Introduction to DB2 & XML

Version 1.0
July 2011

nikos dimitrakas

Table of contents

1 Introduction

This compendium gives a short introduction to DB2 9.7 and its facilities for database administration. We discuss installing DB2 9.7 and using the Command Editor and the Control Center. After that, there is an introduction to some DB2 specific XML features accompanied by SQL/XML features supported by DB2. All the examples are tested on DB2 for Windows on a Windows 7 64-bit platform, but they should work in a similar manner on any platform. It is recommended that you use DB2 for Windows.

The latest version of this compendium is available at http://coursematerial.nikosdimitrakas.com/db2xml/ where all other relevant files can also be found.

1.1 DB2

DB2 is IBM's relational DBMS and it was one of the first such products. Since version 5 or 6 IBM has included XML features in DB2. During the past decade many of those features have become part of the SQL standard, while others have been replaced by similar standard facilities. DB2 9 has abandoned several DB2 specific XML solutions available in the previous versions and is moving closer to the SQL standard. DB2 9 is still missing several XML features that are part of the SQL standard.

DB2 has a set of tools for working with DB2 databases and for managing DB2 installations. The Control Center serves as the main hub for performing almost everything. It has several wizards for practically every command. The Command Editor is a tool for executing SQL commands and scripts and completes the Control Center. There are several other tools bundled with DB2, but they are not relevant to this introduction.

1.2 Prerequisites

It is required that the reader is familiar with database administration and SQL and has a good understanding of XML. This introduction focuses on DB2 specific XML features, so most basic database concepts will not be explained in detail. All the examples can be executed in any interface tool for DB2 but the recommended tool is the Command Editor (which is bundled with DB2).

1.3 Structure

In the next chapter we will take a quick look at the installation and configuration of DB2 and at the Control Center and the Command Editor. After that we will look at the sample data used in the examples to come. In chapter 4 we will go through several examples using the sample data and DB2's XML features.

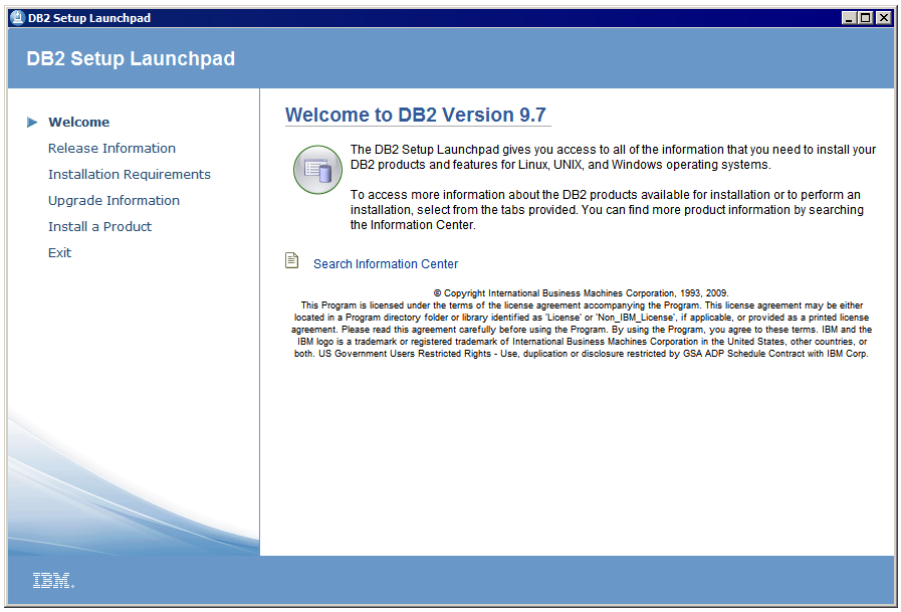
2 DB2 9.7

DB2 9.7 is available as a free trial by IBM. The Express-C Edition is free, but it has limited functionality. In this introduction, we will use the Enterprise Server Edition (referred to even as Data Server Edition).

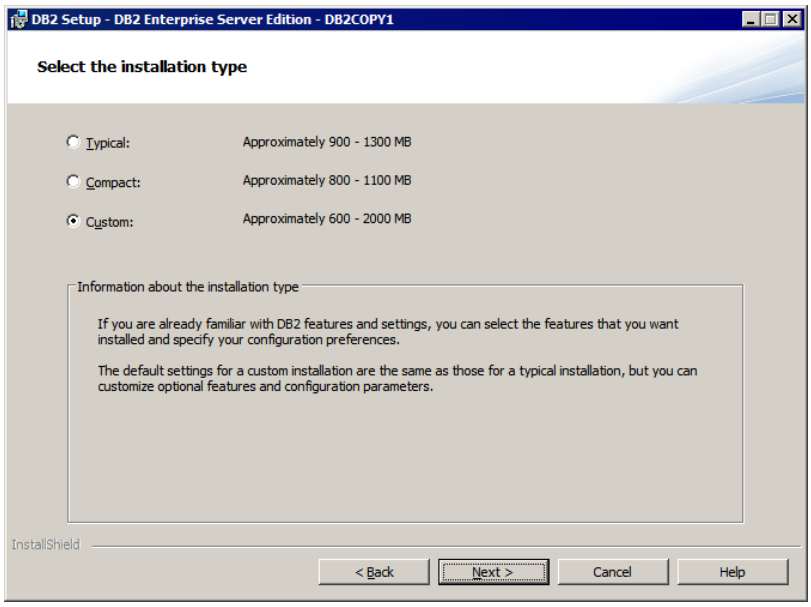
2.1 Installation

Start by downloading the appropriate installation file. This compendium is based on version 9.7 for Windows x64. In order to download the installation file, you may need to create a free account.

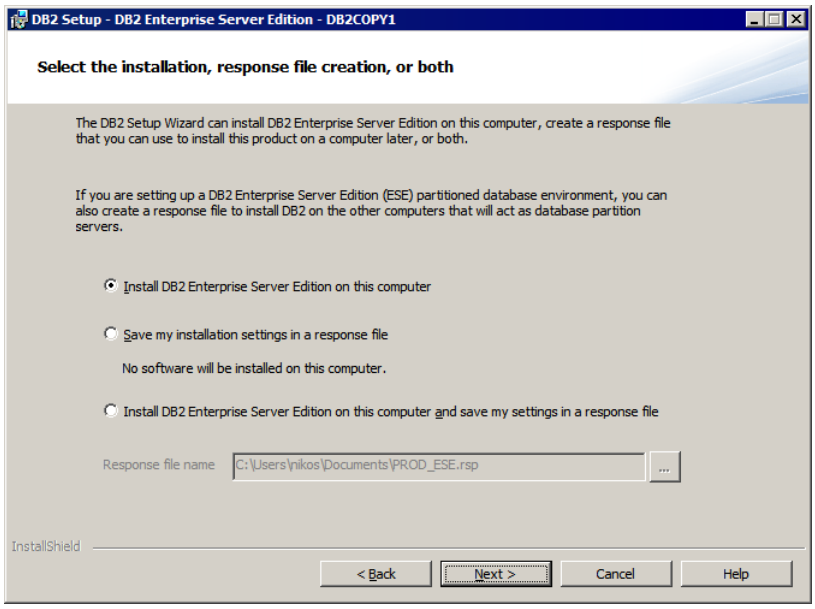
Run the executable to start the installation. You may need to unzip the downloaded file once or twice first. Eventually, the DB2 Setup Launchpad should appear:



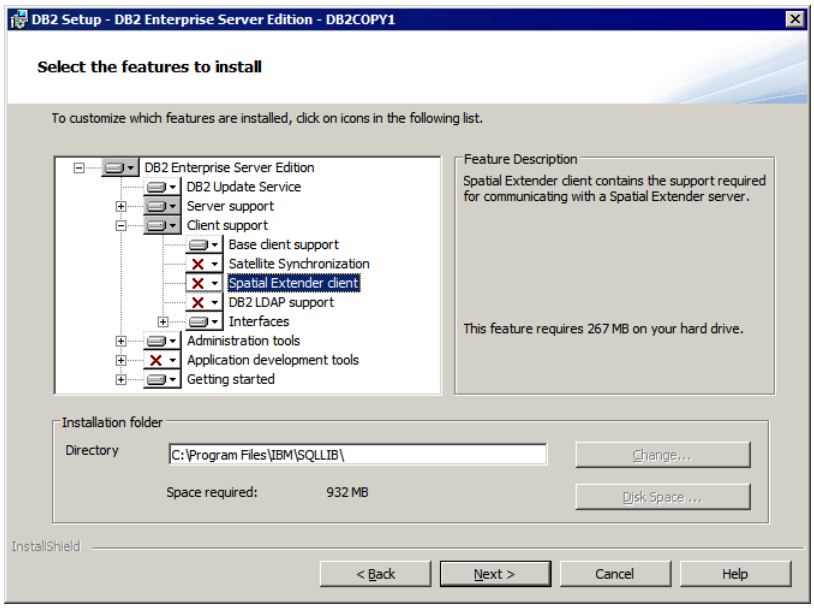
Select "Install a Product" from the menu and install a "DB2 Enterprise Server Edition Version 9.7". The installation wizard will appear which will guide you through the installation and configuration. Press "Next", accept the License Agreement and press "Next" once more. Choose "Custom" as the installation type:



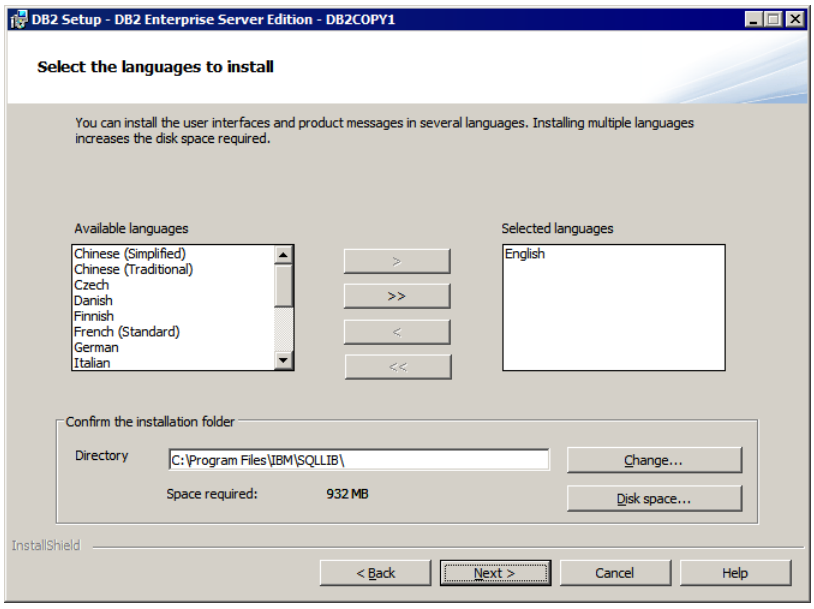
In the next step, choose to install DB2 on this computer:



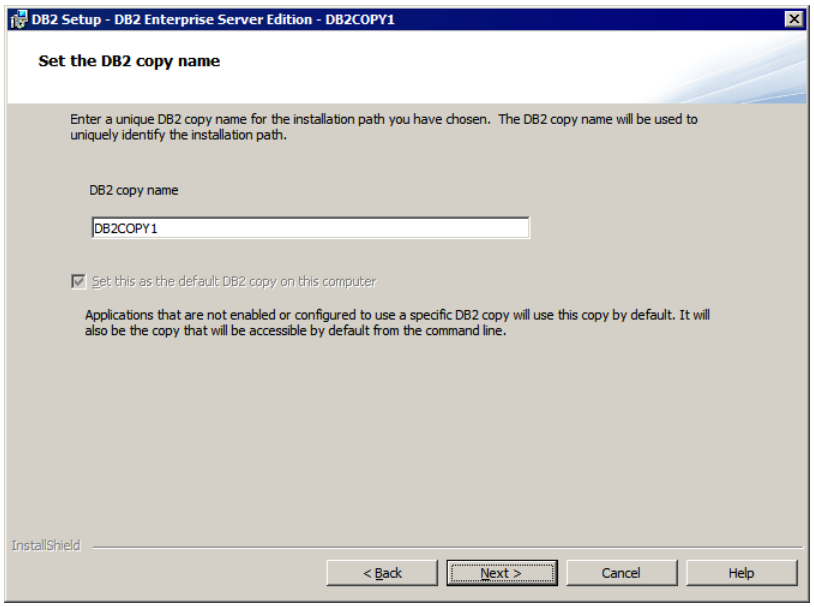
In the next step, you may add or remove features. For example, the Application development tools, LDAP support, and spatial extender are not relevant to this introduction and can be excluded, which saves some space.



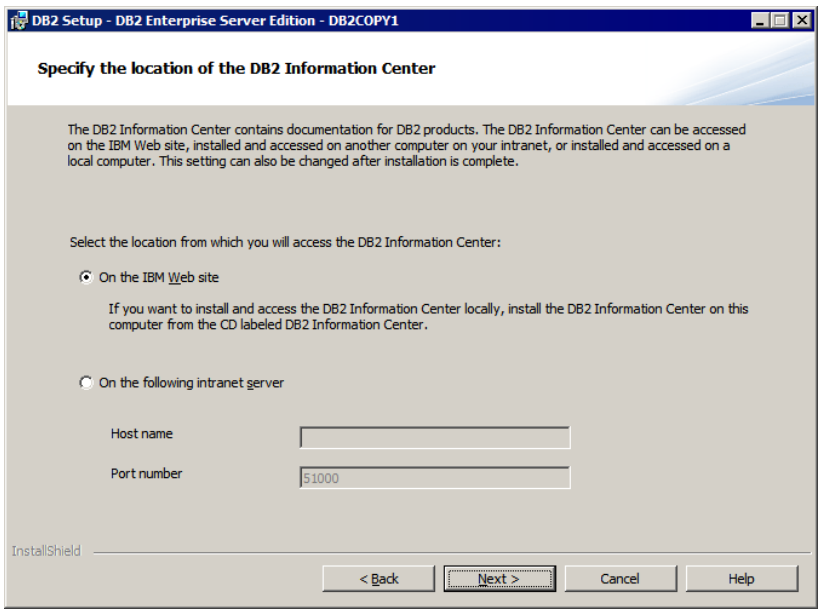
In the next step you can choose to install additional interface languages. We are fine with just English:



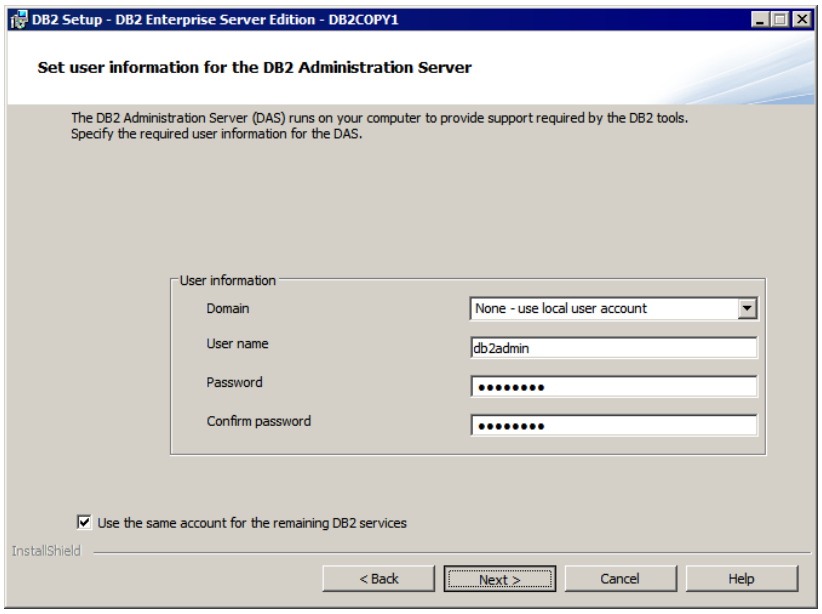
In the next step you can name your installation. The default is DB2COPY1, which is fine.



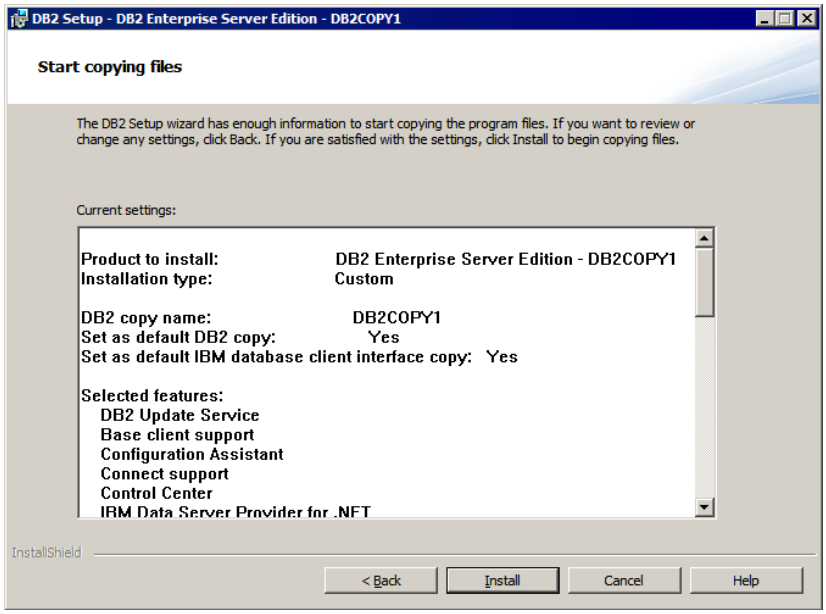
In the next step you can select how you want to access the Information Center, which is the DB2 manual. The IBM web site option is fine.



In the next step you must provide a password for the db2admin account. This will be the administrator account for DB2 and it will be used for running the DB2 services on the system.

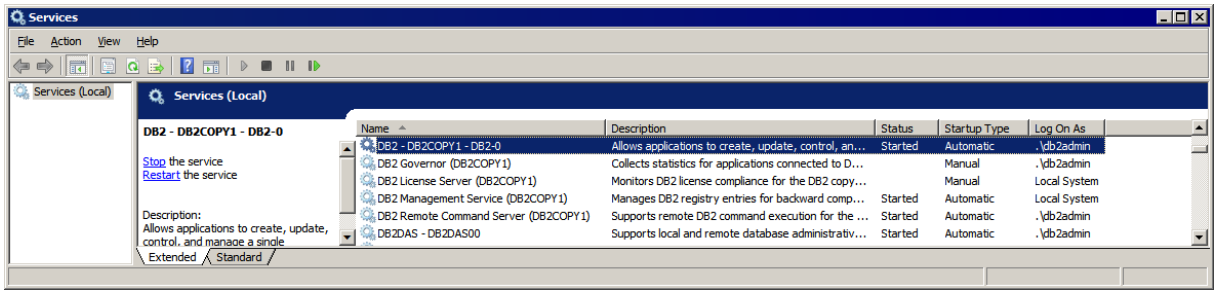


In the next step, you see that a new DB2 instance will be created. You may configure the instance, so that it for example does not start automatically every time Windows starts, but then you will have to manually start it every time you want to use it. Leave the default settings and go to the next step. You may choose to "Prepare the DB2 tools catalog", but this is not necessary. Move on to the next step. Deactivate notifications and move on. If the operating system security is enabled, then Windows users must belong to the right group in order to access the local database instance and databases. By disabling this option, all local accounts will have full access to the database instance. We disable it, but you should choose the appropriate option for your system. If you enable this option, make sure to make your account a member of DB2ADMNS in order to get full access. After this step, the wizard will present a summary, before the installation is completed:

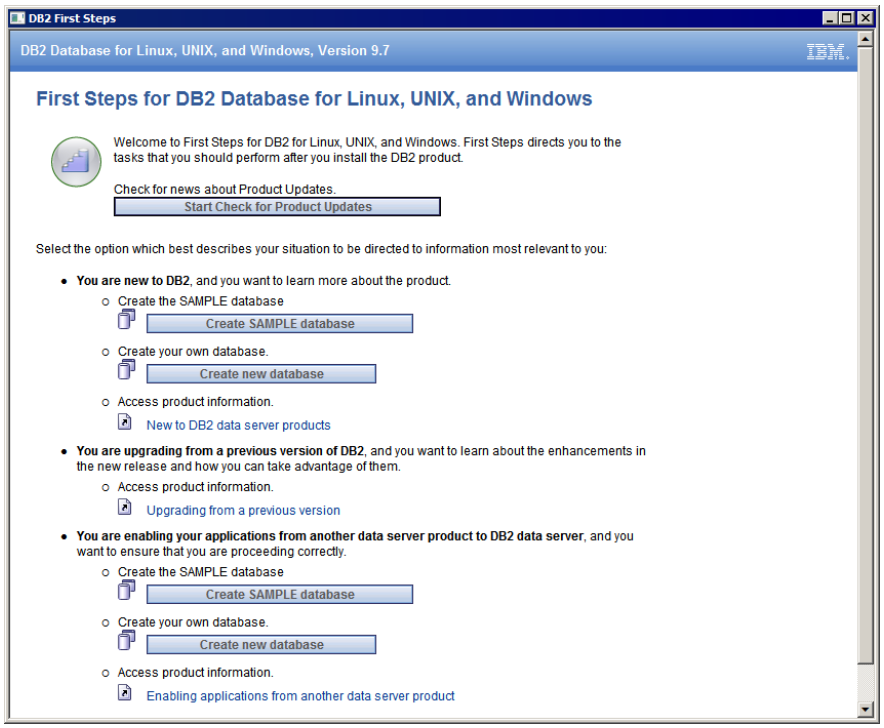


After pressing "Install" you may have to wait for a while. The installation will take a few minutes.

During the installation several Windows services were created and they use the db2admin account, which was also created in the system:



After the installation, the "DB2 First Steps" window will appear.

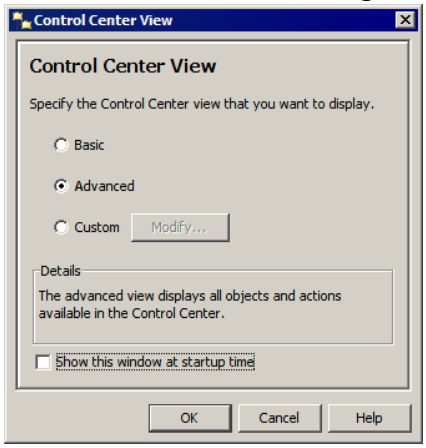


We will instead use the Control Center and the Command Editor for performing tasks like creating databases.

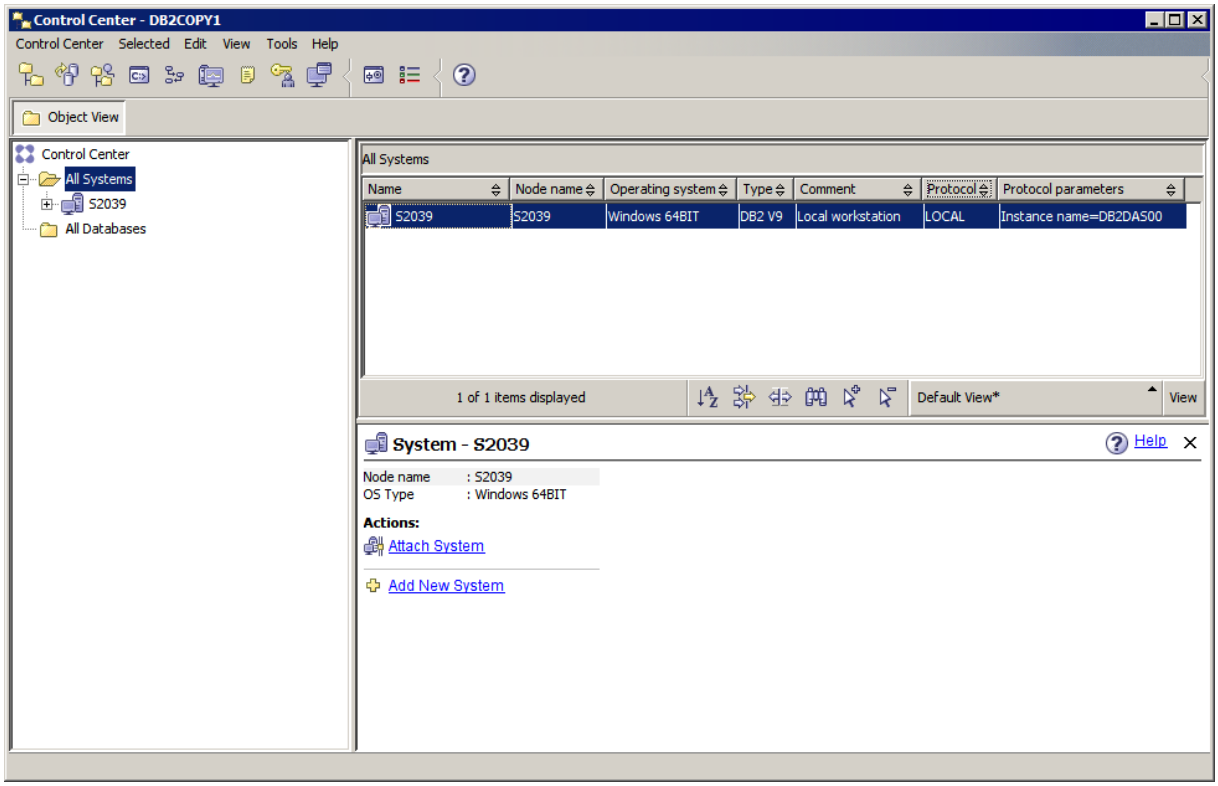
2.2 Control Center

The Control Center is a graphical client that allows us to display systems, databases, and database objects and perform administration tasks on them. Most of the tasks performed here can be achieved with DB2 commands or SQL commands, but the Control Center offers user-friendly wizards with instructions and explanations. This is an efficient way to work, but from an educational perspective, it is better to write the commands instead of letting a wizard create and execute commands automatically.

When you start the Control Center (from the start menu) for the first time, you may be asked to select a view configuration. We use the Advanced view:



The Control Center has a tree structure on the left that presents all the systems, instances, databases, database objects, etc. On the right, details about the selected object are displayed.



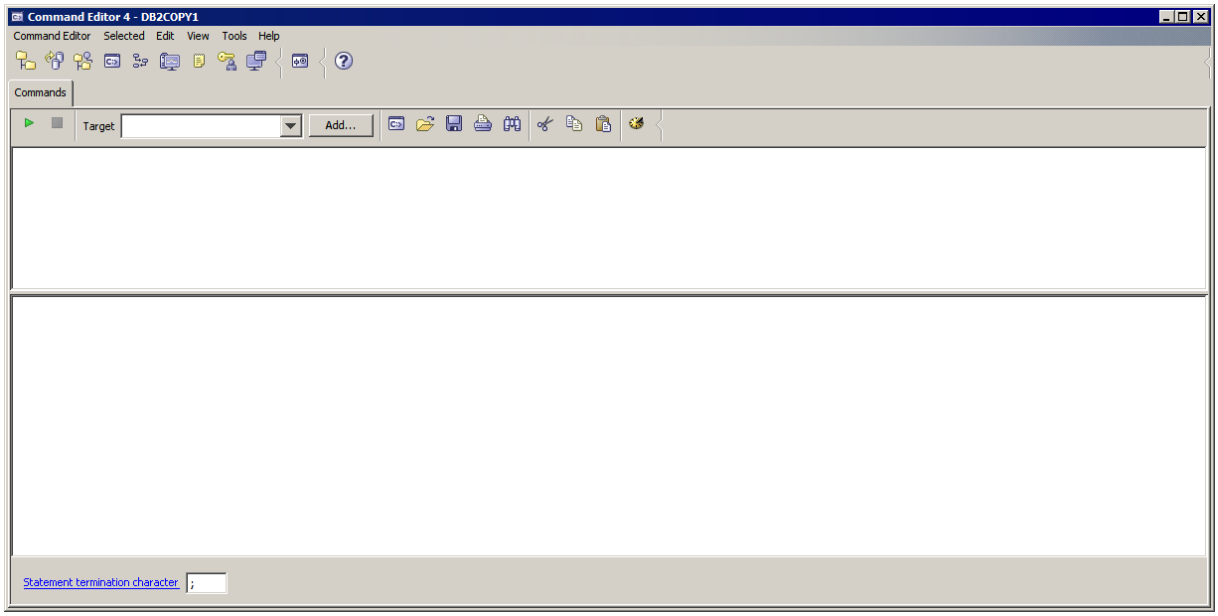
By right-clicking on any object (either in the tree structure or on the right pane), we are presented with all the possible tasks associated with the selected object. We can, for example, click on "All Databases" and select "Create New Database", which will launch the "Create Database Wizard". Once a database is in place, wizards will become available for creating tables, views, triggers, etc.

A very useful feature of all the wizards of the Control Center is that they can display the generated command on the wizard's final step. Thus, we can copy the generated command and store it or append it to a script. In this way we do not become dependent on the wizards, but can take advantage of them.

2.3 Command Editor

The Command Editor is a graphical client with which we can execute DB2 commands, SQL statements and work with command scripts. This is the recommended tool for working with SQL.

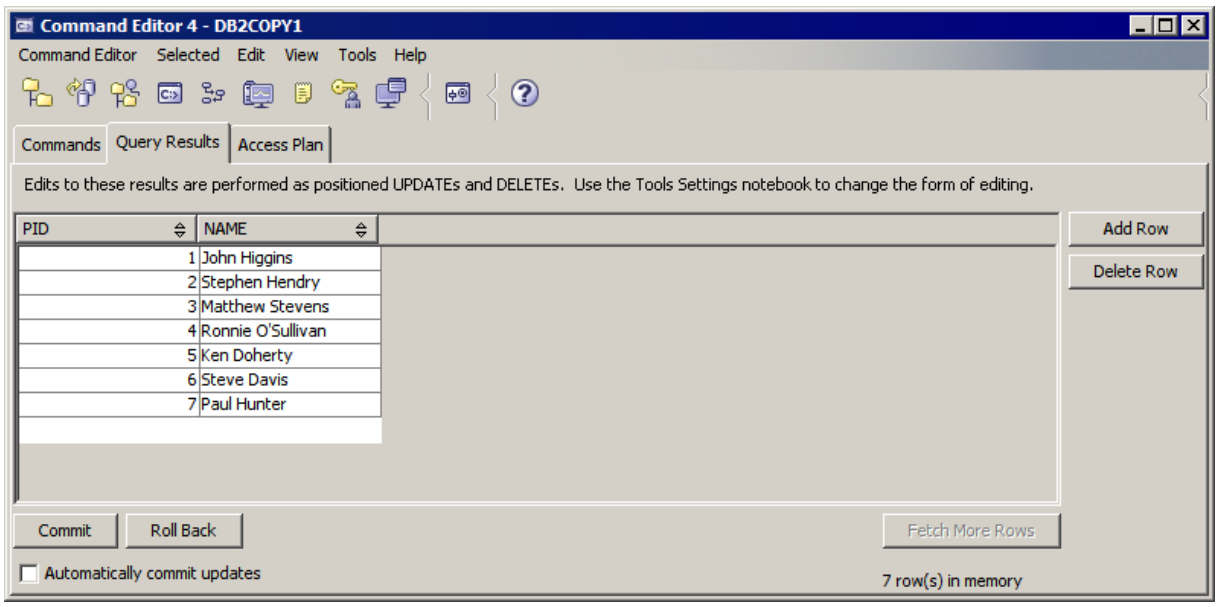
A Command Editor window can be opened from the start menu or from within the Control Center. Several Command Editor windows can be open at the same time and each window has its own database connection. A Command Editor window looks like this:



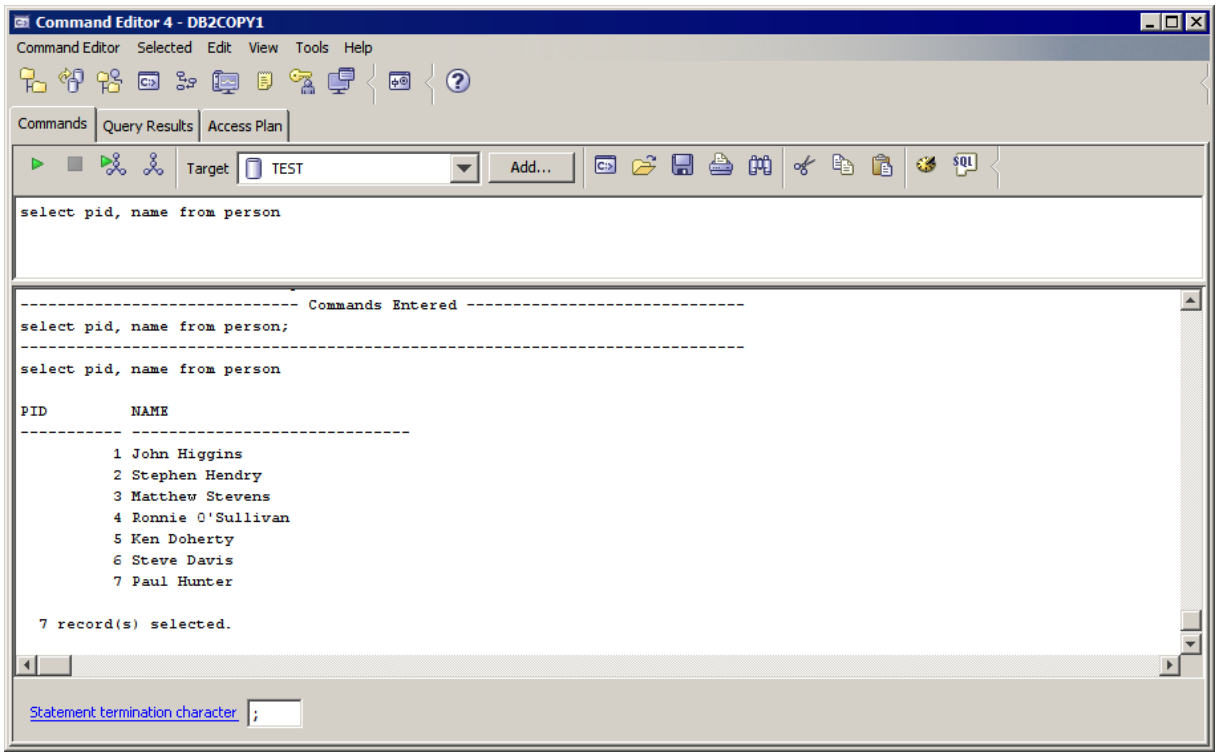
The top part is for input and the bottom part is for output. The green "play" button executes the command or script entered as input. If a part of the input is selected, then only the selection is executed. Control-Enter can also be used to execute the (selected) input. The "Target" is the database that is currently connected. To connect to a database, select it in the drop down box next to the label "Target" or use the command "CONNECT TO". Once connected, any command executed will be in the context of the connected database.

Running a script implies that several commands will be entered and executed at once. The Command Editor must know what character indicates that one command is over and a new one is beginning. The "Command termination character" setting can be used to define the character used. The default is a semi-colon (;) but in some cases it may be necessary to switch to some other character. If no character is specified, each line will be interpreted as a separate command. Many more configurations of the Command Editor can be made in the "Tools Settings".

If the executed command is a single SELECT statement, then the result will be displayed by default in the "Query Results" tab:

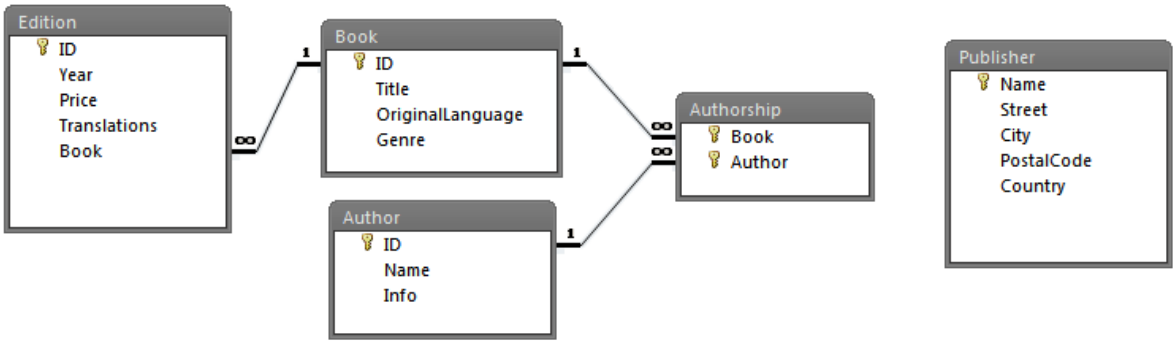


This behaviour can be configured in the "Tools Settings" so that the result is instead displayed in the output area of the "Commands" tab:



3 Sample Data

In this chapter we will take a look at the data that we will use in the examples to follow. We will use a database with both relational data and XML data. That is, a database with tables, columns, keys, integrity constraints, etc. but with a couple of columns containing XML documents (each cell being an XML document).



The columns Edition.Translations and Author.Info contain XML according to the following XML Schemas. The rest of the columns are defined as VARCHAR and INTEGER. The only column that allows NULL is the column Book.Genre.

XML Schema for documents in Edition.Translations:

```
<?xml version="1.0"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema">
   <element name="Translations">
      <complexType>
         <sequence>
            <element name="Translation" minOccurs="0" maxOccurs="unbounded">
               <complexType>
                  <attribute name="Language" type="string" use="required"/>
                  <attribute name="Publisher" type="string" default="N/A"/>
                  <attribute name="Price" type="integer" use="required"/>
               </complexType>
            </element>
         </sequence>
      </complexType>
   </element>
</schema>
```

The value of the attribute Publisher must correspond to a value in the column Publisher.Name. This kind of constraint could be implemented as a set of triggers.

XML Schema for documents in Author.Info:

```
<?xml version="1.0"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema">
   <element name="Info" type="InfoType"/>
   <complexType name="InfoType">
      <all>
         <element name="Email" type="string"/>
         <element name="YearOfBirth" type="integer"/>
         <element name="Country" type="string"/>
      </all>
   </complexType>
</schema>
```

The entire script for creating and populating the database can be found on http://coursematerial.nikosdimitrakas.com/db2xml/

The script can be run in the Command Editor. It creates a database called bookdb.

3.1 XML data type

In DB2 9.7 there is a special data type for XML. The data type itself is called XML. There is no support for associating a DTD or XML Schema with a column of this data type directly. Instead, validation of XML values must be done manually with the function XMLVALIDATE and it supports only XML Schema. Any schema to be used for validation must be registered in the schema repository in advance. In order to avoid invalid XML documents from being stored in a column, a constraint "IS VALIDATED" can be created. Any attempt to put invalid data in the column will result in a constraint violation error.

In the provided database script, there is no validation. The XML data type does on the other hand always check that the input is well-formed.

4 Examples

In this chapter we will go through some examples of SQL/XML in DB2 and some examples that use DB2 specific XML features. All the examples in this chapter assume that the database has been created and that the Command Editor window used is connected to it.

4.1 XMLELEMENT, XMLFOREST, XMLATTRIBUTES

Let's start off with a few simple queries using some basic SQL/XML publishing functions. We want to create an XML document for each author. The root element shall be "Author", the name shall be an attribute and the author info (which is already an XML document) shall be the content. The following SQL statement does that.

```
SELECT XMLELEMENT(NAME "Author", XMLATTRIBUTES(name AS "Namn"), info)
FROM author
```

Here is a portion of the result (2 rows):

```
<Author Namn="John Craft">
   <Info>
      <Email>jc@jc.com</Email>
      <Country>England</Country>
      <YearOfBirth>1948</YearOfBirth>
   </Info>
</Author>
<Author Namn="Arnie Bastoft">
   <Info>
      <Email>bastoft@frei.at</Email>
      <Country>Austria</Country>
      <YearOfBirth>1971</YearOfBirth>
   </Info>
</Author>
```

If we would like to create an XML document for each publisher, it may be better to use XMLFOREST since the table publisher has many columns that we may want to have as elements. Let's assume that for each publisher, we want to have a root element "Publisher" and that all the columns should get their own elements. The following statement does that.

```
SELECT XMLELEMENT(NAME "Publisher", XMLFOREST(name AS "Name", street AS "Street",
city AS "City", postalcode AS "PostalCode", country AS "Country"))
FROM publisher
```

For each row in the table publisher, we get an XML document like this:

```
<Publisher>
   <Name>ABC International</Name><Street>7th Bear St.</Street><City>Berlin</City>
   <PostalCode>44500</PostalCode><Country>Germany</Country>
</Publisher>
```

One thing that is important when working with XML is the case of the element names and attribute names. In the above examples, we used the double quotes in order to enforce the desired case. DB2's default is to capitalize column names when generating XML. So the following statement would capitalize everything except for "City":

```
SELECT XMLELEMENT(NAME Publisher, XMLFOREST(name, street AS StrEEt, city AS "City"))
FROM publisher
```

The result looks like this:

```
<PUBLISHER>
   <NAME>ABC International</NAME><STREET>7th Bear St.</STREET><City>Berlin</City>
</PUBLISHER>
```

4.2 XMLAGG

XMLAGG is an aggregate function and as such, it complies with the rules of aggregate functions. If it is used without a GROUP BY clause, then all the rows will become one group. It can, of course, be mixed with non-aggregated columns in the SELECT clause, but then all non-aggregated columns must also appear in the GROUP BY clause.

If we want to expand on the example from the previous section and put all the authors in one XML document, we need to use XMLAGG. Any column that appears inside the XMLAGG function is considered to be aggregated. The following statement creates a root element "Authors" and aggregates all the Author elements into it.

```
SELECT XMLELEMENT(NAME "Authors",
                  XMLAGG(XMLELEMENT(NAME "Author",
                                    XMLATTRIBUTES(name AS "Namn"),
                                    info)))
FROM author
```

The result looks like this:

```
<Authors>
   <Author Namn="John Craft"><Info><Email>jc@jc.com</Email>
   <Country>England</Country><YearOfBirth>1948</YearOfBirth></Info></Author>
   <Author Namn="Arnie Bastoft"><Info><Email>bastoft@frei.at</Email>
   <Country>Austria</Country><YearOfBirth>1971</YearOfBirth></Info></Author>
   <Author Namn="Meg Gilmand"><Info><Email>megil@archeo.org</Email>
   <Country>Australia</Country><YearOfBirth>1968</YearOfBirth></Info></Author>
   …
</Authors>
```

XMLAGG in combination with GROUP BY is relevant when we need some nesting. Perhaps we want to group the publishers per country. The result shall be one Country element per country containing one or more Publisher elements. If we want to also have a root element, a second XMLAGG is required.

```
SELECT XMLELEMENT(NAME "PublishersByCountry", XMLAGG(countryxml))
FROM (SELECT XMLELEMENT(NAME "Country",
                  XMLATTRIBUTES(country AS "Name"),
                  XMLAGG(XMLELEMENT(NAME "Publisher",
                     XMLATTRIBUTES(name AS "Name", city AS "City")))) AS countryxml
      FROM publisher
      GROUP BY country) innertable
```

The nested statement produces one Country element for each country. The result is a table with as many rows as there were countries (groups). The outer statement aggregates these Country elements and makes them the content of the element PublishersByCountry. In the nested statement the column country is the only one appearing in the SELECT clause outside the aggregate function, and is thus the only column appearing in the GROUP BY clause. The

result of the nested statement is a table with the alias innertable and it has a column named countryxml. The result of the entire statement has the following structure:

```
<PublishersByCountry>
   <Country Name="England">
      <Publisher Name="Benton Inc" City="London"/>
   </Country>
   <Country Name="Sweden">
      <Publisher Name="Bästa Bok" City="Stockholm"/>
      <Publisher Name="KLC" City="Uppsala"/>
      <Publisher Name="SCB" City="Stockholm"/>
   </Country>
   …
</PublishersByCountry>
```

4.3 XMLQUERY

The XMLQUERY function can be used when we want to execute XQuery within an SQL statement. The XMLQUERY function can also accept parameters that map values of the SQL scope to variables in the XQuery scope. We may want to retrieve the name and country of each author:

```
SELECT name, XMLQUERY('$i//Country/text()' PASSING info AS "i")
FROM Author
```

In this case the XQuery expression was quite a simple one, but it can also be complicated. The PASSING keyword allows us to map the current value of the column info as an XQuery variable (in this case "i" which is then referred to as "$i"). The result has two columns:

```
John Craft          England
Arnie Bastoft       Austria
Meg Gilmand         Australia
Chris Ryan          France
Marty Faust         USA
…
```

The result of the XMLQUERY function is actually of the XML data type, but DB2 will serialize it automatically when showing the result. Here is another example that illustrates that the XMLQUERY function returns XML:

```
SELECT name,
      XMLQUERY('$x/text()' PASSING XMLQUERY('$i//Country' PASSING info AS "i") AS "x")
FROM Author
```

This produces the same result as the previous statement, but finds the country in two steps.

XMLQUERY can also be used to create XML from a string. So XMLQUERY('<X>123</X>') will return an XML value. This is because the string '<X>123</X>' is a valid XQuery statement.

4.4 XMLTABLE

When dealing with repeating elements in an XML document, we may want to break it down into smaller XML-documents or even values. The XMLTABLE function can be used in the FROM clause of a SELECT statement and it transforms the result of an XQuery statement into a table. We may want to get one row per translation of each edition. The column translations in the table edition contains multiple Translation elements. So the following statement splits them up and presents them one by one.

```
SELECT id, book,  tt.*
FROM Edition, XMLTABLE('$t//Translation' PASSING translations AS "t") AS tt
```

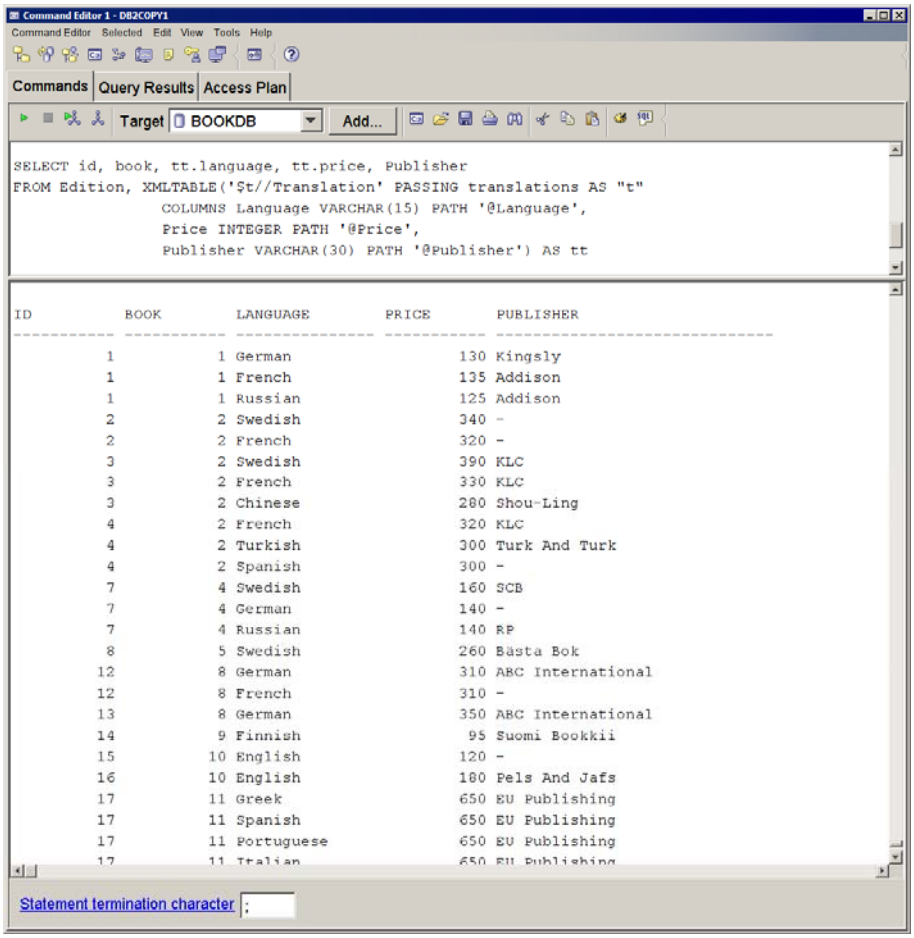
The result should look like this:

```
1     1     <Translation Language="German" Publisher="Kingsly" Price="130"/>
1     1     <Translation Language="French" Publisher="Addison" Price="135"/>
1     1     <Translation Language="Russian" Publisher="Addison" Price="125"/>
2     2     <Translation Language="Swedish" Price="340"/>
2     2     <Translation Language="French" Price="320"/>
…
```

The keyword COLUMNS can break this down further:

```
SELECT id, book, tt.language, tt.price, tt.publisher
FROM Edition, XMLTABLE('$t//Translation'
                    PASSING translations AS "t"
                    COLUMNS Language VARCHAR(15) PATH '@Language',
                              Price INTEGER PATH '@Price',
                              Publisher VARCHAR(30) PATH '@Publisher') AS tt
```

The translations XML is now fully shredded:



4.5 XMLEXISTS

XMLEXISTS is a function that can be used to express conditions based on the existence of a particular XML node. We could for example find any books that have been translated to German (i.e. they have an edition with a translation whose language is German):

```
SELECT title
FROM Book
WHERE id IN (SELECT book
             FROM edition
             WHERE XMLEXISTS('$t//Translation[@Language="German"]'
                             PASSING translations AS "t"))
```

The nested statement does the work of finding the correct books, while the outer statement retrieves the titles. As you can see, the result of the function is a boolean value, so it can be used as a condition. The result looks like this:

```
Misty Nights
Contact
Music Now and Before
Musical Instruments
Oceans on Earth
Le chateau de mon pere
```

4.6 XMLROW

XMLROW is a DB2 specific function that creates one XML element per row in the result of a SELECT statement. The same result can of course be achieved by SQL/XML publishing functions. Here are some examples with XMLROW.

If we want an XML document per publisher we could use the following:

```
SELECT XMLROW(name, street, city)
FROM publisher
```

The default result is always one "row" root element and one element per column:

```
<row><NAME>ABC International</NAME><STREET>7th Bear St.</STREET><CITY>Berlin</CITY></row>
<row><NAME>Addison</NAME><STREET>2nd Monet St.</STREET><CITY>Toulouse</CITY></row>
<row><NAME>Aurora Publ.</NAME><STREET>3rd Uffizi Rd.</STREET><CITY>Florence</CITY></row>
…
```

XMLROW allows for some configuration. We can for example rename the root element or ask for attributes instead of elements for all the columns and even define the attribute/element names:

```
SELECT XMLROW(name AS "Name", street AS "Street", city AS "Town"
                    OPTION ROW "Publisher" AS ATTRIBUTES)
FROM publisher
```

This will still give one row per publisher in the result, but each row is an element according to our design:

```
<Publisher Name="ABC International" Street="7th Bear St." Town="Berlin"/>
<Publisher Name="Addison" Street="2nd Monet St." Town="Toulouse"/>
…
```

We can of course combine this function with XMLAGG and XMLELEMENT in order to create a root element Publishers.

```
SELECT XMLELEMENT(NAME "Publishers",
                XMLAGG(XMLROW(name AS "Name", street AS "Street", city AS "Town"
                                OPTION ROW "Publisher" AS ATTRIBUTES)))
FROM publisher
```

This will result in one XML document, instead of one per row:

```
<Publishers>
   <Publisher Name="ABC International" Street="7th Bear St." Town="Berlin"/>
   <Publisher Name="Addison" Street="2nd Monet St." Town="Toulouse"/>
   …
</Publishers>
```

4.7 XMLGROUP

XMLGROUP is another DB2 specific function. It is an aggregate function and creates an XML document as the result of a SELECT statement (or one per group if there is a GROUP BY clause). The default result is one "root" element per group with one "row" element per row and one subelement per column. We can configure the output in the same manner as we did with XMLROW and even rename the root element. Here is the same example as before, but with XMLGROUP instead:

```
SELECT XMLGROUP(name AS "Name", street AS "Street", city AS "Town"
                    OPTION ROOT "Publishers" ROW "Publisher" AS ATTRIBUTES)
FROM publisher
```

And the result is identical to that of previous example:

```
<Publishers>
   <Publisher Name="ABC International" Street="7th Bear St." Town="Berlin"/>
   <Publisher Name="Addison" Street="2nd Monet St." Town="Toulouse"/>
   <Publisher Name="Aurora Publ." Street="3rd Uffizi Rd." Town="Florence"/>
   …
</Publishers>
```

4.8 DML for XML

In order to manipulate XML in DB2 we need to use the transform statement (specified in the XQuery Update Facility specification). The syntax is a little different from what the specification describes, but this is probably due to the specification being finalized only a few months ago. The transform statement makes a copy of an XML value, modifies it and returns it. Technically, we could return something other than the modified copy, but that is hardly the intended usage of the transform statement. The transform statement, being an XQuery statement, must be used inside the function XMLQUERY. The PASSING keyword can be used to pass an XML value from the SQL context to the XQuery context. The result of the transform statement becomes the result of the function. The passed XML value itself is not affected, which means that we need to use an SQL UPDATE in order to store the modified value inside the table. So if we would like to change the information of an author, we would use the following statement:

```
UPDATE author
SET info = XMLQUERY('transform-statement' PASSING info)
WHERE …
```

The transform statement has three clauses and they are all required. A transform statement has the following structure:

```
(transform) optional keyword
copy variable assignment
modify modify-expression
return return-expression
```

The variable assignment will most probably be used to create a copy of the passed value, thus creating a copy to modify. The variable containing the copy will probably be the return-expression. The modify-expression is where we can add, remove and alter the content of your variable. The modify-expression can be any of the following expressions: do delete, do insert, do rename, or do replace. The modification specified in such an expression will be applied as many times as necessary. So if there are five nodes matching, then the specified modification will be performed five times, once for each matching node. In the following sections we will look at some examples that use the different modify expressions.

4.8.1 insert

When using a transform statement to add nodes to an XML value, you need to use a "do insert" expression. The placement of the new node will be relevant to an XPath expression and dependent on the specified position keyword (before, after, as last, as first). We could, for example, add a Website element to the info of the author Carl Sagan (this would actually violate the XML Schema, but let's ignore that for the sake of this example). The following statement finds Carl Sagan's row in the table author and updates the info column with the result of the XMLQUERY function. The XMLQUERY function takes the current value of the column info and adds a new element as the last child element of the root element.

```
UPDATE author
SET info = XMLQUERY('transform
                copy $res := $i
                modify do insert element Website {"www.carlsagan.com"} as last into $res/Info
                return $res'
            PASSING info AS "i")
WHERE name = 'Carl Sagan'
```

4.8.2 delete

If we want to remove a node, then we use the "do delete" expression in the modify clause. We can for example remove the Email element in the info XML of Carl Sagan:

```
UPDATE author
SET info = XMLQUERY('transform
            copy $res := $i
            modify do delete $res/Info/Email
            return $res'
        PASSING info AS "i")
WHERE name = 'Carl Sagan'
```

If the XPath expression specified after do delete matches several nodes, then all of them will be removed.

You can undo the change caused by the previous statement with the following statement:

```
UPDATE author
SET info = XMLQUERY('transform
                copy $res := $i
                modify do insert element Email {"carlsagan@nasa.gov"} as first into $res/Info
                return $res'
            PASSING info AS "i")
WHERE name = 'Carl Sagan'
```

4.8.3 rename

It is also possible to rename a node without having to remove it and create a new one. The node's location and value will be unchanged. We could, for example, change the name of the element Country to BirthCountry for all the authors (once again, this would violate the XML Schema).

```
UPDATE author
SET info = XMLQUERY('transform
            copy $res := $i
            modify do rename $res/Info/Country as "BirthCountry"
            return $res'
        PASSING info AS "i")
```

The XPath expression specified after "do rename" must match exactly one node. In this case it does, but what if we wanted to change all the Translation elements to Version elements in the XML values stored in the column edition.translations? According to the XML Schema there can be zero to many Translation elements in each Translations element. And that would cause an error. Fortunately, FLWOR expressions can be nested in the modify clause. We can instruct the modify clause to loop through all the Translation elements and do the rename once for each matching element:

```
UPDATE edition
SET translations = XMLQUERY('transform
            copy $res := $trans
            modify for $t in $res//Translation
                        return do rename $t as "Version"
            return $res'
        PASSING translations AS "trans")
```

You can undo the changes caused by the previous statements with these ones:

```
UPDATE author
SET info = XMLQUERY('transform
            copy $res := $i
            modify do rename $res/Info/BirthCountry as "Country"
            return $res'
        PASSING info AS "i")
```

```
UPDATE edition
SET translations = XMLQUERY('transform
        copy $res := $trans
        modify for $t in $res//Version
                return do rename $t as "Translation"
        return $res'
      PASSING translations AS "trans")
```

4.8.4 replace

It is also possible to replace a node with another node or sequence of nodes. A "do replace" expression identifies one node with an XPath expression and then replaces it with a node or a sequence of nodes. We can for example replace the Email element of Carl Sagan with a Skype element:

```
UPDATE author
SET info = XMLQUERY('transform
        copy $res := $info
        modify do replace $res//Email with element Skype {"carl.sagan.author"}
        return $res'
      PASSING info AS "info")
WHERE name = 'Carl Sagan'
```

A replace expression can also be used to replace the value of a node and not the node itself. The keywords "value of" should be used in such case. We could for example change Carl Sagan's year of birth (which is the content of the element YearOfBirth) to 1914.

```
UPDATE author
SET info = XMLQUERY('transform
            copy $res := $i
            modify do replace value of $res/Info/YearOfBirth with 1914
            return $res'
        PASSING info AS "i")
WHERE name = 'Carl Sagan'
```

If you want to restore Carl Sagan's info to the original value, just use the following statement:

```
UPDATE author
SET info = '<Info><Email>carlsagan@nasa.gov</Email><Country>USA</Country>
            <YearOfBirth>1913</YearOfBirth></Info>'
WHERE name = 'Carl Sagan'
```

4.9 XSLTRANSFORM

The function XSLTRANSFORM can be used to transform XML values based on an XSLT. The function requires the XML value to be transformed and the XSLT to be specified as parameters. We could, for example, apply the following XSLT to the info XML of the authors.

```
<xsl:transform xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <xsl:output method="xml"/>
  <xsl:template match="/">
    <xsl:element name="Details">
      <xsl:attribute name="Mailaddress"><xsl:value-of select="//Email"/></xsl:attribute>
      <xsl:attribute name="Country"><xsl:value-of select="//Country"/></xsl:attribute>
      <xsl:attribute name="Birthyear"><xsl:value-of select="//YearOfBirth"/></xsl:attribute>
    </xsl:element>
  </xsl:template>
</xsl:transform>
```

This XSLT restructures the information in the info XML and returns a Details element with three attributes.

We could ask for the info XML of Carl Sagan, transformed according to the XSLT, with the following statement:

```
SELECT XSLTRANSFORM(info USING
'<xsl:transform xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <xsl:output method="xml"/>
  <xsl:template match="/">
    <xsl:element name="Details">
      <xsl:attribute name="Mailaddress"><xsl:value-of select="//Email"/></xsl:attribute>
      <xsl:attribute name="Country"><xsl:value-of select="//Country"/></xsl:attribute>
      <xsl:attribute name="Birthyear"><xsl:value-of select="//YearOfBirth"/></xsl:attribute>
    </xsl:element>
  </xsl:template>
</xsl:transform>')
FROM author
WHERE name = 'Carl Sagan'
```

The result is the following XML value:

```
<?xml version="1.0" encoding="UTF-8"?>
<Details Mailaddress="carlsagan@nasa.gov" Country="USA" Birthyear="1913"/>
```

The function adds an xml declaration by default. It is also possible to pass parameters, which can be retrieved in the XSLT with xsl:param.

Of course the XSLT doesn't have to be provided in this way. We could, for example, create a table and store all of our XSLTs in it and then retrieve the one to use.

4.10 Native XQuery

In DB2 it is also possible to use XQuery independently of SQL. This is simply done by using the keyword XQUERY. We could, for example, execute the following:

```
XQUERY
for $x in (1 to 5)
let $r := attribute Number {$x}
return element Result {$r}
```

This would produce the following result (five rows):

```
<Result Number="1"/>
<Result Number="2"/>
<Result Number="3"/>
<Result Number="4"/>
<Result Number="5"/>
```

Of course, we may want to access our relational data in the XQuery statement. For that purpose, DB2 adds two XQuery functions. These functions retrieve relational data and expose them to the XQuery context.

4.10.1 Function db2-fn:sqlquery

If we want to access some relational data and use them in an XQuery statement, we can use the DB2 specific function db2-fn:sqlquery. This function takes a SELECT statement as its parameter and returns the result as a sequence. The SELECT statement must have exactly one column in its result and that column must be XML. We could retrieve the info XML of the authors in order to find all the countries of authors:

```
XQUERY
for $x in distinct-values(db2-fn:sqlquery("SELECT info FROM author")//Country)
return element Country {$x}
```

The result is one Country element for each unique country name:

```
<Country>England</Country>
<Country>Austria</Country>
<Country>Australia</Country>
<Country>France</Country>
…
```

We could, of course, have a much more complicated SELECT statement as parameter, perhaps generating the XML result with SQL/XML publishing functions. How about this:

```
XQUERY
for $c in db2-fn:sqlquery('SELECT XMLELEMENT(NAME "Country",
                                             XMLATTRIBUTES(country AS "Name"),
                                             XMLAGG(XMLFOREST(city AS "City")))
                           FROM publisher
                           GROUP BY country')
order by count($c/City) descending
return element Country {$c/@Name, attribute Cities {count($c/City)}}
```

This produces the following result:

```
<Country Name="Sweden" Cities="3"/>
<Country Name="Austria" Cities="1"/>
<Country Name="Turkey" Cities="1"/>
<Country Name="Scotland" Cities="1"/>
…
```

Another interesting feature of the function db2-fn:sqlquery is its support for parameters. The following statement groups the books based on the number of editions they have. In XQuery we loop through the sequence 0 to 10 and for each iteration we parameterize the SELECT statement with the current $num. The SELECT statement creates a sequence of Book elements with the books that have the correct number of editions. We eliminate numbers that have no books and return an element Books for each number with at least one book. We also place the Books elements inside a Root element:

```
XQUERY
element Root {
   for $num in (0 to 10)
   let $books := db2-fn:sqlquery('SELECT XMLFOREST(title AS "Book")
                                  FROM book
                                  WHERE (SELECT COUNT(*)
                                         FROM edition
                                         WHERE book = book.id) = parameter(1)',
                                 $num)
   where not(empty($books))
   return element Books {attribute NumberOfEditions {$num}, $books}
}
```

The result has the following structure:

```
<Root>
  <Books NumberOfEditions="1">
    <Book>Misty Nights</Book>
    <Book>Contact</Book>
   …
  </Books>
  <Books NumberOfEditions="2">
    <Book>Database Systems in Practice</Book>
    <Book>Våren vid sjön</Book>
   …
  </Books>
  <Books NumberOfEditions="3">
    <Book>Archeology in Egypt</Book>
  </Books>
   …
</Root>
```

4.10.2 Function db2-fn:xmlcolumn

The other DB2 specific XQuery function is called db2-fn:xmlcolumn. This function takes the qualified name of an XML column as an argument and returns the values in that column as a sequence. So we could use the following statement in order to get all the countries where there are authors.

```
XQUERY
for $a in distinct-values(db2-fn:xmlcolumn('AUTHOR.INFO')//Country)
return $a
```

The column name must be defined in the correct case. The default is upper case.

5 Epilogue

DB2 has been moving closer to the SQL standard with each new version. Many of the XML specific DB2 extensions have been abandoned and standard constructs have been integrated in the DB2 core engine. DB2 is not yet up to speed with the XML functionality described in the SQL standard. On the other hand, DB2 has addressed areas with its extensions that have yet to be covered in the SQL standard and/or in the XQuery standard. Many of the DB2 specific features described here will probably be replaced in the years to come. DB2 is the only major DBMS that supports the XQuery Update Facility. In the examples in the previous chapter we looked at some of the features that are available in DB2 9.7. There are many more details, but it has not been the goal of this introduction to cover everything.

I hope you have found this introduction educational and fun. Do not hesitate to send comments and suggestions that may help improve the next version of the compendium!

The Author
nikos dimitrakas