2
GROUP BY Name) AS part2 ON (part1.Name = part2.Name)
ORDER BY 1
```

Author name	Author Country	Amount of editions	Average edition price	Amount of languages
Alicia Bing	Belgium	2	400	7
Antje Liedderman	Germany	1	650	12
Arnie Bastoft	Austria	3	270	5
Auna Gonzales Perre	Portugal	1	650	12
Chris Ryan	France	3	270	5
Chuck Morrisson	England	4	445	6
John Craft	England	1	120	3
Kay Morrisson	England	4	445	6
Leslie Brenner	USA	1	210	1
Linda Evans	USA	4	445	6
Meg Gilmand	Australia	3	270	5
Mimi Pappas	USA	2	317	2
Pierre Zargone	Belgium	2	135	1

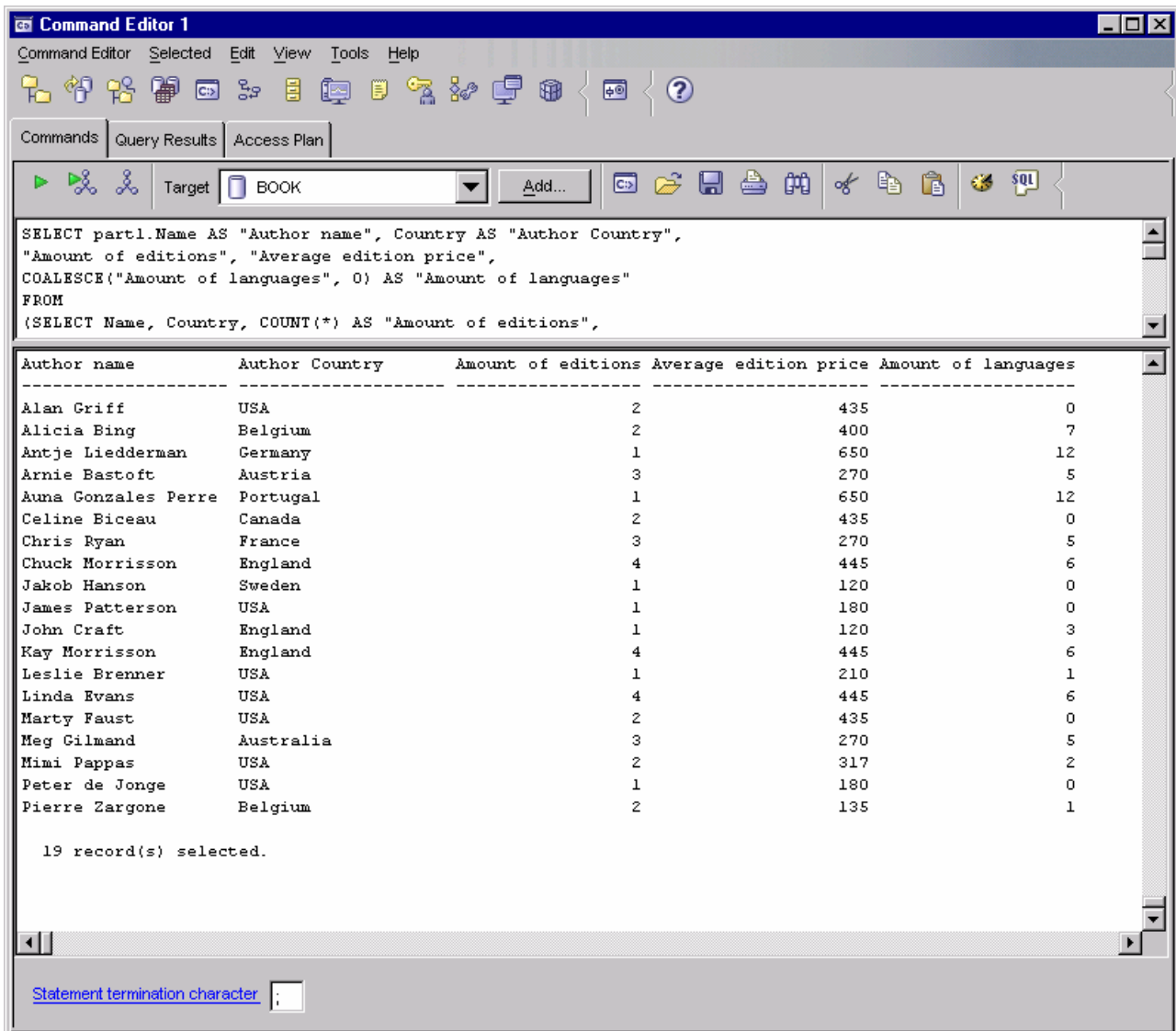
13 record(s) selected.

Statement termination character :

But it only contains 13 rows. That means that 6 of the wanted authors did not have any translations. So we will simply add the value 0 for their fifth column. We can achieve this with a **LEFT OUTER JOIN** and the **COALESCE** function:

```
SELECT part1.Name AS "Author name", Country AS "Author Country",
"Amount of editions", "Average edition price",
COALESCE("Amount of languages", 0) AS "Amount of languages"
FROM
(SELECT Name, Country, COUNT(*) AS "Amount of editions",
AVG(Price) "Average edition price"
FROM (SELECT substr(DB2XML.extractVarchar(AuthorXML,'/Author/@Name'),1,20) AS Name,
substr(DB2XML.extractVarchar(AuthorXML,'/Author/@Country'),1,20) AS Country,
DB2XML.extractInteger(EditionXML,'/Edition/@Price') AS Price
FROM (SELECT DB2XML.XMLCLOB(t1.returnedCLOB) AS AuthorXML,
DB2XML.XMLCLOB(t2.returnedCLOB) AS EditionXML
FROM xmlcol, table(DB2XML.extractCLOBs(xmldoc,'/Book/Author')) AS t1,
table(DB2XML.extractCLOBs(xmldoc,'/Book/Edition')) AS t2) AS temptable1
WHERE DB2XML.extractInteger(AuthorXML,'/Author/@YearOfBirth') > 1940) AS temptable2
GROUP BY Name, Country) AS part1 LEFT OUTER JOIN
(SELECT Name, COUNT (DISTINCT Language) AS "Amount of languages"
FROM (SELECT substr(t1.returnedVarchar,1,20) AS Name,
substr(t2.returnedVarchar,1,20) AS Language
FROM xmlcol, table(DB2XML.extractVarchars(xmldoc,'/Book/Author/@Name')) AS t1,
table(DB2XML.extractVarchars(xmldoc,'/Book/Edition/Translation/@Language')) AS t2) AS t2
GROUP BY Name) AS part2 ON (part1.Name = part2.Name)
ORDER BY 1
```

And we finally have the correct result:



The screenshot shows the 'Command Editor 1' window with the following SQL query:

```
SELECT part1.Name AS "Author name", Country AS "Author Country",  
"Amount of editions", "Average edition price",  
COALESCE("Amount of languages", 0) AS "Amount of languages"  
FROM  
{SELECT Name, Country, COUNT(*) AS "Amount of editions",
```

The results are displayed in a table with the following columns: Author name, Author Country, Amount of editions, Average edition price, and Amount of languages. There are 19 records selected.

Author name	Author Country	Amount of editions	Average edition price	Amount of languages
Alan Griff	USA	2	435	0
Alicia Bing	Belgium	2	400	7
Antje Liedderman	Germany	1	650	12
Arnie Bastoft	Austria	3	270	5
Auna Gonzales Perre	Portugal	1	650	12
Celine Biceau	Canada	2	435	0
Chris Ryan	France	3	270	5
Chuck Morrisson	England	4	445	6
Jakob Hanson	Sweden	1	120	0
James Patterson	USA	1	180	0
John Craft	England	1	120	3
Kay Morrisson	England	4	445	6
Leslie Brenner	USA	1	210	1
Linda Evans	USA	4	445	6
Marty Faust	USA	2	435	0
Meg Gilmand	Australia	3	270	5
Mimi Pappas	USA	2	317	2
Peter de Jonge	USA	1	180	0
Pierre Zargone	Belgium	2	135	1

19 record(s) selected.

For even more explained examples you can take a look an older version of the lab compendium pages 29-36 (model of XML structure on page 10). This can be found at

\\Db-srv-1\\StudentCourseMaterial\\IS4 spring 2006\\DB2-XML\\Comdendium DB2-XML v.2.0
(ht2001).doc

4.2.2 Manipulating data

Retrieving data from the XML documents is not always enough. Sometimes we need to change a value in an XML document, without having to delete the entire document and insert it after manually making a change. We may also want to do some methodic change in the entire XML column, such as change the word "USSR" to "Russia" for any attribute named Country.

In this section we will look at a couple of examples of doing such changes. We will start with the following problem:

Change the e-mail of Jakob Hanson to hanson@home.se!

We can do this in two different ways. The first way is to go through every XML document in the XML column and update the path `/Book/Author[@Name="Jakob Hanson"]/@Email` to `hanson@home.se`. This would of course do a lot of extra work, but in a smaller system it may not matter. The other way would be to first find the XML documents that contain an author name Jakob Hanson and then change the email in those documents only. Both ways will produce the same result.

Here is an UPDATE statement for the first variant:

```
UPDATE xmlcol
SET xmldoc = DB2XML.update(xmldoc,
    '/Book/Author[@Name="Jakob Hanson"]/@Email',
    'hanson@home.se')
```

After running this we get a message that the command was completed successfully, but we may also want to verify that the e-mail address really got updated. We can simply do that with the following SQL statement:

```
SELECT substr(DB2XML.extractVarchar(xclob,'/Author/@Email'),1,20) AS Email
FROM (SELECT DB2XML.XMLCLOB(returnedCLOB) AS xclob
      FROM xmlcol,
      table(DB2XML.extractCLOBs(xmldoc,'/Book/Author')) AS t) AS temp
WHERE DB2XML.extractVarchar(xclob,'/Author/@Name') = 'Jakob Hanson'
```

The other version of the UPDATE statement would look like this:

```
UPDATE xmlcol
SET xmldoc = DB2XML.update(xmldoc,
    '/Book/Author[@Name="Jakob Hanson"]/@Email',
    'hanson@home.se')
WHERE 'Jakob Hanson' IN
    (SELECT returnedvarchar
     FROM table(DB2XML.extractVarchars(xmldoc, '/Book/Author/@Name')) as t)
```

This version is much faster, but it may be difficult to detect when the slow version only takes a second or two. For the exercises in section 4.4 you can use any of the two styles.

4.3 Queries that produce XML explained

In this section we take a look at the functions used to create XML as the output of an SQL statement. The examples in this section are based on the horse riding database described in section 3.2.

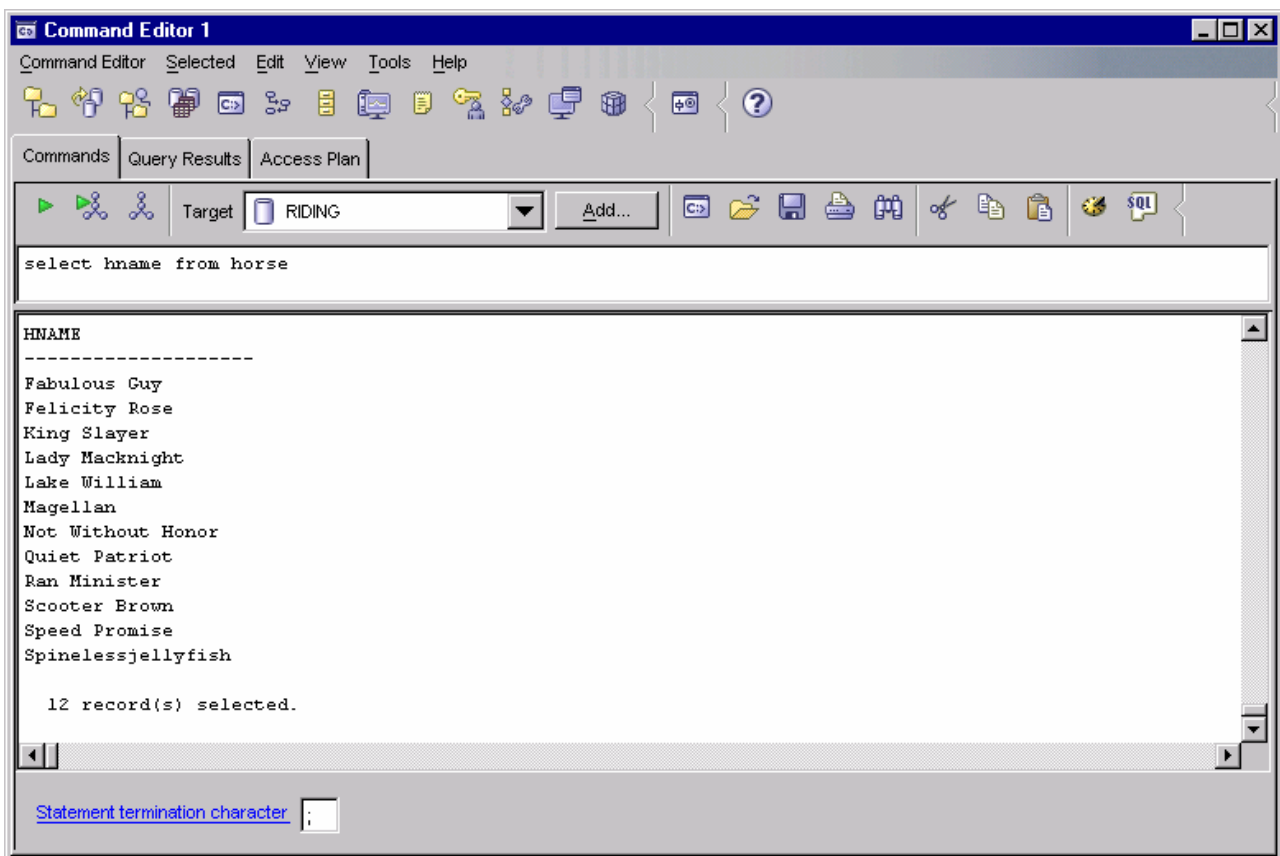
We can start with a simple example:

*Return all the horse names as the content of **Horse** elements!*

First we could just write a standard SQL statement to return the horse names:

```
SELECT Hname FROM Horse
```

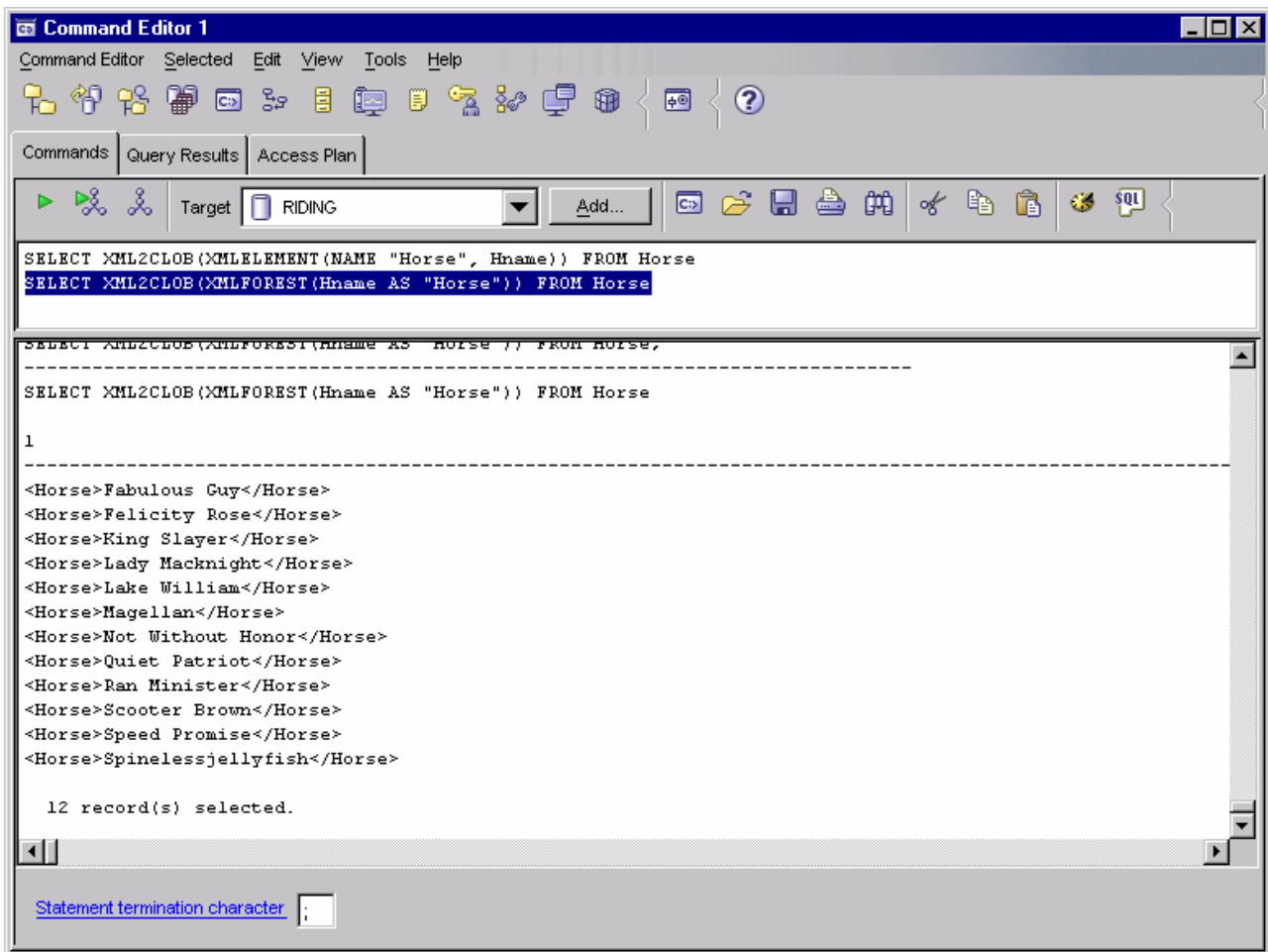
This would simply return all the names:



What we want to do is place these names inside XML elements. To do this we can use the `XMLELEMENT` function or the `XMLFOREST` function. DB2 requires that we transform the result to a data type that can be shown for the user. Since all the XML functions return a result of the special XML data type, we can use the function `XML2CLOB` to cast them to `CLOB`.

```
SELECT XML2CLOB(XMLELEMENT(NAME "Horse", Hname)) FROM Horse  
SELECT XML2CLOB(XMLFOREST(Hname AS "Horse")) FROM Horse
```

Either statement will return the same result:

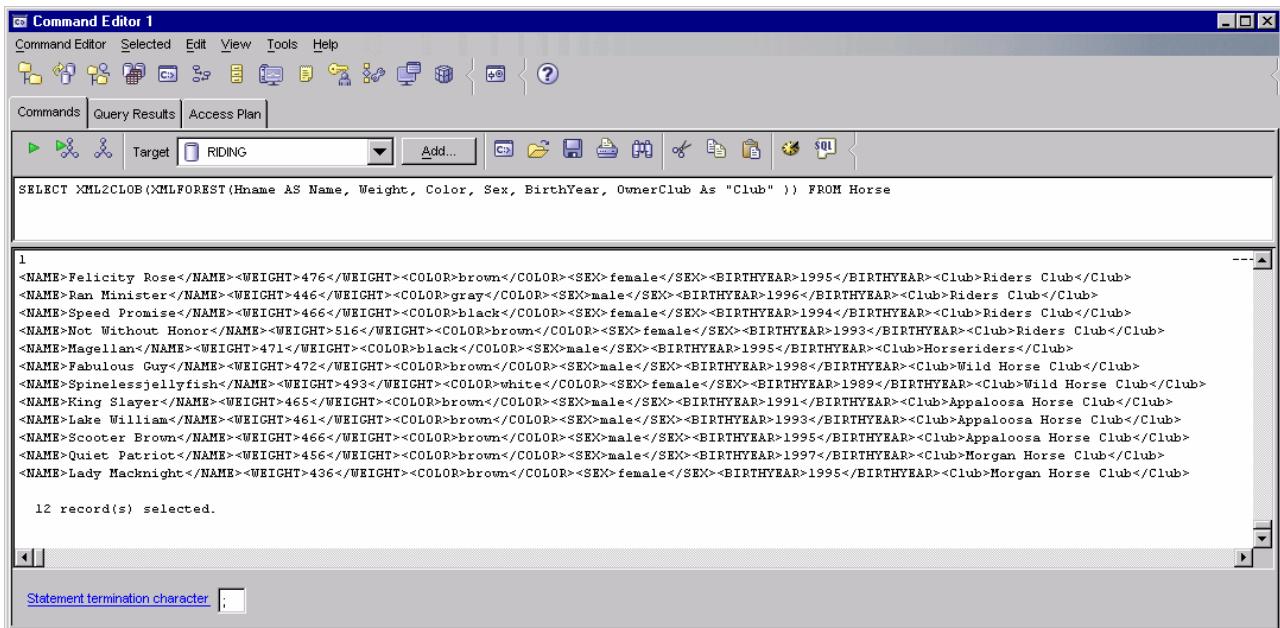


The XMLFOREST function can be used to create more than one element, while the XMLELEMENT only produces one element. For the following we can use the XMLFORSET function:

Show all the information of each horse as elements!

```
SELECT XML2CLOB(XMLFOREST(Hname AS Name, Weight, Color, Sex, BirthYear,  
OwnerClub As "Club" ))  
FROM Horse
```

Note that the element names use capital letters unless we specify the element name enclosed in double quotes. Also note that the result has one column.



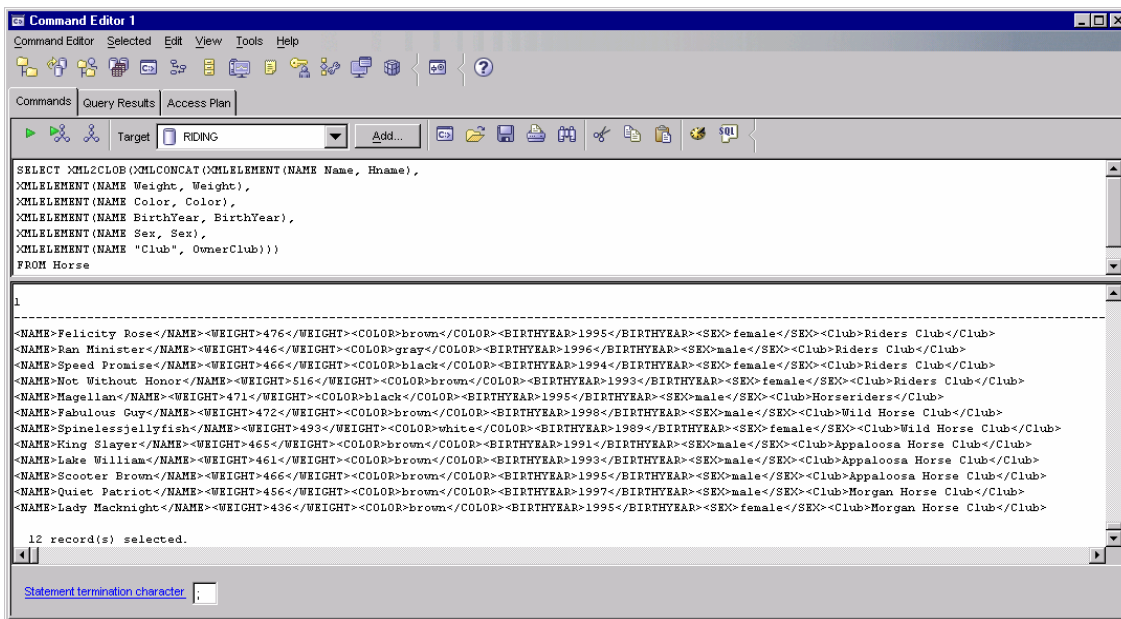
If we try to do a similar thing with the XMLELEMENT function we could have this statement:

```
SELECT XML2CLOB(XMLELEMENT(NAME Name, Hname)),  
XML2CLOB(XMLELEMENT(NAME Weight, Weight)),  
XML2CLOB(XMLELEMENT(NAME Color, Color)),  
XML2CLOB(XMLELEMENT(NAME BirthYear, BirthYear)),  
XML2CLOB(XMLELEMENT(NAME Sex, Sex)),  
XML2CLOB(XMLELEMENT(NAME "Club", OwnerClub))  
FROM Horse
```

This would return six columns and each column would be 4000 characters wide. In order to make the six elements one uninterrupted sequence, we can use the XMLCONCAT function:

```
SELECT XML2CLOB(XMLCONCAT(XMLELEMENT(NAME Name, Hname),  
XMLELEMENT(NAME Weight, Weight),  
XMLELEMENT(NAME Color, Color),  
XMLELEMENT(NAME BirthYear, BirthYear),  
XMLELEMENT(NAME Sex, Sex),  
XMLELEMENT(NAME "Club", OwnerClub)))  
FROM Horse
```

This will produce the exact same result as the version using the XMLFOREST function:



```
SELECT XML2CLOB(XMLCONCAT(XMLELEMENT(NAME Name, Hname),
XMLELEMENT(NAME Weight, Weight),
XMLELEMENT(NAME Color, Color),
XMLELEMENT(NAME BirthYear, BirthYear),
XMLELEMENT(NAME Sex, Sex),
XMLELEMENT(NAME "Club", OwnerClub)))
FROM Horse
```

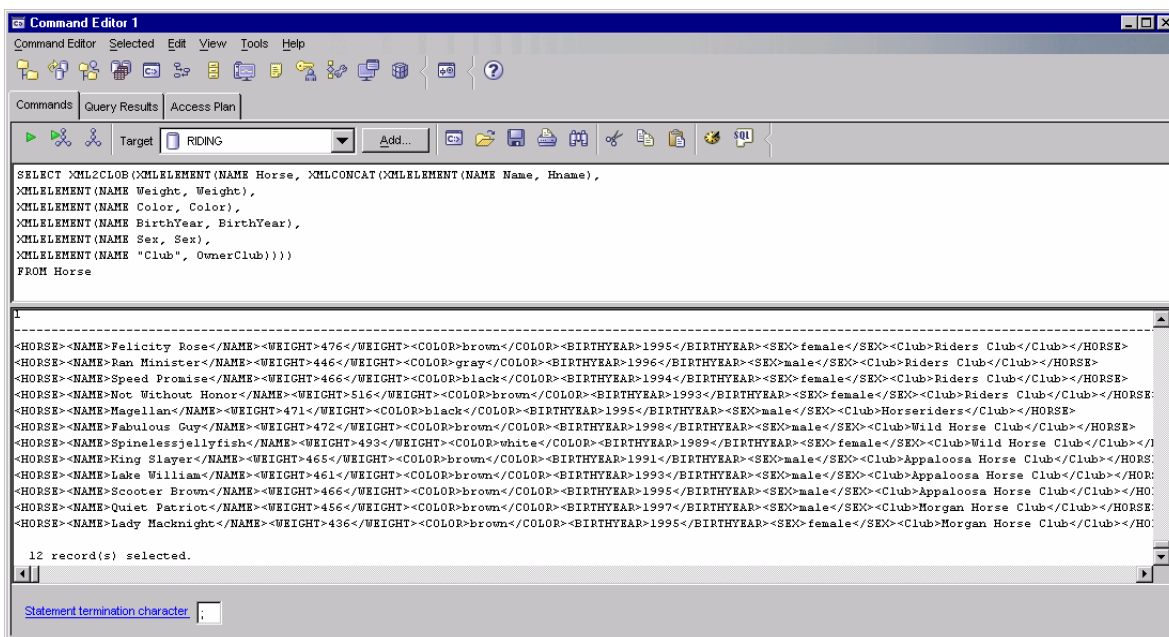
```
1
-----
<NAME>Felicity Rose</NAME><WEIGHT>476</WEIGHT><COLOR>brown</COLOR><BIRTHYEAR>1995</BIRTHYEAR><SEX>female</SEX><Club>Riders Club</Club>
<NAME>Pan Minister</NAME><WEIGHT>446</WEIGHT><COLOR>gray</COLOR><BIRTHYEAR>1996</BIRTHYEAR><SEX>male</SEX><Club>Riders Club</Club>
<NAME>Speed Promise</NAME><WEIGHT>466</WEIGHT><COLOR>black</COLOR><BIRTHYEAR>1994</BIRTHYEAR><SEX>female</SEX><Club>Riders Club</Club>
<NAME>Not Without Honor</NAME><WEIGHT>516</WEIGHT><COLOR>brown</COLOR><BIRTHYEAR>1993</BIRTHYEAR><SEX>female</SEX><Club>Riders Club</Club>
<NAME>Magellan</NAME><WEIGHT>471</WEIGHT><COLOR>black</COLOR><BIRTHYEAR>1995</BIRTHYEAR><SEX>male</SEX><Club>Horseriders</Club>
<NAME>Fabulous Guy</NAME><WEIGHT>472</WEIGHT><COLOR>brown</COLOR><BIRTHYEAR>1998</BIRTHYEAR><SEX>male</SEX><Club>Wild Horse Club</Club>
<NAME>Spinelessjellyfish</NAME><WEIGHT>493</WEIGHT><COLOR>white</COLOR><BIRTHYEAR>1989</BIRTHYEAR><SEX>female</SEX><Club>Wild Horse Club</Club>
<NAME>King Slayer</NAME><WEIGHT>465</WEIGHT><COLOR>brown</COLOR><BIRTHYEAR>1991</BIRTHYEAR><SEX>male</SEX><Club>Appaloosa Horse Club</Club>
<NAME>Lake William</NAME><WEIGHT>461</WEIGHT><COLOR>brown</COLOR><BIRTHYEAR>1993</BIRTHYEAR><SEX>male</SEX><Club>Appaloosa Horse Club</Club>
<NAME>Scooter Brown</NAME><WEIGHT>466</WEIGHT><COLOR>brown</COLOR><BIRTHYEAR>1995</BIRTHYEAR><SEX>male</SEX><Club>Appaloosa Horse Club</Club>
<NAME>Quiet Patriot</NAME><WEIGHT>456</WEIGHT><COLOR>brown</COLOR><BIRTHYEAR>1997</BIRTHYEAR><SEX>male</SEX><Club>Morgan Horse Club</Club>
<NAME>Lady MacKnight</NAME><WEIGHT>436</WEIGHT><COLOR>brown</COLOR><BIRTHYEAR>1995</BIRTHYEAR><SEX>female</SEX><Club>Morgan Horse Club</Club>

12 record(s) selected.
```

But this may not be exactly what we wanted after all (Because the result is not a well-formed XML document.). We probably would like all these elements to be inside a Horse element. To do this we can create an element and place the XMLCONCAT or the XMLFOREST inside it:

```
SELECT XML2CLOB(XMLELEMENT(NAME Horse, XMLCONCAT(XMLELEMENT(NAME
Name, Hname), XMLELEMENT(NAME Weight, Weight),
XMLELEMENT(NAME Color, Color), XMLELEMENT(NAME BirthYear, BirthYear),
XMLELEMENT(NAME Sex, Sex), XMLELEMENT(NAME "Club", OwnerClub))))
FROM Horse
```

And we have a much better result:



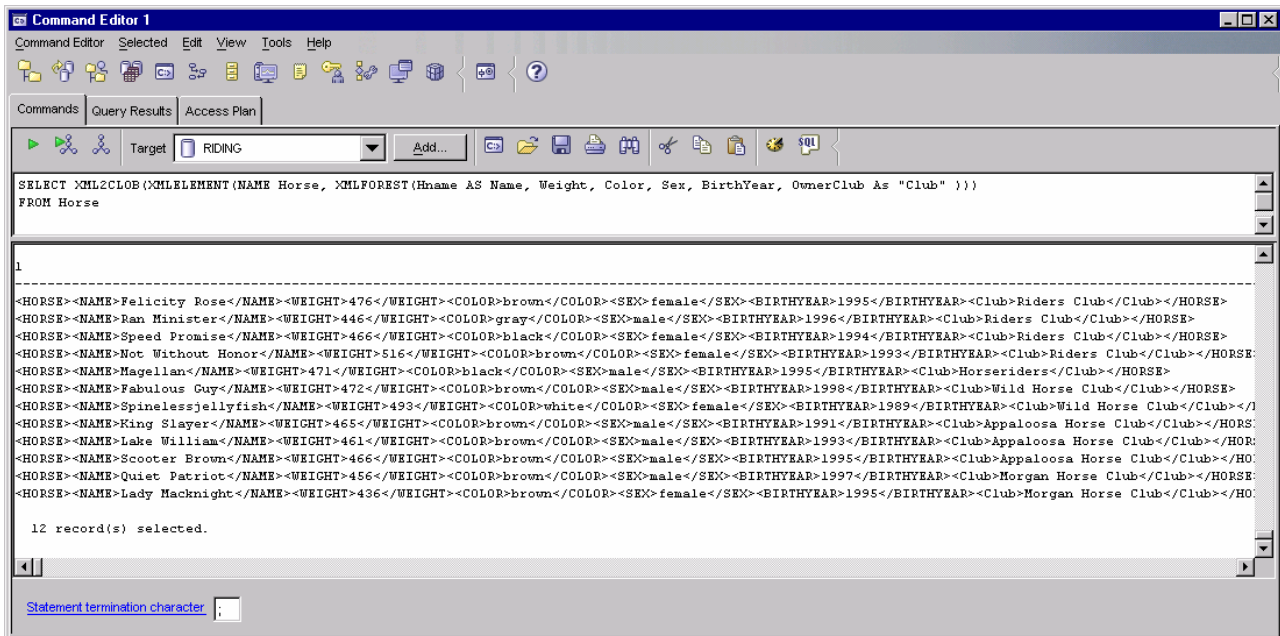
```
SELECT XML2CLOB(XMLELEMENT(NAME Horse, XMLCONCAT(XMLELEMENT(NAME Name, Hname),
XMLELEMENT(NAME Weight, Weight),
XMLELEMENT(NAME Color, Color),
XMLELEMENT(NAME BirthYear, BirthYear),
XMLELEMENT(NAME Sex, Sex),
XMLELEMENT(NAME "Club", OwnerClub))))
FROM Horse
```

```
1
-----
<Horse><NAME>Felicity Rose</NAME><WEIGHT>476</WEIGHT><COLOR>brown</COLOR><BIRTHYEAR>1995</BIRTHYEAR><SEX>female</SEX><Club>Riders Club</Club></Horse>
<Horse><NAME>Pan Minister</NAME><WEIGHT>446</WEIGHT><COLOR>gray</COLOR><BIRTHYEAR>1996</BIRTHYEAR><SEX>male</SEX><Club>Riders Club</Club></Horse>
<Horse><NAME>Speed Promise</NAME><WEIGHT>466</WEIGHT><COLOR>black</COLOR><BIRTHYEAR>1994</BIRTHYEAR><SEX>female</SEX><Club>Riders Club</Club></Horse>
<Horse><NAME>Not Without Honor</NAME><WEIGHT>516</WEIGHT><COLOR>brown</COLOR><BIRTHYEAR>1993</BIRTHYEAR><SEX>female</SEX><Club>Riders Club</Club></Horse>
<Horse><NAME>Magellan</NAME><WEIGHT>471</WEIGHT><COLOR>black</COLOR><BIRTHYEAR>1995</BIRTHYEAR><SEX>male</SEX><Club>Horseriders</Club></Horse>
<Horse><NAME>Fabulous Guy</NAME><WEIGHT>472</WEIGHT><COLOR>brown</COLOR><BIRTHYEAR>1998</BIRTHYEAR><SEX>male</SEX><Club>Wild Horse Club</Club></Horse>
<Horse><NAME>Spinelessjellyfish</NAME><WEIGHT>493</WEIGHT><COLOR>white</COLOR><BIRTHYEAR>1989</BIRTHYEAR><SEX>female</SEX><Club>Wild Horse Club</Club></Horse>
<Horse><NAME>King Slayer</NAME><WEIGHT>465</WEIGHT><COLOR>brown</COLOR><BIRTHYEAR>1991</BIRTHYEAR><SEX>male</SEX><Club>Appaloosa Horse Club</Club></Horse>
<Horse><NAME>Lake William</NAME><WEIGHT>461</WEIGHT><COLOR>brown</COLOR><BIRTHYEAR>1993</BIRTHYEAR><SEX>male</SEX><Club>Appaloosa Horse Club</Club></Horse>
<Horse><NAME>Scooter Brown</NAME><WEIGHT>466</WEIGHT><COLOR>brown</COLOR><BIRTHYEAR>1995</BIRTHYEAR><SEX>male</SEX><Club>Appaloosa Horse Club</Club></Horse>
<Horse><NAME>Quiet Patriot</NAME><WEIGHT>456</WEIGHT><COLOR>brown</COLOR><BIRTHYEAR>1997</BIRTHYEAR><SEX>male</SEX><Club>Morgan Horse Club</Club></Horse>
<Horse><NAME>Lady MacKnight</NAME><WEIGHT>436</WEIGHT><COLOR>brown</COLOR><BIRTHYEAR>1995</BIRTHYEAR><SEX>female</SEX><Club>Morgan Horse Club</Club></Horse>

12 record(s) selected.
```

The same result could be achieved with the following version:

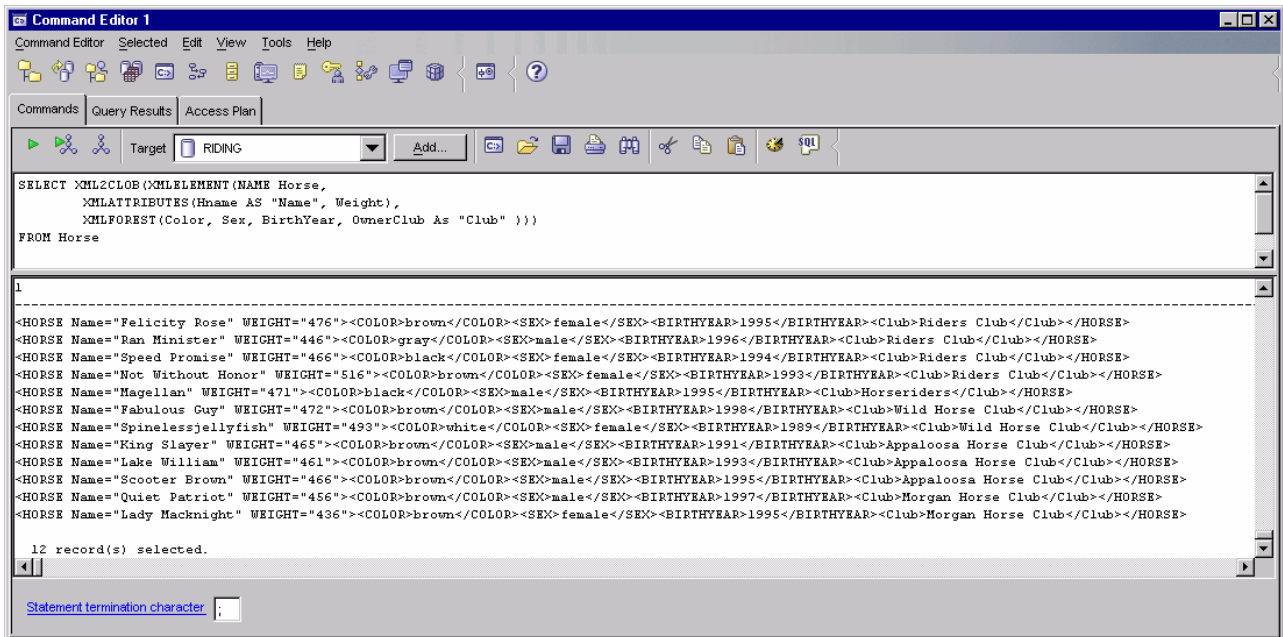
```
SELECT XML2CLOB(XMLELEMENT(NAME Horse, XMLFOREST(Hname AS Name,  
Weight, Color, Sex, BirthYear, OwnerClub As "Club" )))  
FROM Horse
```



Another thing we may want to do is put some data as attributes. We could modify the previous statement so that the horse name and weight can be attributes in the Horse element, instead of sub elements. To do this we will use the XMLATTRIBUTES function (which can only be used as a parameter to the XMLELEMENT function before any sub elements):

```
SELECT XML2CLOB(XMLELEMENT(NAME Horse,  
XMLATTRIBUTES(Hname AS "Name", Weight),  
XMLFOREST(Color, Sex, BirthYear, OwnerClub As "Club" )))  
FROM Horse
```

And the result looks like this:



The next kind of thing we may want to do is aggregate information in many levels. Here is an example:

*Create a **Club** element for each club and a sub element **Horse** for each horse owned by that club. Just show the name of the club and the name of the horse!*

There are several ways to solve this, but, either way, the XMLAGG function will come in handy. Producing the Club element with an attribute Name would be similar to what we did before. The difference here is that there are many horses for each club. We can therefore combine a GROUP BY clause with an XMLAGG function:

```
SELECT XML2CLOB(XMLELEMENT(NAME "Club", XMLATTRIBUTES(OwnerClub AS
"Name"), XMLAGG(XMLELEMENT(NAME "Horse", HName))))
FROM Horse
GROUP BY OwnerClub
```

The result would look like this:

The screenshot shows the 'Command Editor' window. The title bar reads 'Command Editor 1'. The menu bar includes 'Command Editor', 'Selected', 'Edit', 'View', 'Tools', and 'Help'. The toolbar contains various icons for file operations and editing. Below the toolbar are tabs for 'Commands', 'Query Results', and 'Access Plan'. The 'Query Results' tab is active, showing a table with 5 records. The table has 2 columns: 'Club' and 'Horse'. The records are as follows:

Club	Horse
Appaloosa Horse Club	King Slayer
Appaloosa Horse Club	Lake William
Appaloosa Horse Club	Scooter Brown
Morgan Horse Club	Lady Macknight
Morgan Horse Club	Quiet Patriot

At the bottom of the window, there is a status bar that reads 'Statement termination character : '.

We could also use the `XMLAGG` function to aggregate all the rows of the result into one root element:

```
SELECT XML2CLOB(XMLAGG(newcol))
FROM (SELECT XMLELEMENT(NAME "Club",
                        XMLATTRIBUTES(OwnerClub AS "Name"),
                        XMLAGG(XMLELEMENT(NAME "Horse", HName))) AS newcol
      FROM Horse
      GROUP BY OwnerClub) AS innertable
```

The result is now a sequence of **Club** elements:

The screenshot shows the 'Command Editor 1' window. The title bar includes standard window controls. The menu bar contains 'Command Editor', 'Selected', 'Edit', 'View', 'Tools', and 'Help'. The toolbar has icons for file operations, database connections, and help. Below the toolbar are tabs for 'Commands', 'Query Results', and 'Access Plan'. The 'Commands' tab is active, showing a 'Target' dropdown set to 'RIDING' and an 'Add...' button. The main text area contains an SQL query:

```
SELECT XML2CLOB(XMLAGG(newcol))
FROM (SELECT XMLELEMENT(NAME "Club",
XMLATTRIBUTES(OwnerClub AS "Name"),
XMLAGG(XMLELEMENT(NAME "Horse", HName))) AS newcol
      FROM Horse
      GROUP BY OwnerClub) AS innertable
```

Below the query, the 'Query Results' tab is active, displaying the output of the query. It shows a single record with the following XML structure:

```
1
-----
<Club Name="Appaloosa Horse Club"><Horse>King Slayer</Horse><Horse>Lake William</Horse><Horse>Scooter Brown</Horse></Club><Club Name="H
```

At the bottom, it states '1 record(s) selected.' Below the results, there is a 'Statement termination character' field with a semicolon (;) entered.

We can of course place them inside a root element Clubs:

```
SELECT XML2CLOB(XMLELEMENT(NAME "Clubs", XMLAGG(newcol)))
FROM (SELECT XMLELEMENT(NAME "Club",
                        XMLATTRIBUTES(OwnerClub AS "Name"),
                        XMLAGG(XMLELEMENT(NAME "Horse", HName))) AS newcol
      FROM Horse
      GROUP BY OwnerClub) AS innertable
```

We can of course combine any SQL structure we want with the XML functions. We could for example add an attribute to the element Club that indicates how many horses the club has and one for how many riders the club has:

```
SELECT XML2CLOB(XMLELEMENT(NAME "Clubs", XMLAGG(newcol)))
FROM (SELECT XMLELEMENT(NAME "Club",
                        XMLATTRIBUTES(OwnerClub AS "Name",
                                      COUNT(HName) AS "AmountOfHorses",
                                      (SELECT COUNT(*) FROM Rider
                                       WHERE MemberClub = OwnerClub) AS "AmountOfRiders"),
                        XMLAGG(XMLELEMENT(NAME "Horse", HName))) AS newcol
      FROM Horse
      GROUP BY OwnerClub) AS innertable
```

The resulting XML would look like this:

```
<Clubs>
  <Club Name="Appaloosa Horse Club" AmountOfHorses="3" AmountOfRiders="2">
    <Horse>King Slayer</Horse>
    <Horse>Lake William</Horse>
    <Horse>Scooter Brown</Horse>
  </Club>
  <Club Name="Horseriders" AmountOfHorses="1" AmountOfRiders="4">
    <Horse>Magellan</Horse>
  </Club>
  <Club Name="Morgan Horse Club" AmountOfHorses="2" AmountOfRiders="3">
    <Horse>Lady Macknight</Horse>
    <Horse>Quiet Patriot</Horse>
  </Club>
  <Club Name="Riders Club" AmountOfHorses="4" AmountOfRiders="4">
    <Horse>Felicity Rose</Horse>
    <Horse>Not Without Honor</Horse>
    <Horse>Ran Minister</Horse>
    <Horse>Speed Promise</Horse>
  </Club>
  <Club Name="Wild Horse Club" AmountOfHorses="2" AmountOfRiders="3">
    <Horse>Fabulous Guy</Horse>
    <Horse>Spinelessjellyfish</Horse>
  </Club>
</Clubs>
```

Finally we can look at an example that makes use of the functions of this section together with the extract functions from section 4.1.2. (You will need to connect to the **Book** database for this one.)

*Create an XML element **Authors** with one element for each **Author** with the author's name as an attribute and the amount of books as another attribute!*

We can do this with the following statement:

```
SELECT XML2CLOB(XMLELEMENT(NAME "Authors",
    XMLAGG(XMLELEMENT(NAME "Author",
        XMLATTRIBUTES(atable.returnedVarchar AS "Name",
            (SELECT COUNT(*)
                FROM xmlcol
                WHERE atable.returnedVarchar IN (SELECT returnedVarchar
                    FROM table(db2xml.extractVarchars(xmlidoc, '/Book/Author/@Name')) AS temp)
            ) AS "AmountOfBooks"))))
FROM xmlcol, table(db2xml.extractVarchars(xmlidoc, '/Book/Author/@Name')) AS atable
```

And the result would be one XML element **Authors**, shown here indented:

```
<Authors>
  <Author AmountOfBooks="1" Name="John Craft"/>
  <Author AmountOfBooks="1" Name="Arnie Bastoft"/>
  <Author AmountOfBooks="1" Name="Meg Gilmand"/>
  <Author AmountOfBooks="1" Name="Chris Ryan"/>
  <Author AmountOfBooks="1" Name="Alan Griff"/>
  <Author AmountOfBooks="1" Name="Marty Faust"/>
  <Author AmountOfBooks="1" Name="Celine Biceau"/>
  <Author AmountOfBooks="1" Name="Carl Sagan"/>
  <Author AmountOfBooks="1" Name="Leslie Brenner"/>
  <Author AmountOfBooks="2" Name="Marie Franksson"/>
  <Author AmountOfBooks="1" Name="Jakob Hanson"/>
  <Author AmountOfBooks="2" Name="Sam Davis"/>
  <Author AmountOfBooks="1" Name="Mimi Pappas"/>
  <Author AmountOfBooks="2" Name="Marie Franksson"/>
  <Author AmountOfBooks="1" Name="Franc Desteille"/>
  <Author AmountOfBooks="1" Name="Carl George"/>
  <Author AmountOfBooks="1" Name="Peter Feldon"/>
  <Author AmountOfBooks="1" Name="Lilian Carrera"/>
  <Author AmountOfBooks="1" Name="Auna Gonzales Perre"/>
  <Author AmountOfBooks="1" Name="Kostas Andrianos"/>
  <Author AmountOfBooks="1" Name="Andreas Shultz"/>
  <Author AmountOfBooks="1" Name="Antje Liedderman"/>
  <Author AmountOfBooks="1" Name="Christina Ohlsen"/>
  <Author AmountOfBooks="2" Name="Sam Davis"/>
  <Author AmountOfBooks="1" Name="Alicia Bing"/>
  <Author AmountOfBooks="1" Name="Pierre Zargone"/>
  <Author AmountOfBooks="1" Name="Linda Evans"/>
  <Author AmountOfBooks="1" Name="Chuck Morrisson"/>
  <Author AmountOfBooks="1" Name="Kay Morrisson"/>
  <Author AmountOfBooks="1" Name="James Patterson"/>
  <Author AmountOfBooks="1" Name="Peter de Jonge"/>
</Authors>
```

4.4 Assignments

Solve the following questions:

1. Make a list of all the publishers! (No duplicates)
2. How many educational books have been written originally in English?
3. How many translations are there for each book that was originally in English? Even books with no translations should be in the result.
4. Which books were written by more than two authors? (Show the book titles!)
5. Make a list of all non-Swedish authors with their e-mail addresses and year of birth! (No duplicates)
6. Change the year of birth of the Australian author of the book "Archeology in Egypt" to 1966!
7. Create a **Rider** element for each rider of the club Horseriders. The **Rider** element shall have a **Name** attribute, a **Weight** attribute and an attribute with the amount of races this rider has finished. The result should be one **Riders** element with all the **Rider** elements inside it.
8. Create a **Languages** element with **Language** sub elements. Each **Language** element shall have a **Value** attribute (with the actual language). Create one **Language** element for each language that appears as the original language of any book. Each **Language** element shall have one or more **Book** sub elements, based on the books' original language, with the book title as its content.

5 Voluntary Exercises

In this chapter we will look at DB2's facilities for transforming relational data into XML documents. Even though this part is not a requirement for the course it can be interesting to know how to create XML documents from data stored in relational tables.

In this chapter we will go through the following:

- Create a database (relational database).
- Enable the database for XML (as an XML collection) and compose XML documents from the data in the XML collection (the database).
- Extract XML documents into XML files.
- Store XML documents in an XML column (which is similar to what we did with the script in section 3.1).

All the files required in this chapter, as well as a text file with all the commands, are available at:

\\Db-srv-1\StudentCourseMaterial\IS4 spring 2006\DB2-XML\Files for Voluntary Exercises

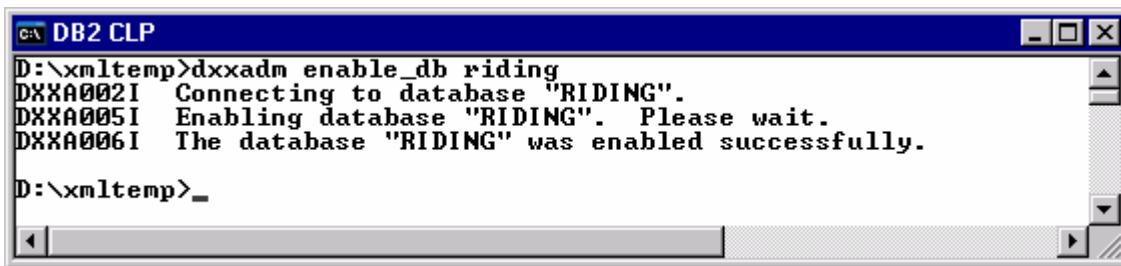
5.1 Create the database

The database can easily be created and populated by running the two scripts (see section 3.2). You have probably already done this in order to complete the exercises in chapter 4.

5.2 Enable the database for XML (as an XML collection) and compose XML documents

When the database has been created, it is just an ordinary relational database. If the database is going to be used as an XML collection then it has to be enabled for XML. That is done as follows:

- Start a DB2 Command Window (Start > Programs > Databases > IBM-DB2 > Command Line Tools > Command Window)
- Execute this command in the Command Window:
dxxadm enable_db riding



```
C:\ DB2 CLP
D:\xmltemp>dxxadm enable_db riding
DXXA002I  Connecting to database "RIDING".
DXXA005I  Enabling database "RIDING". Please wait.
DXXA006I  The database "RIDING" was enabled successfully.

D:\xmltemp>_
```

When that is done there should be a few more tables in the database. Those tables are used by the XML extender. For example the table DTD_REF contains information about DTD files.

The next step is to enable the XML collection. That is not a necessary step. To enable the XML collection we need to have a DAD file. The DAD file is specified when enabling an XML collection. The DAD file can contain information on how to compose XML documents from the XML collection and how to decompose XML documents into the XML collection. If the XML collection is not enabled, then the DAD file must be specified every time an XML document is to be composed or decomposed.

In this exercise we will just specify rules for composition of XML documents in the DAD file and we will enable the XML collection.

First we need to create a DAD file. To do that we need to know how we want the XML document to be structured and where all the XML data are stored in the database. In other words we need to define the XML document structure and map it to the XML collection tables and columns.

Here is the structure for the XML documents that we want to compose:

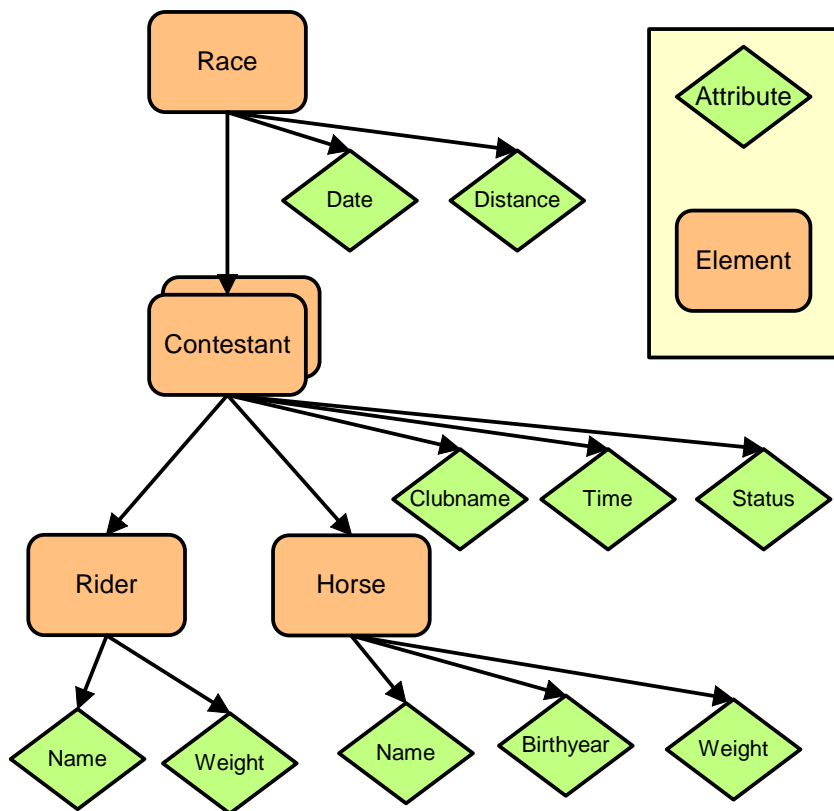


Figure 5 Structure of elements and attributes for the XML documents

The Race element will be the root element of the XML documents. The Race element consists of two attributes (Date, Distance) and one element (Contestant). The Contestant element can appear several times within a Race element. Each Contestant element has three attributes (Clubname, Time, Status) and two elements (Rider, Horse), which in turn have two and three attributes respectively.

An XML document with that structure would look like this:

```
<?xml version="1.0" standalone="yes"?>
<!DOCTYPE Race SYSTEM "">
<Race Date="2001-06-05" Distance="1000">
  <Contestant Clubname="Appaloosa Horse Club" Status="finished" Time="00:02:02">
    <Rider Name="Bill Spawr" Weight="48"></Rider>
    <Horse Name="Lake William" Weight="461" Birthyear="1993"></Horse>
  </Contestant>
  <Contestant Clubname="Horseriders" Status="finished" Time="00:02:02">
    <Rider Name="Warren Stute" Weight="55"></Rider>
    <Horse Name="Magellan" Weight="471" Birthyear="1995"></Horse>
  </Contestant>
  <Contestant Clubname="Wild Horse Club" Status="walkover">
    <Rider Name="Simon Bray" Weight="53"></Rider>
    <Horse Name="Spinelessjellyfish" Weight="493" Birthyear="1989"></Horse>
  </Contestant>
</Race>
```

Creating a DAD file, with the mapping for the transformation from XML data stored in the XML collection into XML documents, is a little more complicated. In the DAD file that we will create, we will use SQL mapping. SQL mapping works as follows:

“SQL mapping allows simple and direct mapping from relational data to XML documents through a single SQL statement... SQL mapping is used for composition; it is not used for decomposition...The SQL_stmt maps the columns in the SELECT clause to XML elements or attributes that are used in the XML document. When defined for composing XML documents, the column names in the SQL statement’s SELECT clause are used to define the value of an attribute_node or a content of text_node. The FROM clause defines the tables containing the data; the WHERE clause specifies the join and search condition.” (XML Extender Administration and Programming).

In addition to that, the SQL statement must contain an ORDER BY clause, where the columns that identify the rows uniquely must be listed. The column names listed in the SELECT clause must be unique, if two columns have the same name then one of them must be renamed using the AS statement (example: SELECT address, address AS address2 ...).

Before we start with the structure we defined above, let’s look at a simpler case!

Here is a simple example of a valid SQL statement:

```
SELECT cname, address FROM club ORDER BY cname
```

Cname is the primary key of the club table, therefore it appears in the ORDER BY clause.

It is then possible to place the values of the columns into elements or attributes of the XML document. Here is how it’s done:

To get an element Club we define (in the DAD file) the following tag:

In the DAD file:

```
<element_node name="Club">  
</element_node>
```

Will produce in the XML document:

```
<Club>  
</Club>
```

To get an attribute address in the Club element:

In the DAD file:

```
<element_node name="Club">  
  <attribute_node name="address">  
  </attribute_node>  
</element_node>
```

Will produce in the XML document:

```
<Club address="">  
</Club>
```

To add a value to the **address** attribute from the SQL statement:

In the DAD file:

```
<element_node name="Club">  
  <attribute_node name="address">  
    <column name="address"/>  
  </attribute_node>  
</element_node>
```

Will produce in the XML document:

```
<Club address="my address">  
</Club>
```

To add a value to the **Club** element from the SQL statement:

In the DAD file:

```
<element_node name="Club">  
  <attribute_node name="address">  
    <column name="address"/>  
  </attribute_node>  
  <text_node>  
    <column name="cname"/>  
  </text_node>  
</element_node>
```

Will produce in the XML document:

```
<Club address="my address">  
  My club  
</Club>
```

So if we put all this (and a little more) together, we should have a DAD file:

First we start with two XML lines. DAD files are also XML files, that follow the rules specified in a DTD file (dad.dtd).

```
<?xml version="1.0"?>  
<!DOCTYPE DAD SYSTEM " C:\Program  
Files\IBM\SQLLIB\samples\db2xml\dtd\dad.dtd ">
```

The DAD element is the root element of any DAD file.

```
<DAD>
```

Validation is applicable only when the DAD file is used for decomposition, therefore we set it to NO.

```
<validation>NO</validation>
```

The Xcollection element is where all our code is placed.

```
<Xcollection>
```

The SQL statement is placed within an element called SQL_stmt

```
<SQL_stmt> SELECT cname, address FROM  
club ORDER BY cname </SQL_stmt>
```

These lines make sure that the resulting XML document contains standard XML lines. The DOCTYPE has to always match the root element of the XML document, therefore we set it to Club.

```
<prolog>?xml version="1.0"?</prolog>  
<doctype>!DOCTYPE Club SYSTEM ""</doctype>
```

This is to define the root element of the resulting XML document

```
<root_node>
```

This is the structure of elements and attributes that we have defined

```
<element_node name="Club">  
  <attribute_node name="address">  
    <column name="address"/>  
  </attribute_node>  
  <text_node>  
    <column name="cname"/>  
  </text_node>  
</element_node>
```

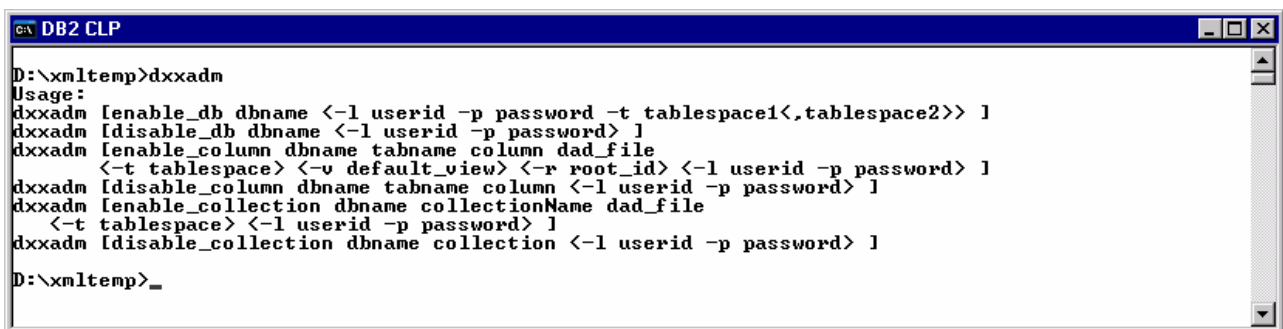
These are the end tags of all the elements

```
</root_node>  
</Xcollection>  
</DAD>
```

Now that the DAD file is ready we can enable the XML collection. The DAD file must be saved as a file with the extension DAD (for example as D:\xmltemp\club.dad). In the DB2 Command Window we can execute the following command.

dxxadm

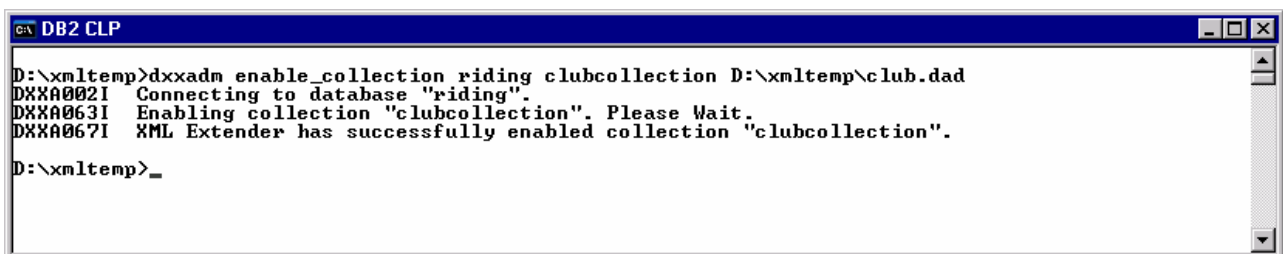
A response with the correct syntax of the dxxadm command comes up:



```
C:\ DB2 CLP  
D:\xmltemp>dxxadm  
Usage:  
dxxadm [enable_db dbname <-l userid -p password -t tablespace1[,tablespace2]>] 1  
dxxadm [disable_db dbname <-l userid -p password>] 1  
dxxadm [enable_column dbname tablename column dad_file  
  <-t tablespace> <-v default_view> <-r root_id> <-l userid -p password>] 1  
dxxadm [disable_column dbname tablename column <-l userid -p password>] 1  
dxxadm [enable_collection dbname collectionName dad_file  
  <-t tablespace> <-l userid -p password>] 1  
dxxadm [disable_collection dbname collection <-l userid -p password>] 1  
D:\xmltemp>_
```

Now for the complete command that enables an XML collection:

dxxadm enable_collection riding clubcollection D:\xmltemp\club.dad



```
C:\ DB2 CLP  
D:\xmltemp>dxxadm enable_collection riding clubcollection D:\xmltemp\club.dad  
DXXA002I Connecting to database "riding".  
DXXA063I Enabling collection "clubcollection". Please Wait.  
DXXA067I XML Extender has successfully enabled collection "clubcollection".  
D:\xmltemp>_
```

clubcollection is the XML collection's name, there can be more than one XML collection enabled on the same database.

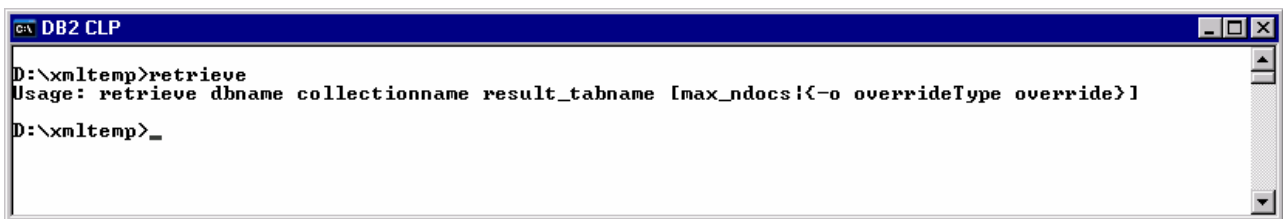
D:\xmltemp\club.dad is the location of the DAD file.

When the XML collection was enabled, a new row was created in the XML_USAGE table. The new row contains information about the XML collection (the collection name, the DAD file, etc).

Note that if for some reason the DAD file needs to be altered, it is not enough to change the file. The XML collection should be disabled (`dxadm disable` command) and then enabled with the altered DAD file. It is only then that the XML collection sees the changes!

Extracting XML documents can be done with the `retrieve` command. Try to execute the following command in the Command Window to get more information about the `retrieve` command:

`retrieve`



```
DB2 CLP
D:\xmltemp>retrieve
Usage: retrieve dbname collectionname result_tabname [max_ndocs!<-o overrideType override>]
D:\xmltemp>
```

The `retrieve` command requires a `result_tablename` argument. It is this table that the XML documents are going to be stored in. Before we can execute the `retrieve` command successfully, we have to define a new table to receive the results. Here is a table definition:

`CREATE TABLE results(xml doc DB2XML.XMLVARCHAR)`

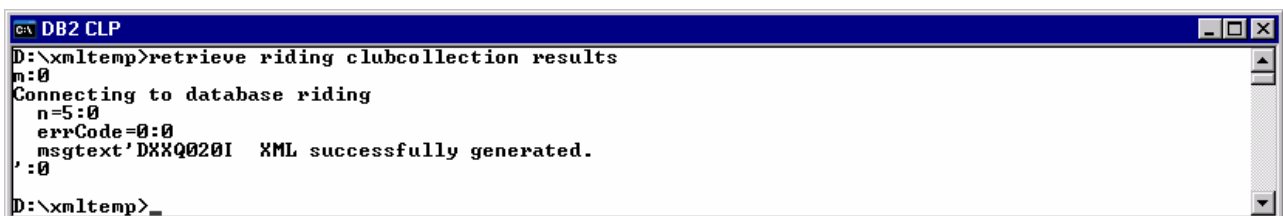
DB2XML.XMLVARCHAR is a user defined type that comes with the XML extender. This type is similar to VARCHAR. We use this type because it is compatible with XML extender user defined functions that we will use later.

- Create a table according to the definition above! You may need to connect to the database first with the command `connect to riding`.

When this table has been created, it can be used as a result table for the `retrieve` command.

Here is the complete `retrieve` command:

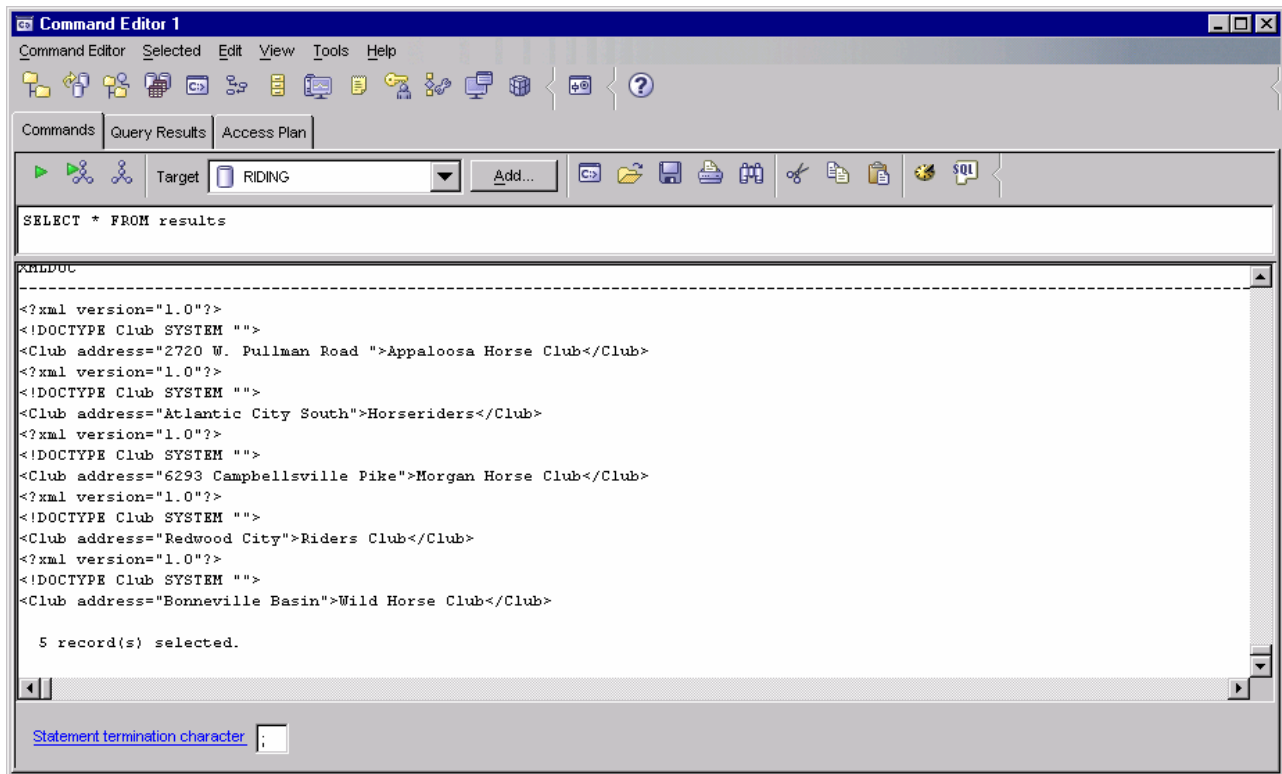
`retrieve riding clubcollection results`



```
DB2 CLP
D:\xmltemp>retrieve riding clubcollection results
n:0
Connecting to database riding
n=5:0
errCode=0:0
msgtext'DXXXQ020I XML successfully generated.
':0
D:\xmltemp>
```

The XML documents (5 documents: `n=5.0`) that have been composed should be stored in the `results` table. You can easily check the contents of the `results` table by executing the following SQL statement:

`SELECT * FROM results`



Let's go back now to the more complicated structure (from page 46), and create a DAD file.

First we must have an SQL statement that returns all the columns that we need for the XML elements and attributes. The following SQL statement returns those columns:

```
SELECT date(race.racetime) as racedate, race.distance, clubname, finishingtime, status,  
rname, r.weight AS rweight, hname, h.weight AS hweight, birthyear  
FROM race, horse AS h, rider AS r, contestants AS c  
WHERE race.raceid = c.raceid  
AND ridername = rname  
AND clubname = Memberclub  
AND clubname = ownerclub  
AND horsename = hname
```

This is a valid SQL statement but it is not valid as a DAD SQL statement. A DAD SQL statement requires an ORDER BY clause that should contain the column that can identify uniquely each entity. That is, one column for each entity. This facility of DB2 XML extender is quite new and it may appear to behave inconsistently. Not all entities' identifiers need to be part of the ORDER BY clause, only the ones that lead to a level where many elements of the same type can appear. To make that more understandable we can look at our structure and the entities that exist:

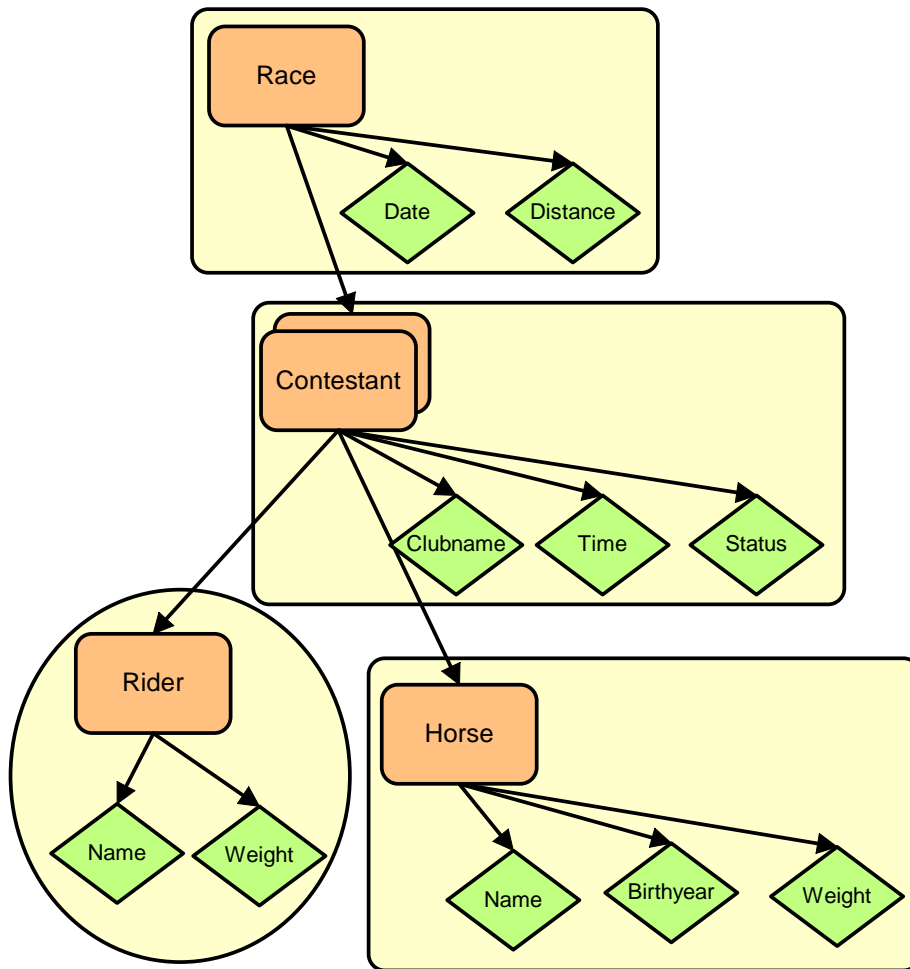


Figure 6 Entities of the XML structure

In this structure each entity is associated with one table. So the unique identifier for each entity is the primary key (or a candidate key) of the associated table. Now there is one problem remaining. There can only be one column that identifies uniquely an entity, but the tables contestant, rider and horse require more than one column to identify a row uniquely. (Of course we only need to include the unique identifiers of the tables race and contestant. The tables rider and horse produce only one entry per contestant, while there can be several contestants per race.) One way to solve this problem is to use the table expression and the `generate_unique()` function to produce a single column unique identifier¹⁰. After making all these changes in the SQL statement, it should look like this:

¹⁰ In certain cases this technique may not work. In those cases we may need to create a unique identifier for an entity in a different way, for example by concatenating the components of the primary key.

```
SELECT race.raceid, date(race.racetime) AS racedate, race.distance, cid, clubname,  
status, finishingtime, rname, r.weight AS rweight, hname, h.weight AS hweight, birthyear  
FROM race, table(SELECT generate_unique() as cid, raceid, ridername, clubname,  
horsename, finishingtime, status FROM contestants) AS c, rider AS r, horse AS h  
WHERE race.raceid = c.raceid  
AND ridername = rname  
AND clubname = memberclub  
AND clubname = ownerclub  
AND horsename = hname  
ORDER BY raceid, cid
```

Creating the element and attribute structure of the XML document is not different from before.

We start with the root element and we continue deeper into the structure.

The root element is the Race element.

Definition in DAD file	Produces in XML document
------------------------	--------------------------

<code><element_node name="Race"></code>	<code><Race></code>
<code></element_node></code>	<code></Race></code>

Now for the attributes of the Race element.

Definition in DAD file	Produces in XML document
------------------------	--------------------------

<code><element_node name="Race"></code>	<code><Race Date="" Distance=""></code>
<code> <attribute_node name="Date"></code>	<code></Race></code>
<code> </attribute_node></code>	
<code> <attribute_node name="Distance"></code>	
<code> </attribute_node></code>	
<code></element_node></code>	

Now for the Contestant element which can exist several times within a Race element.

Definition in DAD file	Produces in XML document
------------------------	--------------------------

<code><element_node name="Race"></code>	<code><Race Date="" Distance=""></code>
<code> <attribute_node name="Date"></code>	<code><Contestant></code>
<code> </attribute_node></code>	<code></Contestant></code>
<code> <attribute_node name="Distance"></code>	<code><Contestant></code>
<code> </attribute_node></code>	<code></Contestant></code>
<code> <element_node name="Contestant"</code>	<code>.</code>
<code> multi_occurrence="YES"></code>	<code>.</code>
<code> </element_node></code>	<code>.</code>
<code></element_node></code>	<code></Race></code>

After adding the rest of the elements and attributes of the structure we should have the following:

```
<element_node name="Race">
  <attribute_node name="Date">
  </attribute_node>
  <attribute_node name="Distance">
  </attribute_node>
  <element_node name="Contestant" multi_occurrence="YES">
    <attribute_node name="Clubname">
    </attribute_node>
    <attribute_node name="Status">
    </attribute_node>
    <attribute_node name="Time">
    </attribute_node>
    <element_node name="Rider">
      <attribute_node name="Name">
      </attribute_node>
      <attribute_node name="Weight">
      </attribute_node>
    </element_node>
    <element_node name="Horse">
      <attribute_node name="Name">
      </attribute_node>
      <attribute_node name="Weight">
      </attribute_node>
      <attribute_node name="Birthyear">
      </attribute_node>
    </element_node>
  </element_node>
</element_node>
```

The last thing to do is to place the values from the SQL statement in the structure. It is important that the order that the values appear in the SQL statement is the same with the order that they appear in the XML structure (even though there can be columns in the SQL statement that do not appear in the XML structure). When that is done, all the parts of the DAD file are done. By putting them together (and changing the XML declaration and the DOCTYPE element of the resulting XML document) we should get this:

```
<?xml version="1.0"?>
<!DOCTYPE DAD SYSTEM " C:\Program Files\IBM\SQLLIB\samples\db2xml\dtd\dad.dtd">
<DAD>
<validation>NO</validation>
<Xcollection>
```

```
<SQL_stmt>
SELECT race.raceid, date(race.racetime) as racedate, race.distance, cid, clubname, status,
finishingtime, rname, r.weight AS rweight, hname, h.weight AS hweight, birthyear FROM race,
table(SELECT generate_unique() as cid, raceid, ridername, clubname, horsename, finishingtime,
status FROM contestants) AS c, rider AS r, horse AS h WHERE race.raceid = c.raceid AND
ridername = rname AND clubname = memberclub AND clubname = ownerclub AND horsename =
hname ORDER BY raceid, cid
</SQL_stmt>
<prolog>?xml version="1.0" standalone="no"?</prolog>
<doctype>!DOCTYPE Race SYSTEM "d:\xmltemp\race.dtd"</doctype>
<root_node>
<element_node name="Race">
  <attribute_node name="Date">
    <column name="racedate"/>
  </attribute_node>
  <attribute_node name="Distance">
    <column name="distance"/>
  </attribute_node>
  <element_node name="Contestant" multi_occurrence="YES">
    <attribute_node name="Clubname">
      <column name="clubname"/>
    </attribute_node>
    <attribute_node name="Status">
      <column name="status"/>
    </attribute_node>
    <attribute_node name="Time">
      <column name="finishingtime"/>
    </attribute_node>
    <element_node name="Rider">
      <attribute_node name="Name">
        <column name="rname"/>
      </attribute_node>
      <attribute_node name="Weight">
        <column name="rweight"/>
      </attribute_node>
    </element_node>
    <element_node name="Horse">
      <attribute_node name="Name">
        <column name="hname"/>
      </attribute_node>
      <attribute_node name="Weight">
        <column name="hweight"/>
      </attribute_node>
      <attribute_node name="Birthyear">
        <column name="birthyear"/>
      </attribute_node>
    </element_node>
  </element_node>
</element_node>
</root_node>
</Xcollection>
</DAD>
```

The DAD file contains information about the XML declaration and the DOCTYPE element of the XML documents to be composed. This information is the following:

The XML document is composed according to XML version 1.0 and it is not standalone (it is associated to a DTD file):

```
<prolog>?xml version="1.0" standalone="no"?</prolog>
```

The DOCTYPE of the XML document is Race. That means that the root element of the XML document is an element called Race. The SYSTEM specifies that the XML document is supposed to follow the rules in the DTD file d:\xmltemp\race.dtd:

```
<doctype>!DOCTYPE Race SYSTEM "d:\xmltemp\race.dtd"</doctype>
```

The file d:\xmltemp\race.dtd does not exist yet. In section 5.4 we will create this DTD file and we will use the XML documents composed with this DAD file.

Assuming that the DAD file has been saved as d:\xmltemp\race.dad we can enable an XML collection called racecollection by submitting the following command in the Command Window:

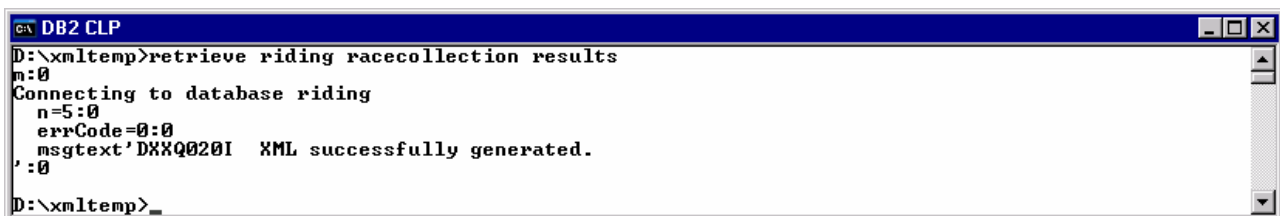
```
dxxadm enable_collection riding racecollection d:\xmltemp\race.dad
```



```
C:\ DB2 CLP
D:\>\xmltemp>dxxadm enable_collection riding racecollection d:\xmltemp\race.dad
DXXA0021I Connecting to database "riding".
DXXA0631I Enabling collection "racecollection". Please Wait.
DXXA0671I XML Extender has successfully enabled collection "racecollection".
D:\>\xmltemp>_
```

When the new XML collection has been enabled, use the retrieve command to compose XML documents and place them in the results table (You may want to remove the previous XML documents from the results table first) :

```
retrieve riding racecollection results
```



```
C:\ DB2 CLP
D:\>\xmltemp>retrieve riding racecollection results
m:0
Connecting to database riding
n=5:0
errCode=0:0
msgtext'DXXQ0201I XML successfully generated.
':0
D:\>\xmltemp>_
```

Five XML documents are now stored in the table results.

5.3 Extract XML documents into XML files

So far we have composed XML documents and stored them in a table. It can be desired to extract these XML documents from the database and keep them as separate files. To do that, we will use the XML extender's **Content** function.

Like all other functions, the **Content** function can be used in a **SELECT** statement. The **Content** function has three different sets of parameters. The one that we will use is the following:

Content(xmlobj, filename)

xmlobj is the XML document as an **XMLVARCHAR**.

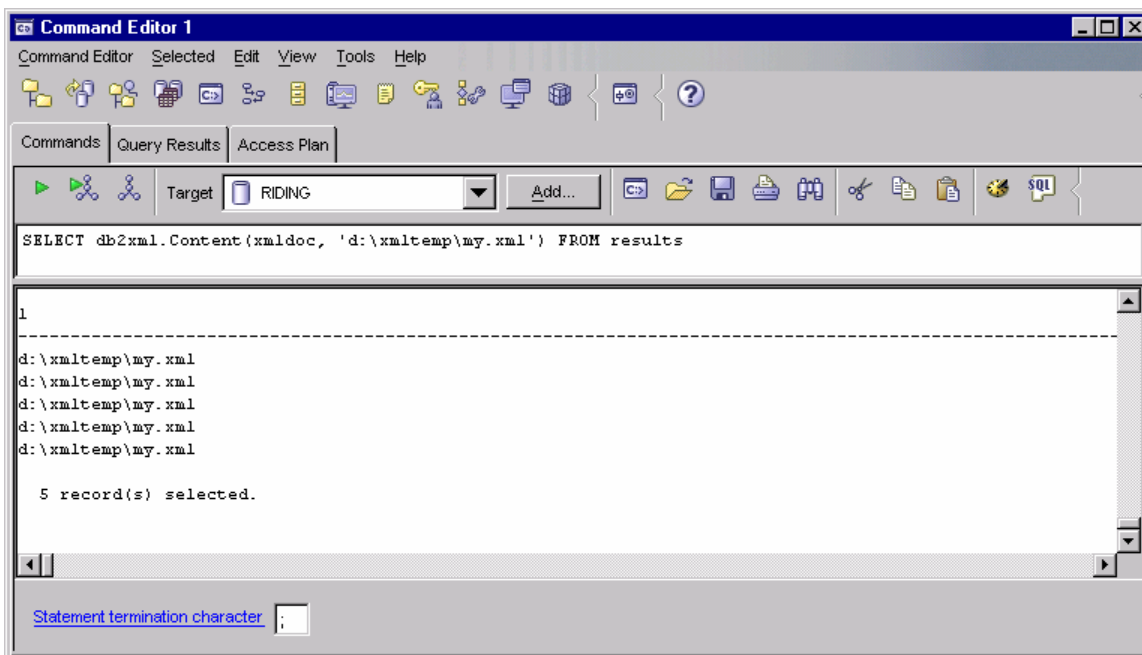
Filename is a string with the fully qualified filename and location of the file where the XML document will be saved.

When this function is executed it returns the filename to where the XML document was saved.

Here is an example of how to use this function:

```
SELECT db2xml.Content(xmldoc, 'd:\xmltemp\my.xml') FROM results
```

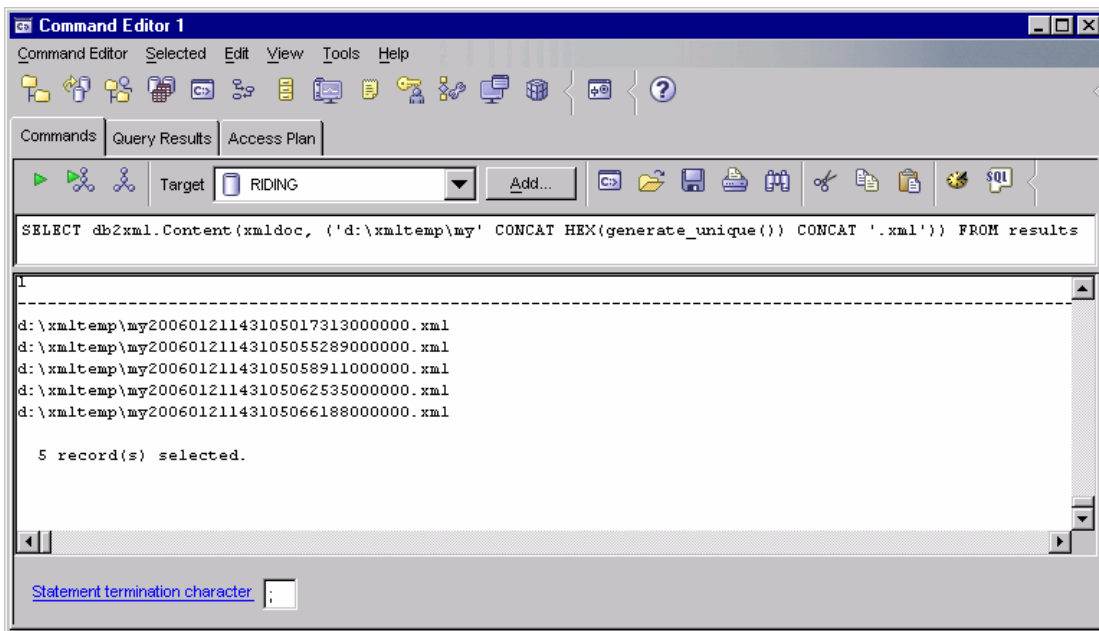
This command produces a file called **d:\xmltemp\my.xml** which contains the XML document that is stored in the **xmlidoc** column of the **results** table. The problem with this command is that it tries to save each and every XML document from the **xmlidoc** column as a file called **d:\xmltemp\my.xml**. Consequently only the last XML document gets saved. The next figure shows what this command returns:



An easy way to produce unique names for all the XML files saved, is to use the `generate_unique()` function to produce the filename:

```
SELECT db2xml.Content(xml doc, ('d:\xmltemp\my' CONCAT HEX(generate_unique())  
CONCAT '.xml')) FROM results
```

This command will produce a unique key for every row in the `results` table, and then concatenate a hexadecimal representation of that unique key into the filename. The next figure shows a result of this command:



The files are now stored on the hard disk and can be viewed with any editor, attached to an email, etc.

5.4 Store XML documents in an XML column

In this section we will create an XML column and store in it the XML documents that we generated before. This is basically the same procedure that we followed in section 3.1 when we created a database and stored in it the XML documents for the books. In this section we will look closer and in more detail at how the procedure works. We will do the following:

1. Create a database with a table where the XML documents will be stored
 2. Enable the database for XML
 3. Prepare a DTD for controlling the incoming XML documents
 4. Store the DTD in the DTD_REF table
 5. Prepare a DAD file for the XML column
 6. Enable the XML column
 7. Insert XML documents into the XML column
- Start by creating a database! You will need to disconnect from the other database if you are still connected. Use the command `DISCONNECT riding`.

Here is a command that creates a database:

```
CREATE DATABASE myxmlcol
```

The database is now ready to be enabled for XML.

- Enable the database for XML by issuing the following command in the Command Window:

```
dxxadm enable_db myxmlcol
```

- Connect to the new database and create a table for the XML documents! The table should have a column of one of the three XML extender data types (XMLVARCHAR, XMLCOLB, XMLFILE). Here we use XMLVARCHAR.

```
CONNECT TO myxmlcol  
CREATE TABLE xmlcol (xmldoc DB2XML.XMLVARCHAR)
```

Note that this table can contain many other columns. Those columns do not interfere with the XML column.

When an XML document is inserted into the database, it has to be controlled. If there is no control of incoming XML documents, the database will soon become corrupt. To control an XML document we need a set of rules of what is and is not allowed. Those rules can be defined in a DTD file.

Before defining a DTD, we must know the exact structure of the XML documents that we want the DTD file to control (and accept). The XML documents that we want to insert into the XML column, are the ones we created earlier from the XML data in the XML collection. So the structure is already defined.

Now let's create a DTD file to represent that structure.

First we have a Race element.

```
<!ELEMENT Race>
```

The Race element has a sub-element called Contestant, that can occur zero or more times (denote this with an asterisk after the element name).

```
<!ELEMENT Race (Contestant*)>
```

The Race element has two attributes (Date and Distance).

```
<!ELEMENT Race (Contestant*)>  
<!ATTLIST Race  
    Date CDATA #REQUIRED  
    Distance CDATA #REQUIRED>
```

We continue with the Contestant element. <!ELEMENT Contestant>

The Contestant element has two sub-elements called Rider and Horse, that can occur once and only once within a Contestant element. <!ELEMENT Contestant (Rider, Horse)>

The Contestant element has three attributes (Clubname, Status and Time) The first two have to be there, the third can be missing. Status can only be one of four predefined values: finished, walkover, disqualified and dropout.

```
<!ELEMENT Contestant (Rider, Horse)>
<!ATTLIST Contestant
  Clubname CDATA #REQUIRED
  Status (finished | walkover | disqualified
| dropout) #REQUIRED
  Time CDATA #IMPLIED>
```

The Rider element. The Rider element has no content. <!ELEMENT Rider EMPTY>

The Rider element has two attributes (Name and Weight). Name is required, Weight is not.

```
<!ELEMENT Rider EMPTY>
<!ATTLIST Rider
  Name CDATA #REQUIRED
  Weight CDATA #IMPLIED>
```

The Horse element. The Horse element has no content <!ELEMENT Horse EMPTY>

The Horse element has three attributes (Name, Weight and Birthyear). Only Name is required

```
<!ELEMENT Horse EMPTY>
<!ATTLIST Horse
  Name CDATA #REQUIRED
  Weight CDATA #IMPLIED
  Birthyear CDATA #IMPLIED>
```

- Put all the elements together and save the file, for example as d:\xmltemp\race.dtd

Here is the content of the file race.dtd:

```
<!ELEMENT Race (Contestant*)>
<!ATTLIST Race
  Date CDATA #REQUIRED
  Distance CDATA #REQUIRED>
<!ELEMENT Contestant (Rider, Horse)>
<!ATTLIST Contestant
  Clubname CDATA #REQUIRED
  Status (finished | walkover | disqualified | dropout) #REQUIRED
  Time CDATA #IMPLIED>
<!ELEMENT Rider EMPTY>
<!ATTLIST Rider
  Name CDATA #REQUIRED
  Weight CDATA #IMPLIED>
<!ELEMENT Horse EMPTY>
```

```
<!ATTLIST Horse
  Name CDATA #REQUIRED
  Weight CDATA #IMPLIED
  Birthyear CDATA #IMPLIED>
```

Now we can insert the DTD file into the DTD_REF table (which was created when we enabled the database for XML).

Execute the following INSERT statement, to insert the DTD file into the DTD_REF table of the database:

```
INSERT INTO db2xml.DTD_REF VALUES ('d:\xmltemp\race.dtd',
db2xml.XMLClobFromFile('d:\xmltemp\race.dtd'), 0, 'userX', 'userZ', 'userY')
```

The first value specifies a name for the inserted DTD file, this is also the primary key of the DTD_REF table. It is usual to set the fully qualified name of the file as this value.

The second value is the DTD file itself. This value has to be of XMLCLOB type, hence we use the XML extender's function XMLClobFromFile to import the DTD file into an XMLCLOB.

The third value (called USAGE_COUNT) shows how many DAD files refer to this DTD file. It has to always be set to 0 when a DTD file is first being inserted.

The rest of the parameters are optional and specify the following: AUTHOR, CREATOR, UPDATOR.

When a DTD file has been inserted into the DTD_REF table, it can be referenced by DAD files associated with XML columns or XML collections in the database in question.

Note that as with DAD files, if the DTD file has to be altered then it is not enough to change the file. The row for the old DTD has to first be removed from the DTD_REF table. If the DTD is in use then the XML column or XML collection that is using it has to first be disabled. It is always possible to see if a DTD in the DTD_REF table is in use by checking the usage_count value for a specific DTD.

We can now define the DAD file for the XML column. The DAD file will contain a reference to the DTD file and information about the side tables. It is not important to have side tables but we will use one side table to illustrate how this feature works. We will have a side table with two columns: Date and Distance.

The DAD file starts, as before, with the following lines:

```
<?xml version="1.0"?>
<!DOCTYPE DAD SYSTEM " C:\Program
Files\IBM\SQLLIB\samples\db2xml\dtd\dad.dtd">
<DAD>
```

Then we have an element called `dtdid`, where we define the DTD to be used to control incoming XML documents:

```
<dtdid>d:\xmltemp\race.dtd</dtdid>
```

Then the `validation` element, in this case we set the validation to YES. This activates the control of the incoming XML documents:

```
<validation>YES</validation>
```

Now we have the `Xcolumn` element:

```
<Xcolumn>
```

Within this element we can specify the side tables (in this case only one side table), and the mapping between elements or attributes and the columns of the side tables. In this way the side tables will be automatically updated every time a new XML document is inserted. Here is the content of the `Xcolumn` element:

A `table` element with a `name` attribute. That is the name of the side table. `<table name="race_st">`

A `column` element for each column of the side table. The `name` attribute indicates the name of the column, the `type` attribute indicates the data-type of the column, the `path` attribute indicates where in the XML document's structure to get the value from, the `multi_occurrence` attribute indicates whether or not the specified path can appear many times within an XML document. (Note that an empty element can be closed with a `"/"` in the end of the opening tag)

```
<column name="Racedate"  
        type="date"  
        path="/Race/@Date"  
        multi_occurrence="NO"/>
```

```
<column name="Racedistance"  
        type="integer"  
        path="/Race/@Distance"  
        multi_occurrence="NO"/>
```

And the closing tag of the table element. `</table>`

And of course the closing tags of the `Xcolumn` element and the `DAD` element:

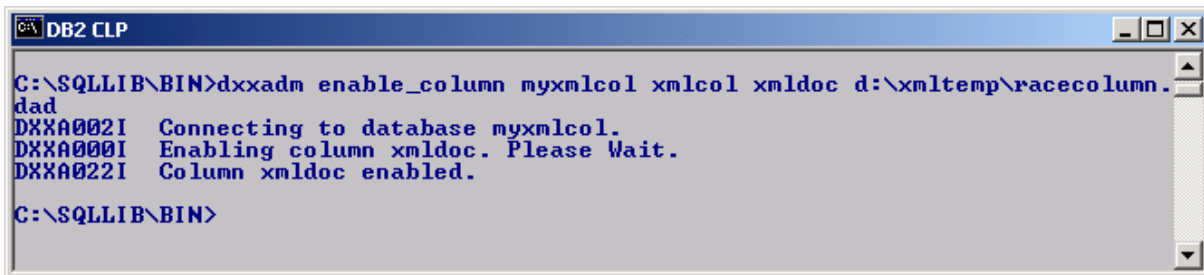
```
</Xcolumn>  
</DAD>
```

- Now save the DAD file (for example `d:\xmltemp\racecolumn.dad`)!

- Enable the XML column! Here is the command:

```
dxxadm enable_column myxmlcol xmlcol xmldoc d:\xmltemp\racecolumn.dad
```

where myxmlcol is the database name, xmlcol is the name of the table and xmldoc is the name of the column in the table.



```
DB2 CLP
C:\SQLLIB\BIN>dxxadm enable_column myxmlcol xmlcol xmldoc d:\xmltemp\racecolumn.dad
DXXA002I  Connecting to database myxmlcol.
DXXA000I  Enabling column xmldoc. Please Wait.
DXXA022I  Column xmldoc enabled.
C:\SQLLIB\BIN>
```

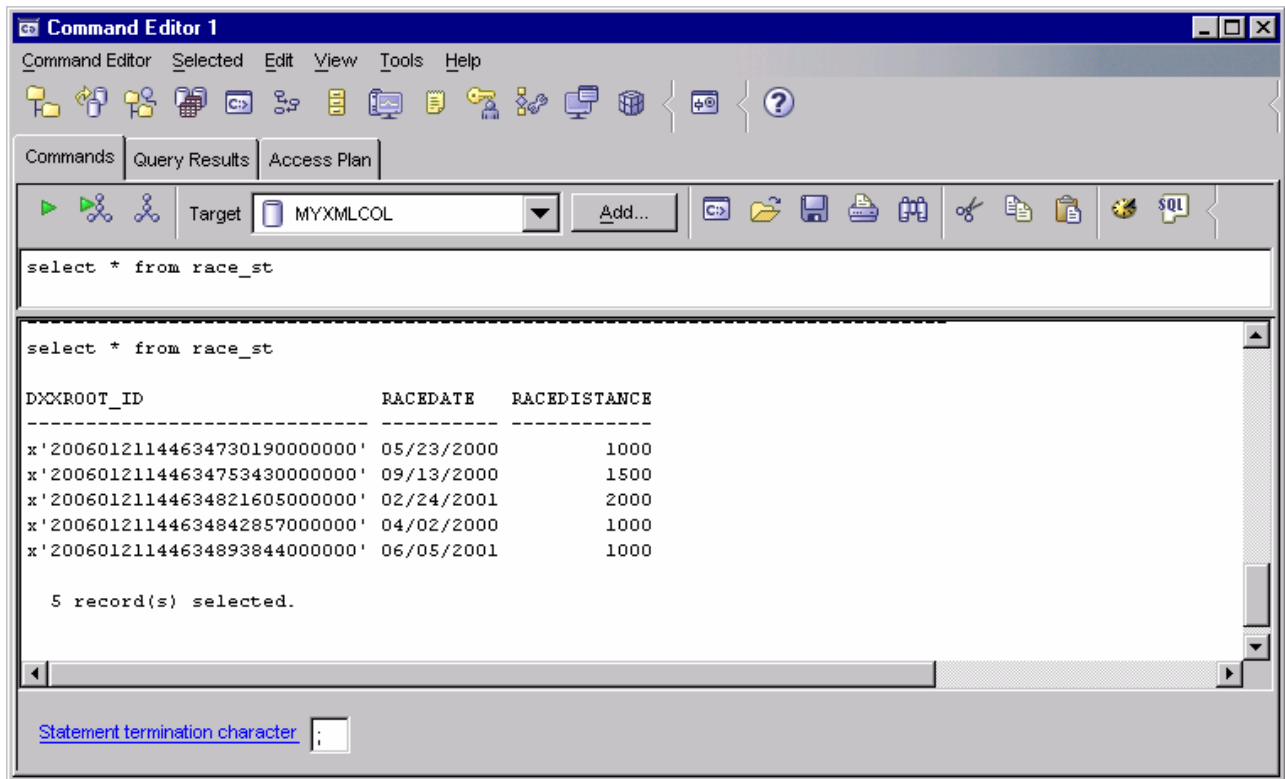
Now that the XML column has been enabled, we can insert XML documents into it. To insert an XML document we can execute an INSERT statement. When inserting an XML document into a column of a table, we must always think of the data type of the column. The column, to which we will insert the XML documents is of the following type: DB2XML.XMLVARCHAR. Fortunately, there is a set of functions for transforming XML documents to and from all the different XML data types. One of those functions is this: DB2XML.XMLVarcharFromFile(). This function takes one argument: the full filename as a string and returns the content of that file (the XML document) as a DB2XML.XMLVARCHAR. Here is an example of an INSERT statement:

```
INSERT INTO xmlcol (xmldoc) VALUES
(DB2XML.XMLVarcharFromFile('d:\xmltemp\my20000603132654013484000000.xml'))
```

The file d:\xmltemp\my20000603132654013484000000.xml is just one of the files we generated before (see section 5.3). The filenames are random, so the files that you have, have different filenames from the filenames that appear in section 5.3.

When the XML document has been inserted into the database, the side tables have also been updated. In our case there should be one new record in the `race_st` table.

After inserting all five XML documents in the XML column the content on the side table is following:



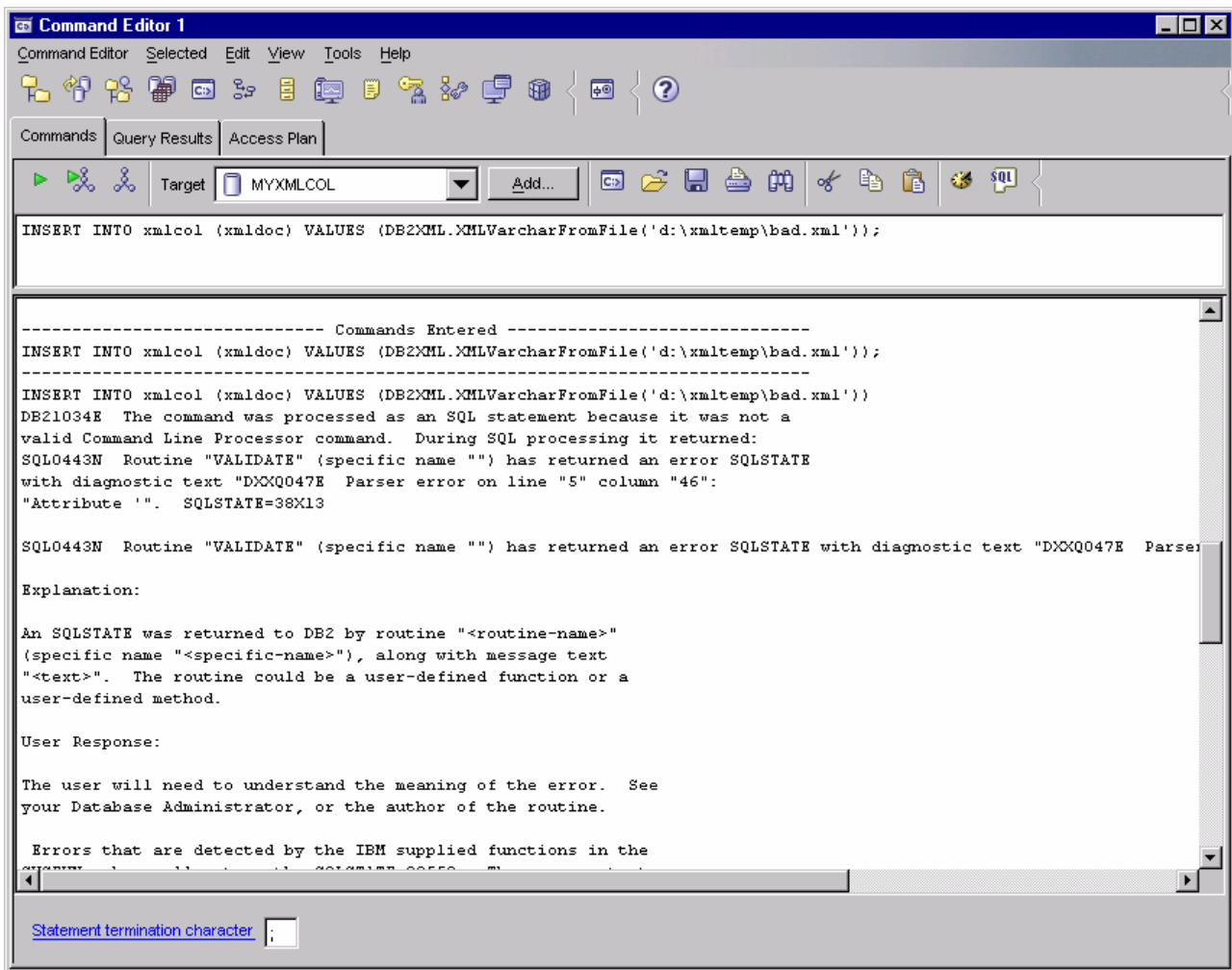
The screenshot shows a 'Command Editor 1' window with a menu bar (Command Editor, Selected, Edit, View, Tools, Help) and a toolbar. Below the toolbar are tabs for 'Commands', 'Query Results', and 'Access Plan'. The 'Query Results' tab is active, showing a table with 5 records. The table has columns 'DXXROOT_ID', 'RACEDATE', and 'RACEDISTANCE'. The data is as follows:

DXXROOT_ID	RACEDATE	RACEDISTANCE
x'200601211446347301900000000'	05/23/2000	1000
x'200601211446347534300000000'	09/13/2000	1500
x'200601211446348216050000000'	02/24/2001	2000
x'200601211446348428570000000'	04/02/2000	1000
x'200601211446348938440000000'	06/05/2001	1000

Below the table, it says '5 record(s) selected.' At the bottom, there is a 'Statement termination character' field with a semicolon (;) entered.

If the XML document does not comply with the DTD file, specified in the DAD file, then it will be rejected. That can easily be tested; try to insert an XML document with the wrong type of elements or attributes.

Here is what happened when an invalid XML document was inserted into the XML column:



The screenshot shows a 'Command Editor 1' window with a menu bar (Command Editor, Selected, Edit, View, Tools, Help) and a toolbar. Below the toolbar is a 'Commands' tab with a 'Query Results' sub-tab. The 'Target' dropdown is set to 'MYXMLCOL'. The main text area contains the following SQL command and error message:

```
INSERT INTO xmlcol (xmldoc) VALUES (DB2XML.XMLVarcharFromFile('d:\xmltemp\bad.xml'));
```

----- Commands Entered -----
INSERT INTO xmlcol (xmldoc) VALUES (DB2XML.XMLVarcharFromFile('d:\xmltemp\bad.xml'));

DB21034E The command was processed as an SQL statement because it was not a
valid Command Line Processor command. During SQL processing it returned:
SQL0443N Routine "VALIDATE" (specific name "") has returned an error SQLSTATE
with diagnostic text "DXXQ047E Parser error on line "5" column "46":
"Attribute ". SQLSTATE=38X13

SQL0443N Routine "VALIDATE" (specific name "") has returned an error SQLSTATE with diagnostic text "DXXQ047E Parser

Explanation:

An SQLSTATE was returned to DB2 by routine "<routine-name>"
(specific name "<specific-name>"), along with message text
"<text>". The routine could be a user-defined function or a
user-defined method.

User Response:

The user will need to understand the meaning of the error. See
your Database Administrator, or the author of the routine.

Errors that are detected by the IBM supplied functions in the

Statement termination character: ;

An XML document is rejected when:

- The element structure is not as specified in the DTD file
- The attributes of the elements are not following the rules of the DTD file
- The SYSTEM of the XML document (specified in the DOCTYPE element) is not the same as the one in the DAD file. Both should point to the same DTD file.

On the other hand an XML document that is defined as standalone (in the XML declaration) can be accepted if it does not break any of the rules above.

The XML column can now be queried in the way we saw in section 4.2.

6 Internet Resources

XML & DTD Tutorials

<http://L238.dsv.su.se:86/tutorial/>

<http://www.w3schools.com/xml/default.asp>

<http://www.spiderpro.com/bu/buxxmlm001.html>

<http://msdn.microsoft.com/library/en-us/xmlsdk/html/79c78508-c9d0-423a-a00f-672e855de401.asp>

DB2 XML extender

<http://www-4.ibm.com/software/data/db2/extenders/xmlxt/>

XML/SQL

http://docs.openlinksw.com/virtuoso/fn_XMLELEMENT.html

7 Epilogue

When all this is done, you should have quite a good understanding of how to use DB2 to manage XML documents and XML data.

I hope you have enjoyed this compendium. Please give me feedback!

The Author

nikos dimitrakas