# Introduction to SQL Server & XML

Version 1.0
July 2011

nikos dimitrakas

# Table of contents

# 1 Introduction

This compendium gives a short introduction to SQL Server 2008 R2 and its facilities for database administration. We discuss installing SQL Server 2008 R2 and using the Microsoft SQL Server Management Studio. After that, there is an introduction to some Microsoft specific XML features. All the examples are tested on SQL Server for Windows on a Windows 7 64-bit platform, but they should work in a similar manner on any platform.

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

## *1.1 SQL Server*

SQL Server is Microsoft's major relational DBMS. It has certain similarities to Access, which is a smaller relational DBMS included in Microsoft Office. Microsoft has decided not to implement any XML support according to the latest SQL standards in SQL Server. Instead, Microsoft specific extensions have been designed, that add corresponding XML features to the ones described in the SQL standard. These Microsoft specific extensions are named SQLXML, while the XML part of the SQL standard goes under the name SQL/XML. These two are not to be confused with one another.

SQL Server has a set of tools for working with databases and for managing SQL Server systems. The Microsoft SQL Server Management Studio (Management Studio for short) serves as the main hub for performing most tasks. It has several wizards and support for SQL queries and scripts. There are several other tools bundled with SQL Server, but they are not relevant for 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 SQL Server 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 SQL Server but the recommended tool is the Management Studio (which is bundled with SQL Server).

## *1.3 Structure*

In the next chapter we will take a quick look at the installation and configuration of SQL Server and at the Management Studio. After that we will look at the sample data used in the examples to come. In chapter 3.1 we will go through several examples using the sample data and SQL Server's XML features.

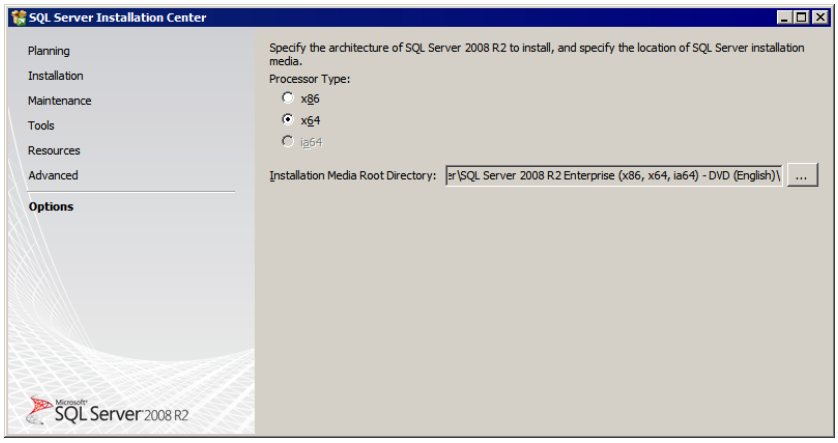
# 2 SQL Server 2008 R2

SQL Server 2008 R2 is available as a free trial by Microsoft. For students and faculty at most universities, SQL Server and other Microsoft products are available for free for non-commercial use through MSDNAA (Microsoft Developers Network Academic Alliance).

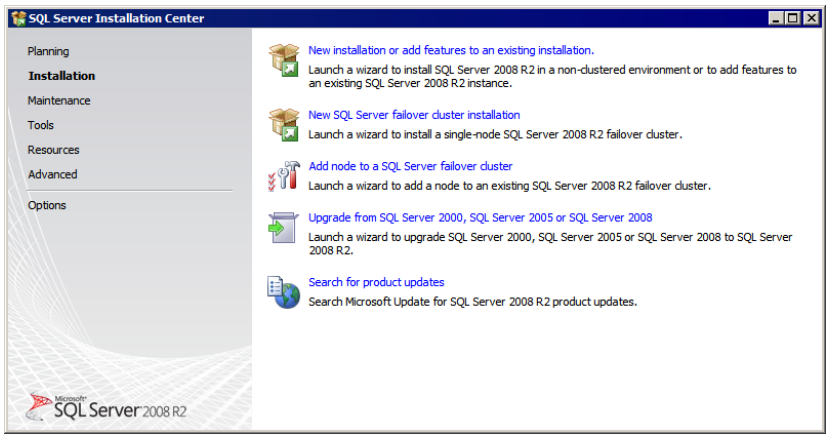
## *2.1 Installation*

Start by downloading the appropriate installation file. This compendium is based on SQL Server 2008 R2 Enterprise Edition for Windows x64.

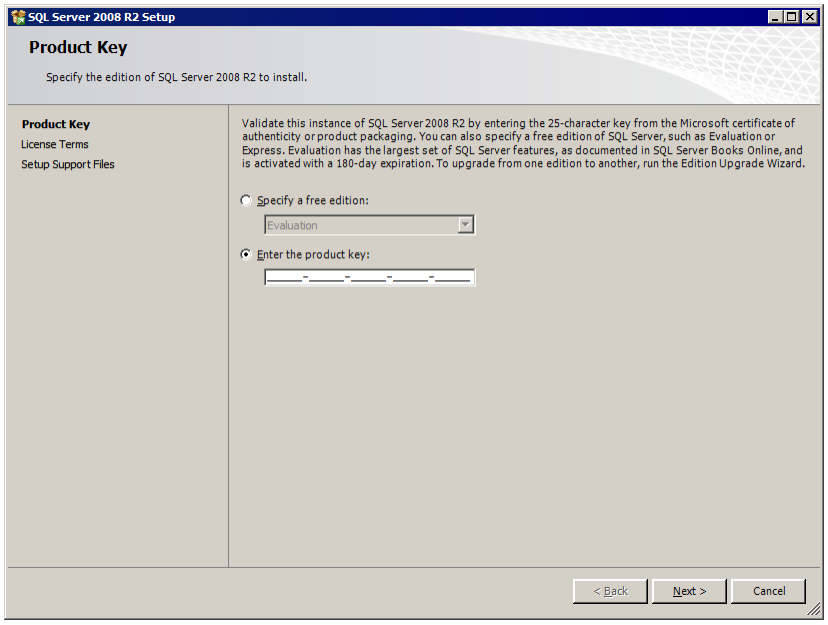
Run the executable setup.exe to start the installation. The SQL Server Installation Center will soon appear. In the menu on the left you can choose an activity. If you choose "Installation", then you will be presented with alternative installation options. Before starting an installation, you may want to check under "Options". Here you can specify if you want to install a 32-bit version or a 64-bit version.



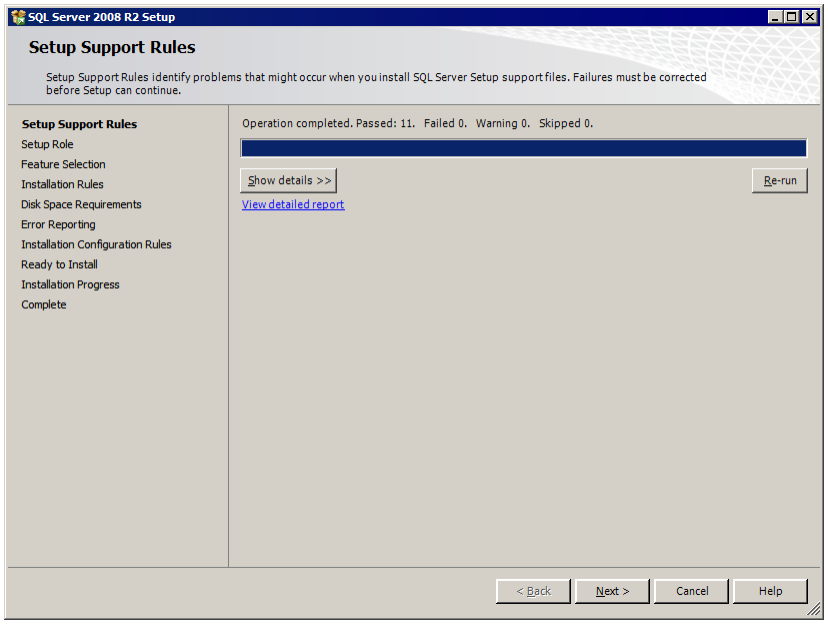

Start a new installation by selecting the option "New installation or add features to an existing installation".



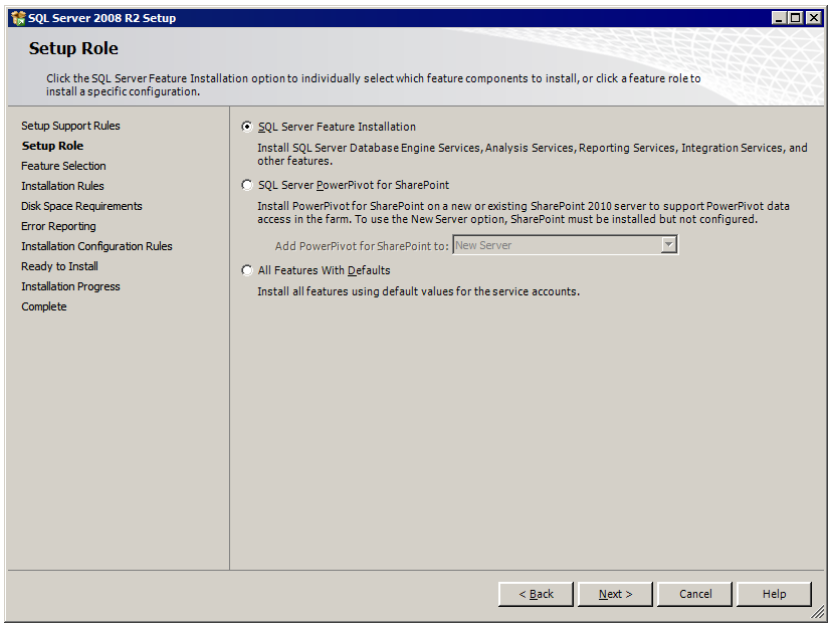
The installation wizard will launch and it may take a few minutes before it completes its preparations. You may also have to press "OK" a few times. Eventually, you will be asked to specify a product key:



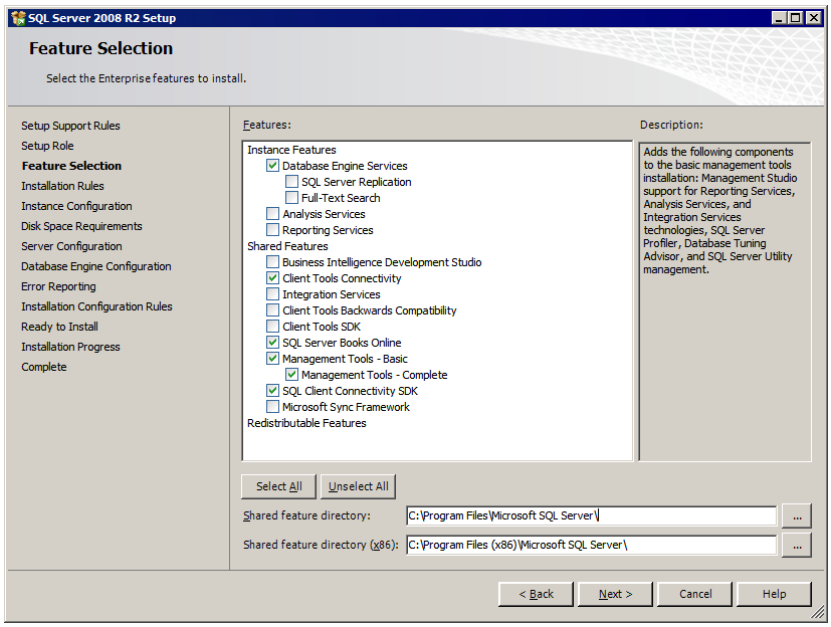

Specify your key or choose Evaluation and go to the next step where you must accept the license terms in order to continue. Press "Next" and then "Install" and the wizard will work for a while before the installation configuration wizard shows up.



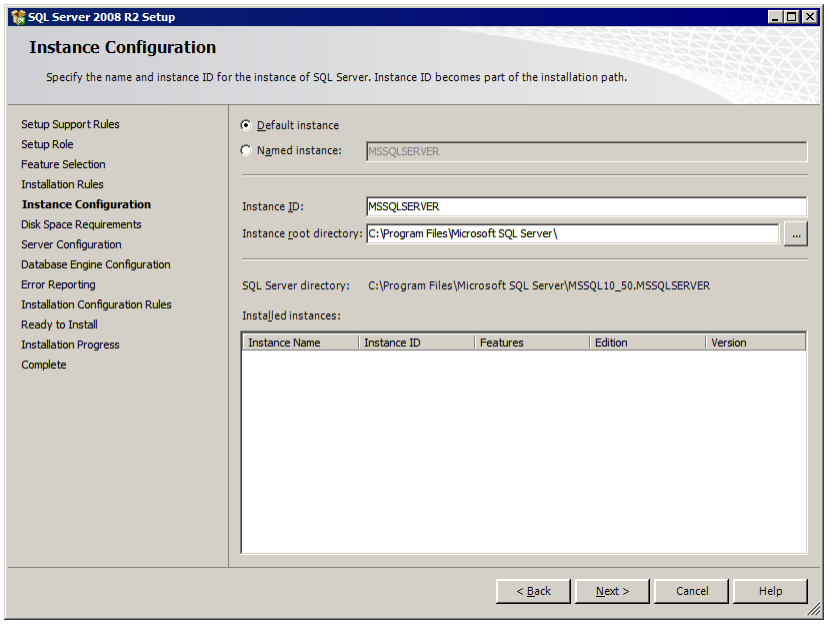
Go to the next step and select "SQL Server Feature Installation".



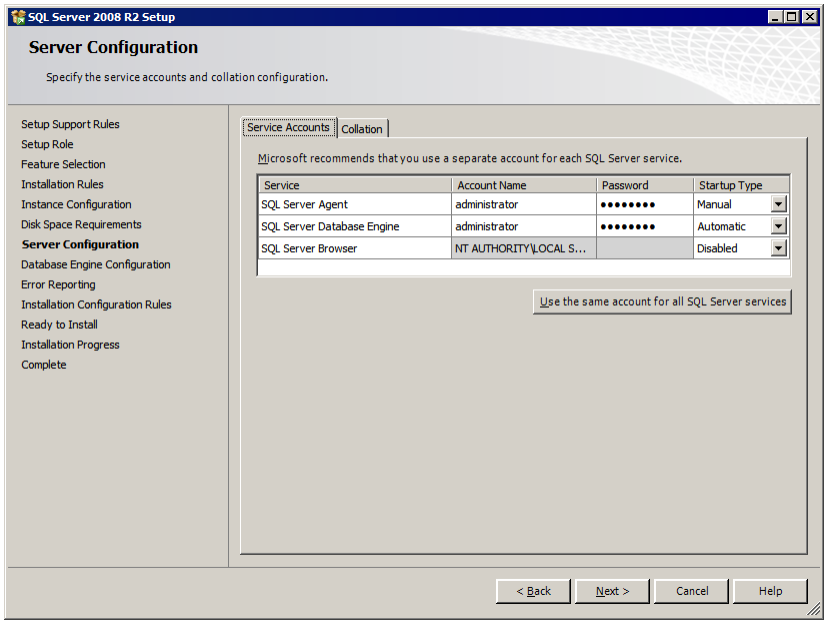

In the next step you must select which features to install. You can install what you want, but the Database Engine Services and Management Tools Basic must be installed. Here is our configuration.

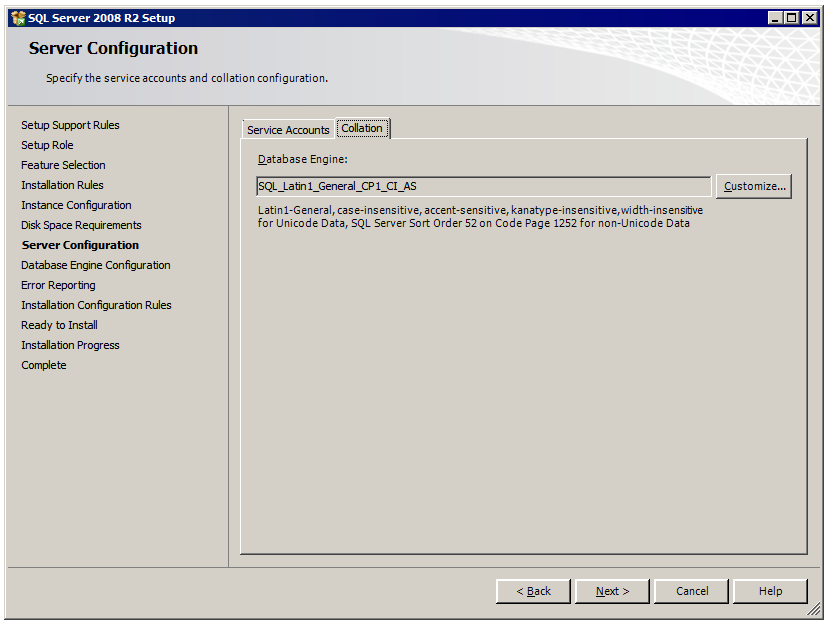


As soon as you press "Next" the wizard will check that your configuration and your system are compatible and consistent. Go to the next step to configure the instance to be installed. The defaults are just fine:

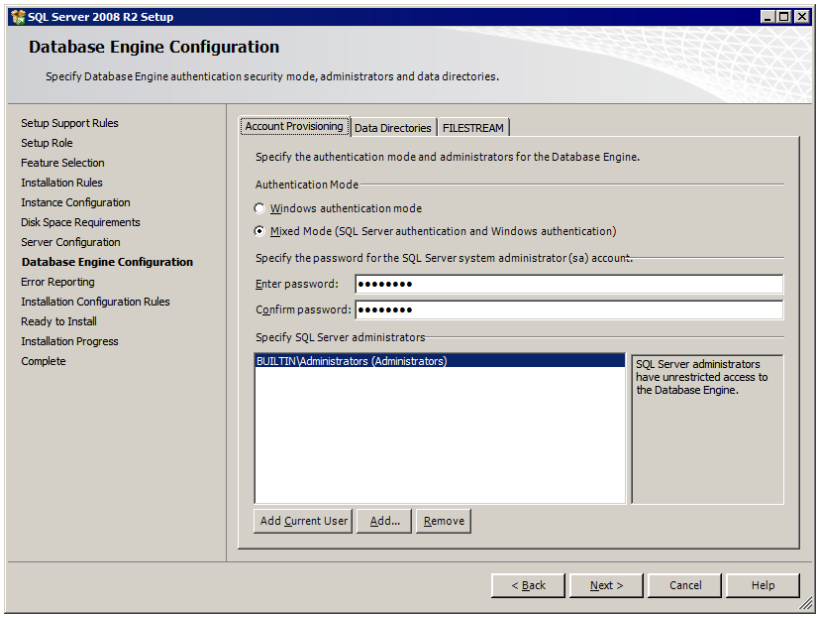



Go to the next step and the wizard will confirm that there is adequate disk space. In the next step you must configure the system services. You can specify which account should be used for starting each service. We use the same account for both services, but you can have separate accounts for each service. Just make sure the accounts have high enough permissions. During this step you can also configure the default collation for the server. The default (Latin1 general case-insensitive accent-sensitive) will be sufficient for this introduction.

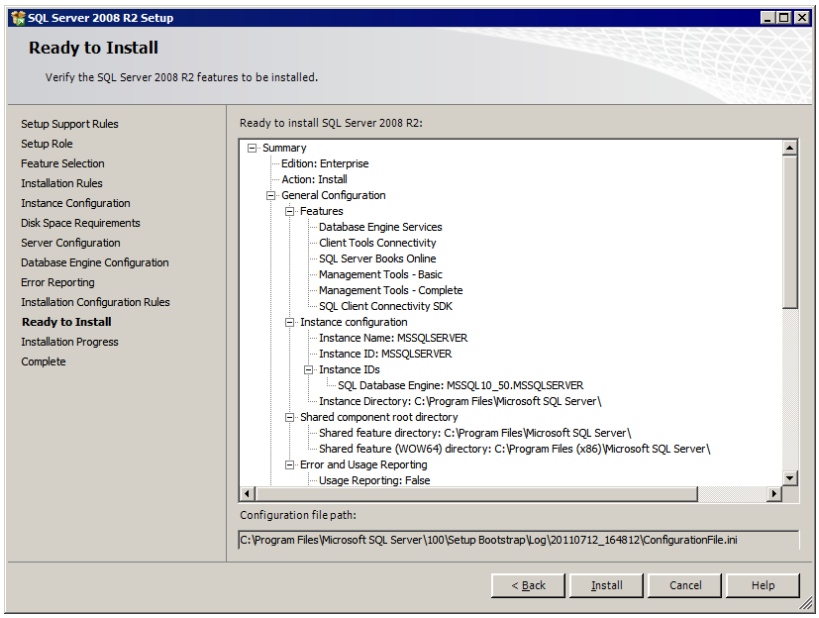





In the next step you can decide the authentication mode for users that connect to the SQL Server. Windows authentication mode requires that a user must have a Windows account in order to access the database server. With mixed mode, users can be created directly in the database server and are not required to have a Windows account. Users with Windows accounts can still access the database server. Mixed mode is more flexible, but perhaps less secure. You can also specify which Windows users should be administrators for SQL Server. We have added all Windows administrators to be administrators for SQL Server.

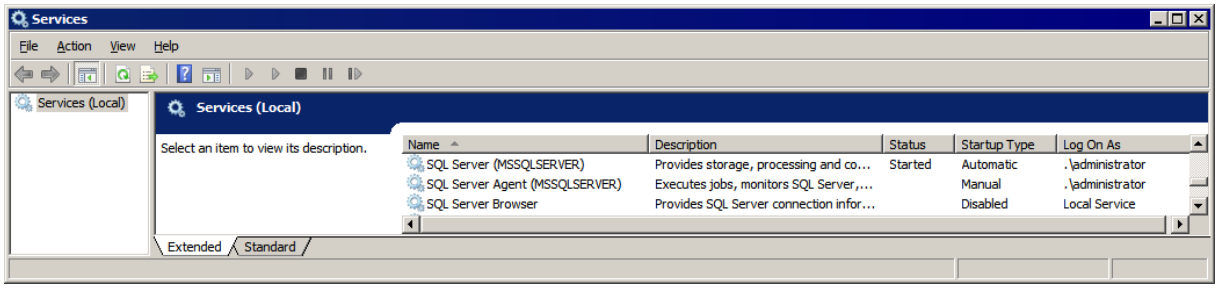


In the next step you can choose to provide error reports to Microsoft. After that the wizard will once again check that your configuration is good, before starting the actual installation. The wizard will present a summary of the configuration and the installation can be started by pressing "Install".




The installation will take a few minutes.

During the installation the Windows services configured earlier were created:

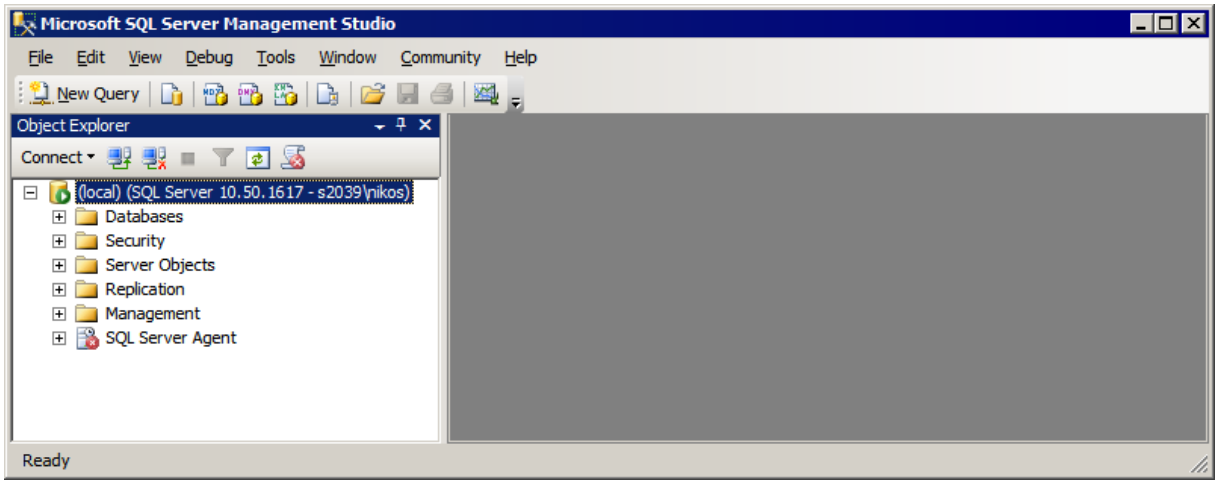


## *2.2 Management Studio*

The Management Studio is a graphical client where you can manage your SQL Server and execute SQL statements. This is the recommended tool for working with SQL.

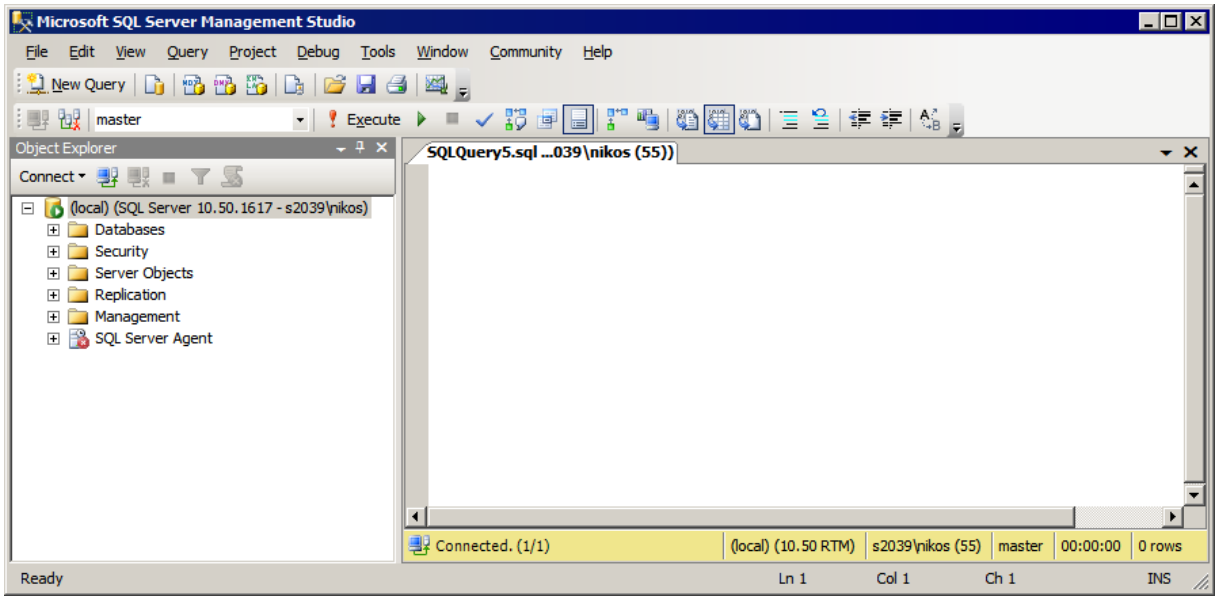
When you start the Management Studio, you will be asked to connect to a server. The default is to connect to the local server using the current Windows account. This is probably what you want.


Once connected, you will see a tree structure on the left, representing the different objects on the connected server.

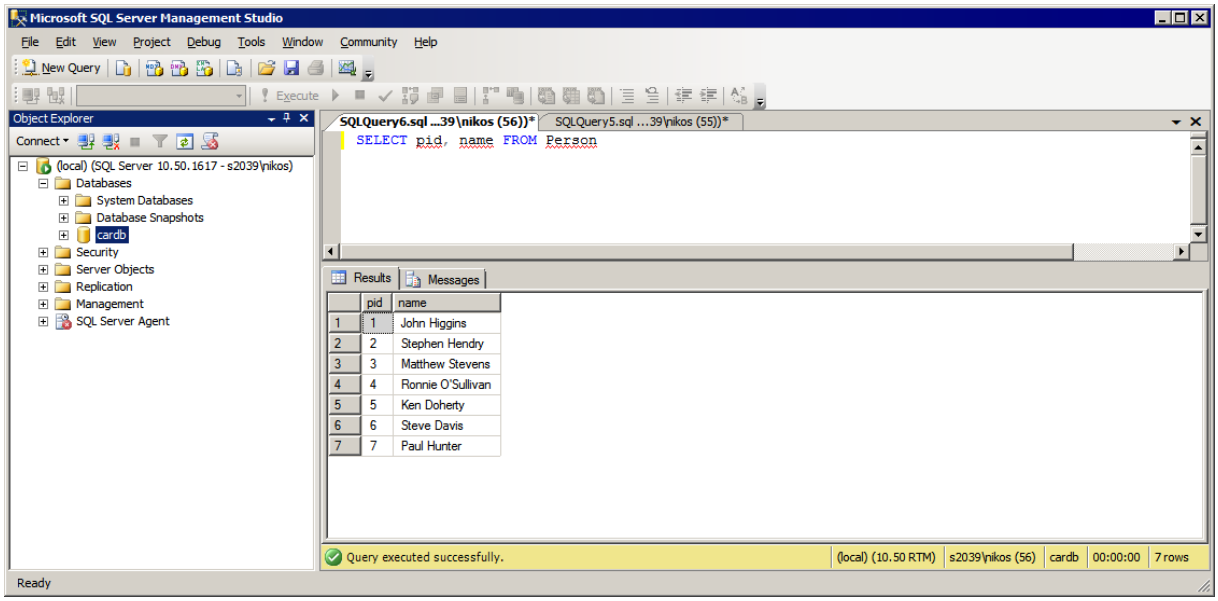



Right-click on any object in order to see the available options. The Management Studio offers several wizards for performing tasks.

On the right side of the window, you can open one or more query tabs. Each query tab is associated with a database. The default is the system database "master". The connected database is indicated in the toolbar and in the status bar of the query tab.



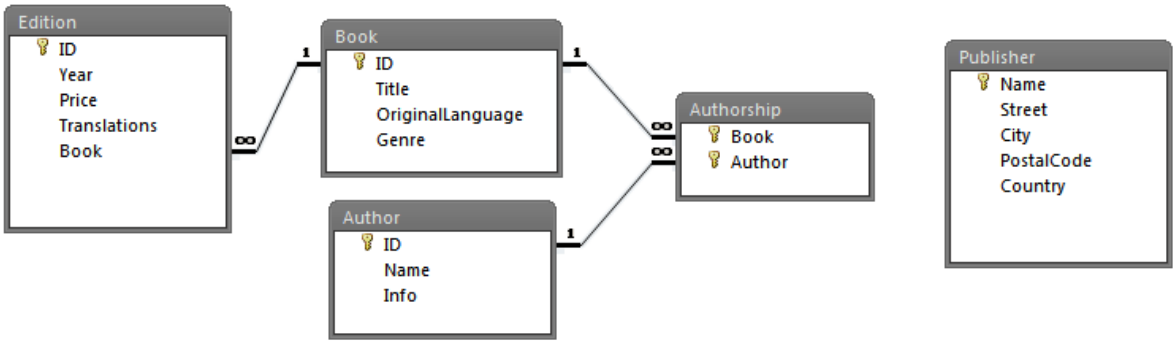

If you execute a query or a script (by pressing the Execute button or Ctrl + E), you will see the result in the lower half of the query tab.




The execute command will execute the selection and if no selection was made, it will execute the entire content of the query area.

# 3 Sample Data

In this chapter we will take a look at the data that we will use in all 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/sqlserverxml/

The script can be run in the Management Studio. It creates a database called bookdb.

## *3.1 XML data type*

SQL Server 2008 R2 has a data type called XML. This data type can be typed or untyped. The untyped XML data type accepts any well-formed XML or fragment, while the typed XML data type is associated with an XML Schema and allows only valid XML documents. There is no real support for DTDs, but inline DTDs are allowed and can be used to provide defaults. Any schema to be used, must first be registered as a SCHEMA COLLECTION.

In the provided database script, there is no validation.

# 4 Examples

In this chapter we will go through some examples using SQL Server specific XML features. All the examples in this chapter assume that the database has been created and that there is a connection to it.

## *4.1 FOR XML*

In order to create XML as output from an SQL SELECT statement, SQL Server adds an extra clause after the ORDER BY clause. The FOR XML clause can be used in different modes and it transforms the result of the SQL statement into an XML document or fragment. The result is serialized by default, but the keyword TYPE can be used in order to keep the result as a value of the XML data type. The three modes (RAW, AUTO and PATH) are described in the following sections.

### 4.1.1 RAW

If we want to create an XML document with all the publishers, we could use the RAW mode which by default will create one element per row and one attribute per column.

```
SELECT *
FROM publisher
FOR XML RAW ('Publisher'), ROOT('Publishers')
```

The result looks like this:

```
<Publishers>
   <Publisher name="ABC International" street="7th Bear St."
                city="Berlin" postalcode="44500" country="Germany" />
   <Publisher name="Addison" street="2nd Monet St."
               city="Toulouse" postalcode="98700" country="France" />
   …
</Publishers>
```

The element names and attribute names will have the specified case, and the default is lower case. The following example shows how we can change the case or even the entire name.

```
SELECT CitY, NamE, Country AS Land
FROM publisher
FOR XML RAW ('Publisher'), ROOT('Publishers')
```

This will give us the following result:

```
<Publishers>
   <Publisher CitY="Berlin" NamE="ABC International" Land="Germany" />
   <Publisher CitY="Toulouse" NamE="Addison" Land="France" />
   …
</Publishers>
```

The root element is created through the use of the optional keyword ROOT. The name of the root element can be specified as in the previous examples, but if not, the default is "root". Omitting the keyword ROOT will give us a fragment as the result (a sequence of Publisher elements in the previous examples). After the keyword RAW we can specify the element name for each row. The default is "row".

If we would prefer to have elements instead of attributes for the columns, we can specify the keyword ELEMENTS. Here is an example that also illustrates the absence of ROOT and element names.

```
SELECT Name, City, Country
FROM publisher
FOR XML RAW, ELEMENTS
```

The result is an XML fragment with one "row" element for each row and three subelements for the three selected columns:

```
<row>
  <Name>ABC International</Name>
  <City>Berlin</City>
  <Country>Germany</Country>
</row>
<row>
  <Name>Addison</Name>
  <City>Toulouse</City>
  <Country>France</Country>
</row>
…
```

Now, if a column is already XML, we may not get the result that we would want using the RAW mode. Consider the following example where we want to have one element per author.

```
SELECT name, info
FROM author
FOR XML RAW ('Author'), ROOT('Authors')
```

The column info is already XML so it will not be added as an attribute, but rather, as a subelement. The column name is "info" so a subelement "info" will be created. This will unfortunately lead to two subsequent "info" elements:

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

One way to avoid this is to have an unnamed column in the SELECT clause. That means that it needs to be generated either with a nested statement or with a function/method.

```
SELECT name, (SELECT info FROM author a WHERE a.id = author.id)
FROM author
FOR XML RAW ('Author'), ROOT('Authors')
```

The following also produces the same result:

```
SELECT name, info.query('/')
FROM author
FOR XML RAW ('Author'), ROOT('Authors')
```

### 4.1.2 AUTO

The AUTO mode is very similar to RAW. It will also create one attribute per column and one element per row, but the element name will be the same as the table name (in the case specified in the FROM clause). It can also be configured for elements and to include a root element. AUTO differs from RAW when more tables are involved in the SELECT clause. Then each table will get a new element, thus possibly creating several levels of subelements.

If we want to get an XML document with all the books and their respective editions, we can use the following statement.

```
SELECT Book.Title, Edition.Year
FROM Book, Edition
WHERE Edition.book = Book.id
FOR XML AUTO, ROOT('Books')
```

This will create a root element Books with one Book element for each book. Each Book element will have one Edition element for each edition. The join condition will, of course, eliminate any book without editions. The result looks like this:

```
<Books>
 <Book Title="Archeology in Egypt">
  <Edition Year="1992" />
  <Edition Year="1994" />
  <Edition Year="1999" />
 </Book>
 <Book Title="Contact">
  <Edition Year="1988" />
 </Book>
 <Book Title="Database Systems in Practice">
  <Edition Year="2000" />
  <Edition Year="2002" />
 </Book>
 …
</Books>
```

How does SQL Server know to nest Edition elements inside Book elements? The order of the columns in the SELECT clause decides how the elements will be nested. But only if subsequent rows have the same values in the outer level. So the ORDER BY clause may affect the result. Try the following statement with and without the ORDER BY clause.

```
SELECT Year, Title
FROM Book, Edition
WHERE Edition.book = Book.id
ORDER BY year
FOR XML AUTO, ROOT('Books')
```

And how about this?

```
SELECT Title, Year
FROM Book, Edition
WHERE Edition.book = Book.id
ORDER BY Year DESC
FOR XML AUTO, ROOT('Books')
```

How can we order the editions without destroying the nesting?

When many columns appear in the SELECT clause, it is still the order in which each table first appears in the SELECT clause that decides the nesting. Consider the following example:

```
SELECT Title, Year, Genre, Price
FROM Book, Edition
WHERE Edition.book = Book.id
FOR XML AUTO, ROOT('Books')
```

The first and third columns come from the table book, while the second and fourth columns come from the table edition. But the element creating and nesting is only affected by the first appearance of each table. Thus, the table book was first and the table edition was second. Any extra columns appearing later in the SELECT clause will be placed in the existing element corresponding to the relevant table.

The AUTO mode is completely dependent on the tables that appear in the FROM clause. So if we want to create a nesting that is not based on tables, we have to work around that. How about creating an XML document with books per genre? Since the genre and title are in the same table, AUTO mode would place them on the same element level. With the following statement, we let AUTO mode see two tables, which creates two element levels.

```
SELECT Name, Title
FROM Book, (SELECT DISTINCT genre AS name FROM book) AS Genre
WHERE genre = name
ORDER BY name
FOR XML AUTO, ROOT('Books')
```

The first column in the SELECT clause comes from the table Genre, so Genre elements are created under the root element. The second column in the SELECT clause comes from the table Book, so the Book elements will be subelements to Genre elements. Here is the result:

```
<Books>
  <Genre Name="Educational">
    <Book Title="European History" />
    <Book Title="Musical Instruments" />
    <Book Title="Oceans on Earth" />
    <Book Title="Archeology in Egypt" />
    <Book Title="Database Systems in Practice" />
    <Book Title="Music Now and Before" />
  </Genre>
  <Genre Name="Novel">
    <Book Title="Midsommar i Lund" />
    <Book Title="Våren vid sjön" />
    <Book Title="The Beach House" />
  </Genre>
  <Genre Name="Science Fiction">
    <Book Title="Contact" />
    <Book Title="The Fourth Star" />
  </Genre>
  <Genre Name="Thriller">
    <Book Title="Dödliga Data" />
    <Book Title="Misty Nights" />
  </Genre>
</Books>
```

### 4.1.3 PATH

Working with RAW and AUTO modes can be quite challenging when we want to create structures that combine attributes and elements and have several levels of arbitrary nesting. PATH mode is the most flexible mode of the FOR XML clause. It allows us to fully control the XML structure and to place each value at the level we want. One drawback is that it does not have automatic grouping when nesting like the AUTO mode has. But that can be handled with nesting SELECT statements appropriately.

In PATH mode, the column names and aliases define where in the generated structure different values should be placed. Consider the following example.

```
SELECT name AS "@Name", city AS "Location/@City", country AS "Location/@Country"
FROM publisher
FOR XML PATH ('Publisher'), ROOT('Publishers')
```

This will create a root element Publishers with one Publisher element for each row. The values of the three columns in the SELECT clause will be placed in the nodes (relative to the Publisher element) specified by the paths in the column aliases. Here is the result:

```
<Publishers>
  <Publisher Name="ABC International">
    <Location City="Berlin" Country="Germany" />
  </Publisher>
  <Publisher Name="Addison">
    <Location City="Toulouse" Country="France" />
  </Publisher>
  …
</Publishers>
```

Grouping subelements (as AUTO mode does) requires that we nest SELECT statements. If we want to get all the books per genre, we may write the following statement.

```
SELECT name AS "@Name", (SELECT Title
                         FROM Book
                         WHERE genre = g.name
                         FOR XML AUTO, TYPE)
FROM (SELECT DISTINCT genre AS name
      FROM book
      WHERE genre IS NOT NULL) AS g
FOR XML PATH('Genre'), ROOT('Books')
```

In the outer SELECT statement we work with one table named g. This table contains all the genres. In the SELECT clause we have one column to be placed as a Name attribute in the Genre element, and another column (the nested statement) that is unnamed, which means that it should be the content of the Genre element. It is also possible to specify that a column should become the content of the Genre element by using the column alias "*".

The result of the previous statement looks like this:

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

It is very important that the result of the nested statement is XML. That's why we use the TYPE keyword. If the result had been a serialized XML (as VARCHAR), then it would have become the text node of the Genre element and not subelements.

## *4.2 XML data type methods*

SQL Server offers a handful of methods that can be used on objects of the data type XML. The methods query, value, exist and nodes provide similar functionality to the SQL standard's functions XMLQUERY, XMLTABLE and XMLEXISTS. The method modify addresses DML for XML, which the SQL standard has yet to address.

### 4.2.1 Methods query and value

Using XQuery on an XML object can be done with the methods query and value. The first returns XML (either a fragment or a document), while the second returns a singleton value of a basic data type.

If we want to get the name and country of each author, we could use the following statement.

```
SELECT name, info.value('(//Country)[1]', 'VARCHAR(20)')
FROM author
```

The value method requires that the XQuery expression (often just an XPath expression) is statically a single node. This means that in most cases (even when we know that the result is one node), we need to add the predicate [1] after the expression (which we place inside parentheses). The second argument of the method is the data type of the result (as a string).

The above statement returns the content of the first Country element. Compare it with the following, which returns the first text node under the Country element:

```
SELECT name, info.value('(//Country/text())[1]', 'VARCHAR(20)')
FROM author
```

In this case the result would be the same since the Country element only has one text node.

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

But consider the following example:

```
DECLARE @x XML;
SET @x = '<A>hey<B>all</B>you</A>'
SELECT @x.value('(//A)[1]', 'VARCHAR(20)')
SELECT @x.value('(//A/text())[1]', 'VARCHAR(20)')
SELECT @x.value('(//A/text())[2]', 'VARCHAR(20)')
SELECT @x.value('(//A//text())[2]', 'VARCHAR(20)')
```

The first SELECT returns the content of the first A element, while the second one returns the first text node under the A element. The third SELECT returns the second text node directly

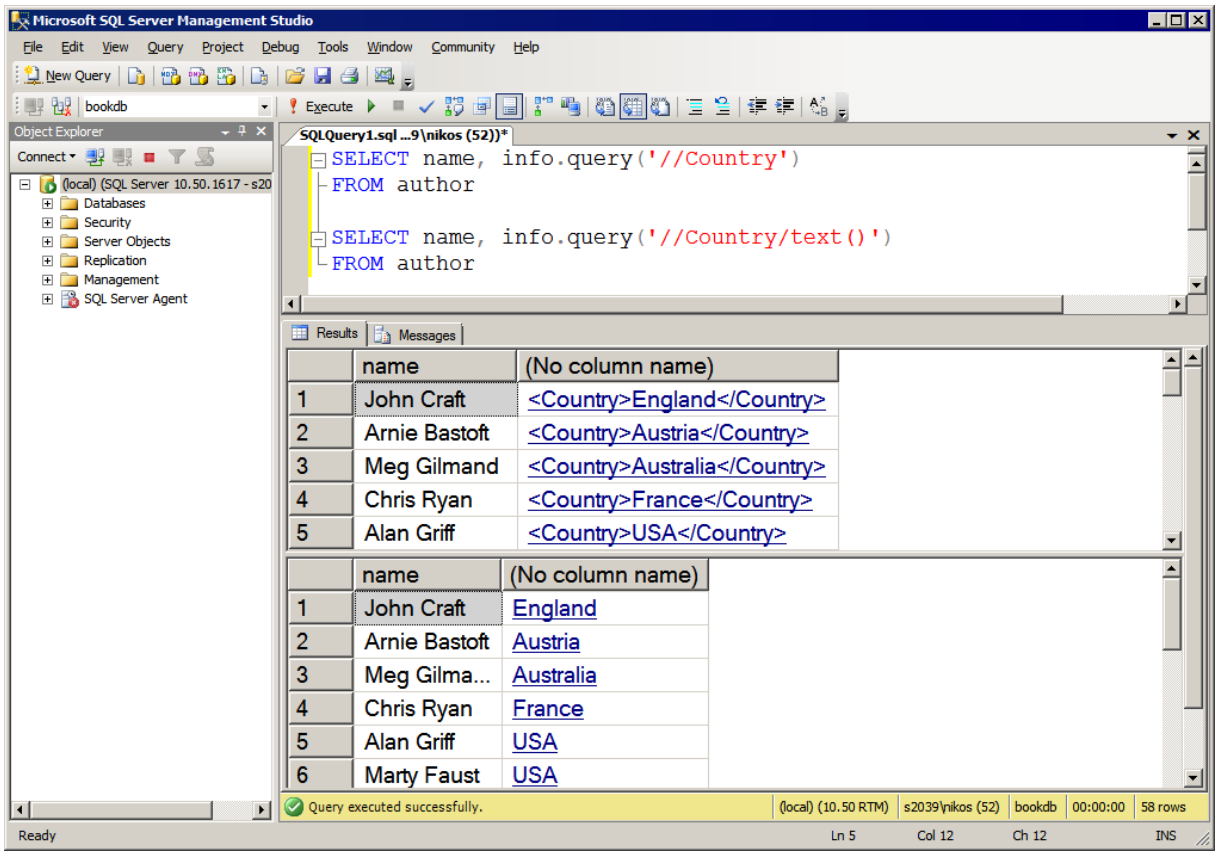
under the A element, and the fourth one returns the second text node inside the element A (which is the text node of the B element).

The method query is similar to value, but it always returns XML, so it does not need a second parameter. The following two statements, therefore, do not produce the same result:

```
SELECT name, info.query('//Country')
FROM author
```

```
SELECT name, info.query('//Country/text()')
FROM author
```

The first one returns the Country element, while the second one returns the text node under the Country element. The result may appear to be the same as when using the method value, but its data type is different, which is indicated by the blue color and underline in the Management Studio:




The query method can, of course, return XML fragments, so it is not bound by the same restrictions as the value method. We may want to find all the languages a book has been translated into and have them as XML. The following statement does that. Well, almost.

```
SELECT title, translations.query('for $l in //Translation/@Language
                                  return element Language {data($l)}')
FROM edition, book
WHERE book = book.id
```

The problem is that a book may have many editions and thus the result would be one row per edition:

```
Misty Nights          <Language>German</Language>
                      <Language>French</Language>
                      <Language>Russian</Language>
Archeology in Egypt <Language>Swedish</Language>
                      <Language>French</Language>
Archeology in Egypt <Language>Swedish</Language>
                      <Language>French</Language>
                      <Language>Chinese</Language>
Archeology in Egypt <Language>French</Language>
                      <Language>Turkish</Language>
                      <Language>Spanish</Language>
…
```

We could, of course, correct this by first merging all the translations of every book's editions, or by nesting in some practical way. Here is an example.

```
SELECT title, (SELECT translations.query('for $l in //Translation/@Language
                                          return element Language {data($l)}')
               FROM Edition
               WHERE book = book.id
               FOR XML PATH(''), TYPE)
FROM book
```

This will, of course, return duplicates if two different editions have been translated into the same language and the DISTINCT keyword is not allowed when XML columns are involved. Using distinct-values in the XQuery expression would not help either, since it works with one edition at the time. But we could add a second XQuery expression on the result (which is XML since we used the keyword TYPE):

```
SELECT title, (SELECT translations.query('for $l in //Translation/@Language
                                          return element Language {data($l)}')
               FROM Edition
               WHERE book = book.id
               FOR XML PATH(''), TYPE).query('for $l in distinct-values(//Language)
                                          return element Language {$l}')
FROM book
```

The result is now perfect:

```
Archeology in Egypt <Language>Swedish</Language>
                    <Language>French</Language>
                    <Language>Chinese</Language>
                    <Language>Turkish</Language>
                    <Language>Spanish</Language>
Contact             <Language>Swedish</Language>
                    <Language>German</Language>
                    <Language>Russian</Language>
…
```

If you feel that the previous statement is too complex, then try the following statement.

```
SELECT title, T.x.query('for $l in distinct-values(//@Language)
                        return element Language {$l}')
FROM book CROSS APPLY (SELECT translations
                       FROM edition
                       WHERE book = book.id
                       FOR XML RAW, TYPE) AS T(x)
```

The use of the keywords CROSS APPLY allows us to use a column from the table book inside the nested statement that creates the table T (with column x). So what this does, is that it takes the translations of the editions of the current book and merges them into one XML. This XML can then be queried and it contains all the languages of all the editions.

### 4.2.2 Method exist

The method exist can be used to check if a particular XQuery expression (in most cases an XPath expression) has a non-empty result. We could, of course, use the method value or the method query and then add an SQL predicate on the result, but the method exist is far more convenient. We may want to find all the authors from Sweden. The following statement has a condition that does exactly that.

```
SELECT name
FROM author
WHERE info.exist('//*[Country="Sweden"]') = 1
```

What the condition says, is that there must exist a node that has a subelement Country with the value "Sweden". The result of the method is 1 if true and 0 if false. The result has two rows:

```
Jakob Hanson
Marie Franksson
```

The condition could, of course, be written in many ways. All the following are equivalent (in this case):

```
info.exist('//Country[.="Sweden"]') = 1
info.exist('//Country[text()="Sweden"]') = 1
info.exist('/Info/Country[text()="Sweden"]') = 1
info.exist('/Info[Country="Sweden"]') = 1
info.value('(//Country)[1]', 'VARCHAR(20)') = 'Sweden'
```

### 4.2.3 Method nodes

The method nodes can be used to create a relational table from an XML sequence. Each node in the sequence becomes one row in the resulting table. We could, for example, use the following statement to retrieve books that have been translated into Swedish.

```
SELECT DISTINCT title
FROM book, edition
WHERE book = book.id
AND 'Swedish' IN (SELECT x.value('.', 'VARCHAR(20)')
                  FROM translations.nodes('//@Language') AS C(x))
```

The method nodes creates a table where each row is a Language attribute node. This table is given the alias T with column name x. We then use the value method on x to retrieve the value of the node as a VARCHAR so that it can be compared to 'Swedish'. This would of course be a very complex way to solve this particular problem, which can be solved with the following statement instead:

```
SELECT DISTINCT title
FROM book, edition
WHERE book = book.id
AND translations.exist('//Translation[@Language = "Swedish"]')=1
```

A more suitable situation for the method nodes is the following. Let's get the translations of each book.

```
SELECT title, year, x.query('.')
FROM book, edition CROSS APPLY translations.nodes('//Translation') AS Translation(x)
WHERE book = book.id
```

Here we use the keywords CROSS APPLY in order to use the column translations directly in the FROM clause and generate the table Translation. The result of the method nodes is a table where each row is a reference to a node in the original XML object. The generated column can therefore not be used directly. We use the method query in order to retrieve a new XML value. The result looks like this:

```
Misty Nights            1987  <Translation Language="German" Publisher="Kingsly" Price="130" />
Misty Nights            1987  <Translation Language="French" Publisher="Addison" Price="135" />
Misty Nights            1987  <Translation Language="Russian" Publisher="Addison" Price="125" />
Archeology in Egypt     1992  <Translation Language="Swedish" Price="340" />
Archeology in Egypt     1992  <Translation Language="French" Price="320" />
Archeology in Egypt     1994  <Translation Language="Swedish" Publisher="KLC" Price="390" />
Archeology in Egypt     1994  <Translation Language="French" Publisher="KLC" Price="330" />
Archeology in Egypt     1994  <Translation Language="Chinese" Publisher="Shou-Ling" Price="280" />
Archeology in Egypt     1999  <Translation Language="French" Publisher="KLC" Price="320" />
Archeology in Egypt     1999  <Translation Language="Turkish" Publisher="Turk And Turk" Price="300" />
Archeology in Egypt     1999  <Translation Language="Spanish" Price="300" />
…
```

We can, of course, shred this further and get the language, publisher and price as columns:

```
SELECT title, year, x.value('@Language', 'VARCHAR(20)'),
        x.value('@Publisher', 'VARCHAR(30)'), x.value('@Price', 'INTEGER')
FROM book, edition CROSS APPLY translations.nodes('//Translation') AS Translation(x)
WHERE book = book.id
```

The result looks like this:

```
Misty Nights            1987  German   Kingsly        130
Misty Nights            1987  French   Addison        135
Misty Nights            1987  Russian  Addison        125
Archeology in Egypt     1992  Swedish  NULL           340
Archeology in Egypt     1992  French   NULL           320
Archeology in Egypt     1994  Swedish  KLC            390
Archeology in Egypt     1994  French   KLC            330
Archeology in Egypt     1994  Chinese  Shou-Ling      280
Archeology in Egypt     1999  French   KLC            320
Archeology in Egypt     1999  Turkish  Turk And Turk  300
Archeology in Egypt     1999  Spanish  NULL           300
…
```

We could, of course, use XQuery to return "N/A" instead of NULL when the Publisher attribute is not present:

```
SELECT title, year, x.value('@Language', 'VARCHAR(20)'),
      x.value('if (empty(@Publisher)) then "N/A" else string(@Publisher)', 'VARCHAR(30)'),
      x.value('@Price', 'INTEGER')
FROM book, edition CROSS APPLY translations.nodes('//Translation') AS Translation(x)
WHERE book = book.id
```

SQL Server does not like mixing attribute nodes with literals so we use the XQuery function string.

### 4.2.4 Method modify

The method modify can be used to perform DML operations similar to SQL's INSERT, DELETE and UPDATE on XML. This method accepts three kinds of expressions: insert, delete, replace value of. We discuss these expressions further in section 4.3.

## *4.3 DML for XML*

In order to manipulate XML with operations similar to SQL's INSERT, DELETE and UPDATE, we need to use the XML method modify. The method itself is designed in such a way that it is valid in a context where an assignment is expected. So it cannot be used inside a SELECT statement. But it can be used in the SET clause of an UPDATE statement. If we would like to modify in some manner Carl Sagan's info XML we would use the following statement:

```
UPDATE author
SET info.modify('modify-expression')
WHERE name = 'Carl Sagan'
```

The method modify actually affects the object on which it is called, so the modification is made to the value in the column. The method modify does not return anything (and that is why it cannot be used in any other context).

There are three types of expressions that can be used as the method's parameter. The following sections give some examples of them.

### 4.3.1 insert

An insert expression can be used to add new nodes. The new nodes can be added before or after a particular node (as siblings), or as the first or last children of a particular node. We could for example add a Website element as the last child element of the info element in Carl Sagan's info XML. This would, of course, violate the XML Schema, but we can ignore that right now. Here is the statement that performs the update:

```
UPDATE author
SET info.modify('insert element Website {"www.carlsagan.com"} as last into (/Info)[1]')
WHERE name = 'Carl Sagan'
```

As usual in SQL Server, when an XPath expression must statically give a single node, we use the predicate [1]. The new element node will become the last child node of the (first) Info element. The new node doesn't have to be constructed. It can be specified in its serialized form. So the following would have the exact same effect:

```
UPDATE author
SET info.modify('insert <Website>www.carlsagan.com</Website> as last into (/Info)[1]')
WHERE name = 'Carl Sagan'
```

If we would prefer to add the Website element immediately after the Email element, then we could use the following instead:

```
UPDATE author
SET info.modify('insert <Website>www.carlsagan.com</Website> after (/Info/Email)[1]')
WHERE name = 'Carl Sagan'
```

The new element node becomes the next sibling to the node identified by the XPath expression after the keyword "after".

### 4.3.2 delete

To remove one or more nodes, a delete expression can be used. The delete expression deletes any node matching a specified XPath expression. So if the XPath expression doesn't match any node, the XML value will be unaffected. We could perhaps remove the Email element from Carl Sagan's info XML:

```
UPDATE author
SET info.modify('delete /Info/Email')
WHERE name = 'Carl Sagan'
```

This removes the element node, while the following would instead remove the text node leaving an empty Email element:

```
UPDATE author
SET info.modify('delete /Info/Email/text()')
WHERE name = 'Carl Sagan'
```

Carl Sagan's info XML would look like this after the previous statement:

```
<Info>
  <Email />
  <Country>USA</Country>
  <YearOfBirth>1913</YearOfBirth>
</Info>
```

If you want to restore Carl Sagan's info XML to the original value, 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'
```

### 4.3.3 replace value of

In some cases, we may want to change the value of a particular node, instead of removing it and creating a new one. The expression "replace value of" lets us identify a node and provide a new value for it. We can, for instance, change Carl Sagan's e-mail:

```
UPDATE author
SET info.modify('replace value of (/Info/Email/text())[1] with "carl@sagan.info"')
WHERE name = 'Carl Sagan'
```

The XPath expression must be statically a single node, so we use the predicate [1]. In most cases the node must be a text node or an attribute node.

## *4.4 XQuery functions*

SQL Server does not support so many of the functions defined in the XQuery standard. Refer to the documentation in order to see which functions are supported. SQL Server adds two extra XQuery functions that make it possible to pass SQL context values to the XQuery context. These two functions are described in this section.

### 4.4.1 sql:column

The function sql:column can be used inside an XQuery statement in order to access a column available in the SQL context that initiated the XQuery statement. We could, for example, create an XML document per author where we have the name and country as attributes:

```
SELECT info.query('let $c := //Country/text()
                   return element Author {attribute Country {$c},
                                          attribute Name {sql:column("name")}}')
FROM author
```

The XQuery statement accesses the name of the current author from the SQL context with the function sql:column. We construct the attributes in the return clause because SQL Server does not support computed constructors in the let clause. The result of the previous statement has one row per author and one column containing the XML document:

```
<Author Country="England" Name="John Craft" />
<Author Country="Austria" Name="Arnie Bastoft" />
<Author Country="Australia" Name="Meg Gilmand" />
<Author Country="France" Name="Chris Ryan" />
…
```

### 4.4.2 sql:variable

The function sql:variable is similar to sql:column, but is relevant in T-SQL when we execute a block that has variables. Here is an example:

```
DECLARE @x XML
SET @x = ''
DECLARE @val INTEGER
SET @val = 2
SELECT @x.query('for $e in (1,2,3,4,5,6,7,8,9,10)
        let $n := sql:variable("@val")
        return element Math {$n , "*", $e, "=", $n*$e}')
```

We create an empty XML value just so that we can call the method query which we can use to execute XQuery. In the XQuery statement we retrieve the variable @val from the SQL context and place it (its value) in the XQuery variable $n. The for clause loops through the sequence and for each value of $e a Math element is produced. SQL Server does not support sequence construction with the keyword "to", so we cannot use (1 to 10). The result is the following XML fragment:

```
<Math>2 * 1 = 2</Math>
<Math>2 * 2 = 4</Math>
<Math>2 * 3 = 6</Math>
<Math>2 * 4 = 8</Math>
<Math>2 * 5 = 10</Math>
<Math>2 * 6 = 12</Math>
<Math>2 * 7 = 14</Math>
<Math>2 * 8 = 16</Math>
<Math>2 * 9 = 18</Math>
<Math>2 * 10 = 20</Math>
```

The following example is not supported, since SQL Server does not support variable node names in computed constructors:

```
DECLARE @x XML
SET @x = ''
DECLARE @en VARCHAR(10)
SET @en = 'Number'
SELECT @x.query('for $n in (1,2,3)
        return element {sql:variable("@en")} {$n}')
```

The function sql:variable is quite limited. It may only contain a single value and may only be used in specific contexts.

# 5 Epilogue

SQL Server has chosen not to implement the XML functionality described in the latest SQL standards. Instead, there are several SQL Server specific extensions to SQL that can be used to produce the same results. In some cases the SQL Server specific solutions are simpler, while in other cases they are more complex. Either way, it is uncertain how long it will take before SQL Server implements the functionality described in the SQL standard. All the XML-related keywords from the SQL standard are considered reserved words for future use. SQL Server also has features that complement the SQL standard. Maybe some of them will be incorporated in the coming versions of the SQL standard.

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*