

Introduction to Oracle & XML

Version 1.0.1

July 2012

nikos dimitrakas



Table of contents

1	INTRODUCTION	3
1.1	ORACLE	3
1.2	PREREQUISITES	3
1.3	STRUCTURE	3
2	ORACLE 11G R2	3
2.1	INSTALLATION	3
2.1.1	<i>Services.....</i>	7
2.2	SQL DEVELOPER	7
3	SAMPLE DATA	8
3.1	XML DATA TYPE	10
4	EXAMPLES.....	10
4.1	XMLELEMENT, XMLFOREST, XMLATTRIBUTES.....	10
4.2	XMLAGG	11
4.3	XMLQUERY.....	13
4.4	XMLTABLE.....	14
4.5	XMLEXISTS	15
4.6	METHOD/FUNCTION EXTRACT AND FUNCTION EXTRACTVALUE	16
4.7	METHOD/FUNCTION EXISTSNode	16
4.8	XMLCOLATTVAL	17
4.9	DML FOR XML.....	17
4.9.1	<i>UpdateXML.....</i>	18
4.9.2	<i>DeleteXML.....</i>	18
4.9.3	<i>Insert and Append functions</i>	19
4.10	OTHER XMLTYPE METHODS.....	20
4.11	XMLTRANSFORM.....	21
4.12	XQUERY FUNCTION ORA:VIEW	22
5	EPILOGUE.....	23

1 Introduction

This compendium gives a short introduction to Oracle 11g R2 and its facilities for database administration. We discuss installing Oracle 11g R2 and using SQL Developer. After that, there is an introduction to some Oracle specific XML features accompanied by SQL/XML features supported by Oracle. All the examples are tested on Oracle 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 Oracle for Windows.

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

1.1 Oracle

Oracle is one of the major DMBSs and its latest versions have added support for XML mostly according to the latest SQL standards. The main tool for working with an Oracle database (in version 11) is called SQL Developer. SQL Developer requires Java (also an Oracle product). The Oracle Enterprise Manager is a web-based tool for administrating an Oracle server and its database objects.

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 Oracle 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 Oracle (like SQL Plus, TOAD, etc) but the recommended tool is SQL Developer (which is bundled with Oracle).

1.3 Structure

In the next chapter we will take a quick look at the installation and configuration of Oracle and at SQL Developer. 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 Oracle's XML features.

2 Oracle 11g R2

Oracle 11g R2 is available for free by Oracle for non-commercial use. The installation is divided into two zip-files that are available on oracle.com. On the same site there are detailed instructions for installation, configuration and other tasks.

2.1 Installation

Start by downloading the appropriate installation files. This compendium is based on version 11.2.0.1.0 for Windows x64. In order to download the installation files, you may need to create a free account.

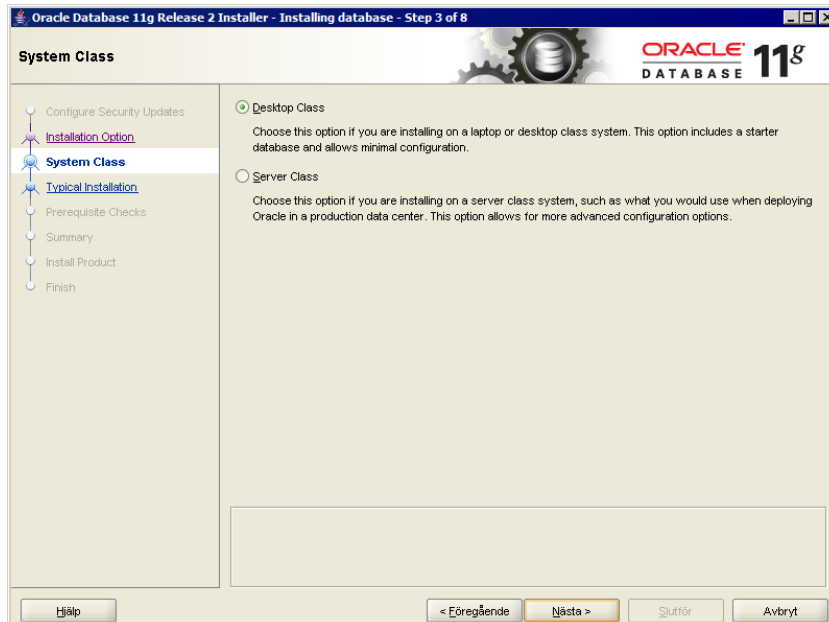
Unzip the two files in the same folder prior to initiating the installation. All the files should be in the folder "database" where you should also find the executable setup.exe.

Run setup.exe to start the installation.

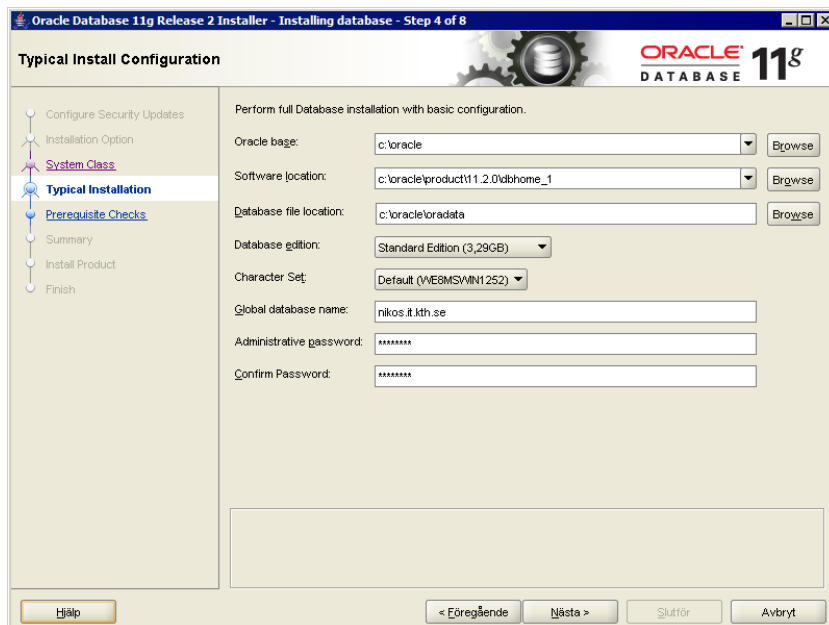
Eventually you will see three install options. Choose to "Create and configure a database":



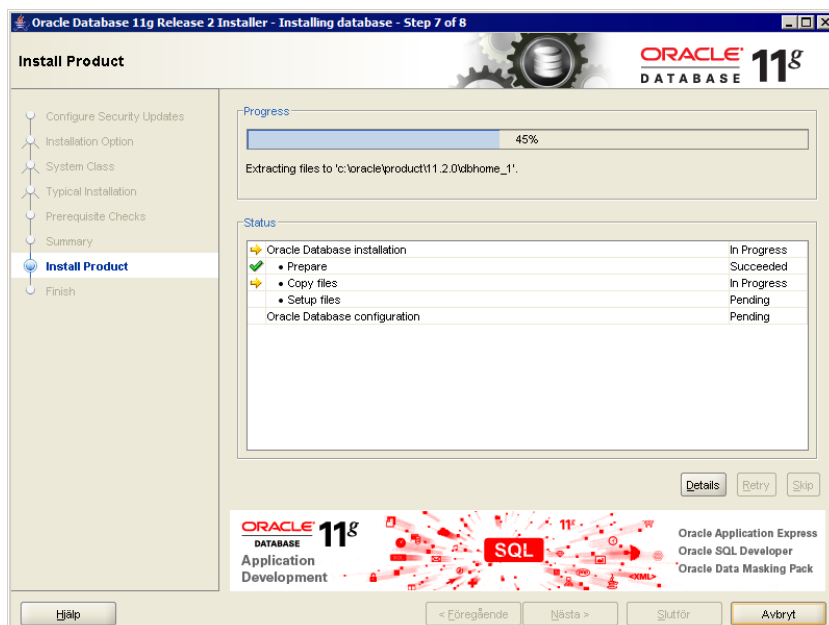
Next, choose a Desktop class installation:



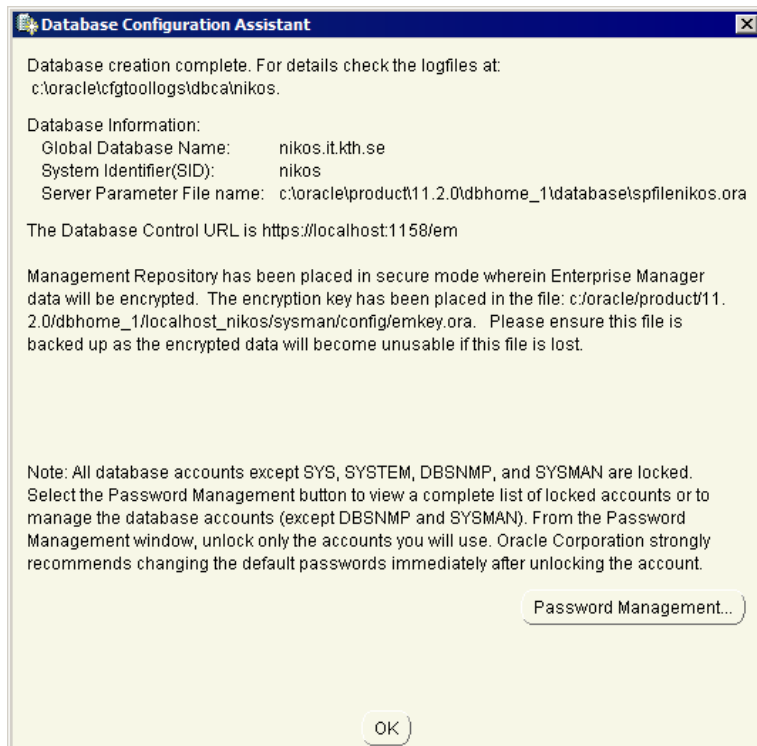
Now, choose where Oracle should be installed and specify to use the standard edition and the default character set. Specify a unique global database name and set the system password. You will need to use this password later, to connect to the database.



Oracle will now check that everything is ok and present a summary before the installation can begin. Press "Finish" or "Slutför" and the installation will begin. The installation process will take a while.

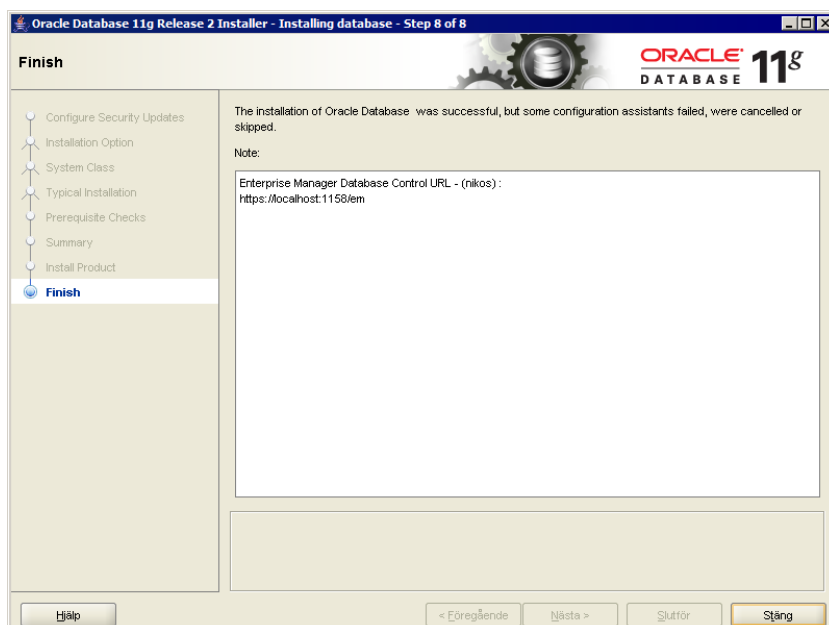


Depending on the platform some errors may be reported by the wizard. Just ignore them. Eventually, the installation will be completed and the wizard will present a summary.



In this summary, you will notice the Database Control URL. This URL opens the Oracle Enterprise Manager where you can manage your database instance. You can manage performance settings, users, tables, views, triggers, etc. Log in as SYSTEM with the password you specified earlier and take a look.

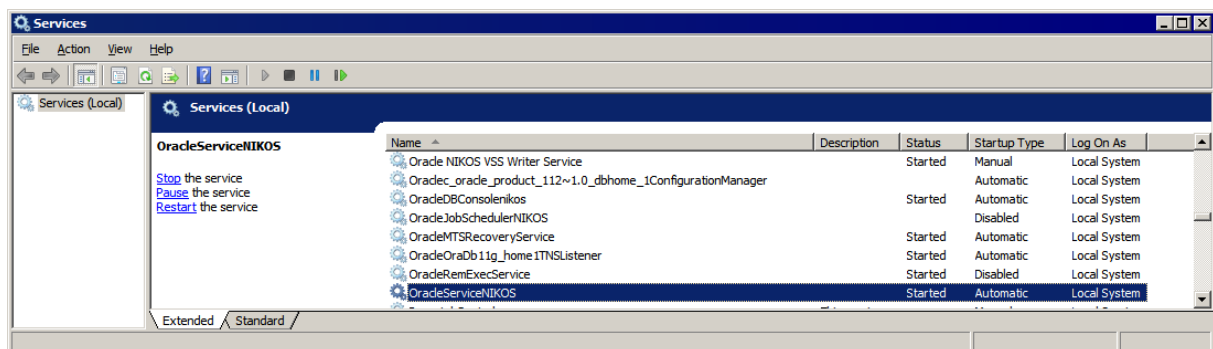
After pressing OK on the above summary window, the main wizard window will reach Step 8 "Finish" and Oracle will have been installed.



In the start menu, you will find several shortcuts to different Oracle tools. We will use SQL Developer. SQL Developer is bundled with Oracle, but requires Java. Depending on your configuration (32-bit or 64-bit) you may need to download SQL Developer separately and replace the bundled version. When starting SQL Developer for the first time, you will also have to point to java.exe (the one in the JDK/bin directory). It is recommended to use the latest version. The version used in this introduction is 3.1.07.

2.1.1 Services

During the installation several Windows services were created:

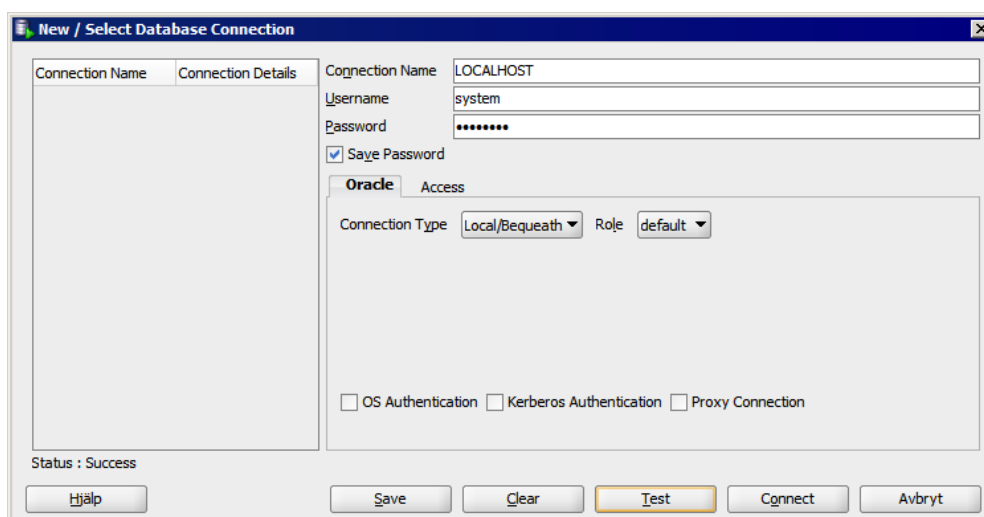


The one called OracleService is the main service for the database instance.

2.2 SQL Developer

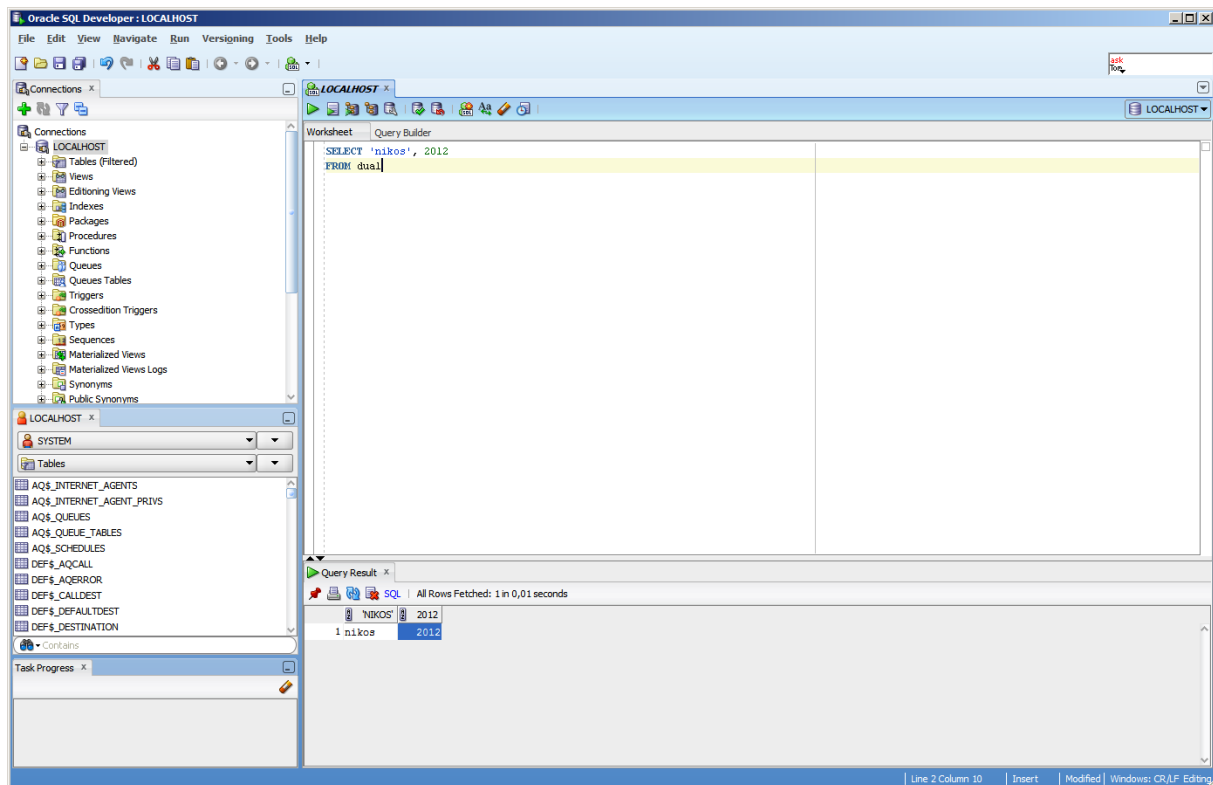
SQL Developer is a tool for performing common database tasks easier. It provides several wizards for database object creation, code completion for SQL, monitoring tools, etc.

When you start SQL Developer, you need to create a connection or use an existing one. To work locally, create a local connection and log in as SYSTEM with your password.



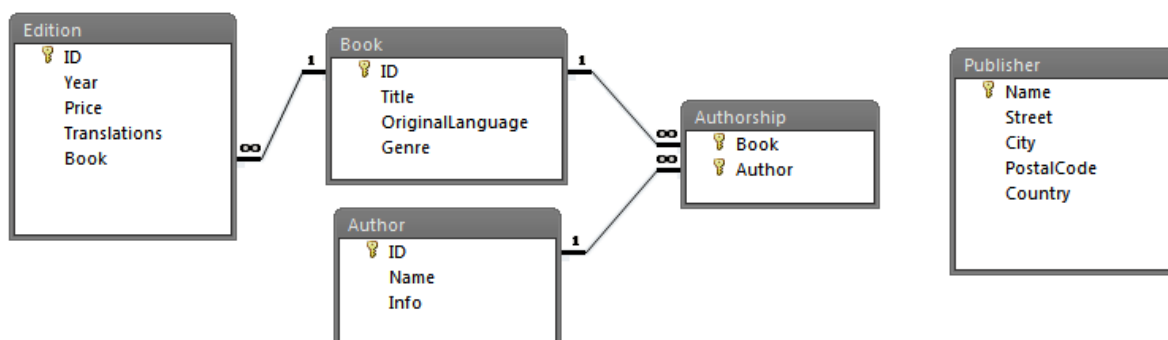
Once the connection has been created, you will see several panes. On the left you should have the connections pane where you can explore all the objects of your connection. On the right, you have one or more worksheets where you can write SQL commands or scripts. Each

worksheet is associated to one connection. Below the worksheet area, there is the result area (or at least it will show up there after you execute a command). The placement of each pane is freely configurable, so it could look like this:



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 VARCHAR2 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/oraclexml/>

The script can be run through SQL Developer. It creates a schema called bookdb as well as all the tables and other relevant objects in this schema.

3.1 XML data type

Oracle 11g R2 has an XML data type called XMLTYPE. This data type can be used with and without an XML Schema, thus allowing for validation or no validation. There is no support for DTD. Any schema to be used must be already registered. The validation performed is only structural. Full validation can be done with the function XMLIsValid which can be used in a constraint in order to ensure that only fully validated documents make it into the database.

In the provided database script, there is no validation. On the other hand, the XML data type always checks that the input is well-formed.

4 Examples

In this chapter we will go through some examples of SQL/XML in Oracle and some examples that use Oracle specific XML features. All the examples in this chapter assume that the database has been created and that the default schema is bookdb.

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 want 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. Oracle'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 "Name"),
                                     info)))
FROM author
```

The result looks like this:

```
<Authors>
  <Author Name="John Craft"><Info><Email>jc@jc.com</Email>
  <Country>England</Country><YearOfBirth>1948</YearOfBirth></Info></Author>
  <Author Name="Arnie Bastoft"><Info><Email>bastoft@frei.at</Email>
  <Country>Austria</Country><YearOfBirth>1971</YearOfBirth></Info></Author>
  <Author Name="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 may 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 are 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" RETURNING CONTENT)
FROM Author
```

In this case the XQuery expression is 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"). In Oracle, the keywords RETURNING CONTENT are required and there is no alternative. The result has two columns:

John Craft	England
Arnie Bastoft	Austria
Meg Gilmand	Australia
Chris Ryan	France
Alan Griff	USA
Marty Faust	USA
...	

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

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

This produces the same result as the previous statement, but finds the country in two steps. The nested XMLQUERY function in Oracle returns an XML value, but its root node is not the Country element, even though it appears to be. In the outer XMLQUERY call, we must therefore go from the root to the Country element. This behaviour is due to the RETURNING CONTENT keywords that create a document node as the root node of the result. RETURNING SEQUENCE (which is not yet implemented by Oracle) would let the Country element be the root node.

XMLQUERY can also be used to create XML from a string. So XMLQUERY('<X>123</X>' RETURNING CONTENT) 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.column_value
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 resulting column of the XMLTABLE function is called column_value when the keyword COLUMNS is not present.

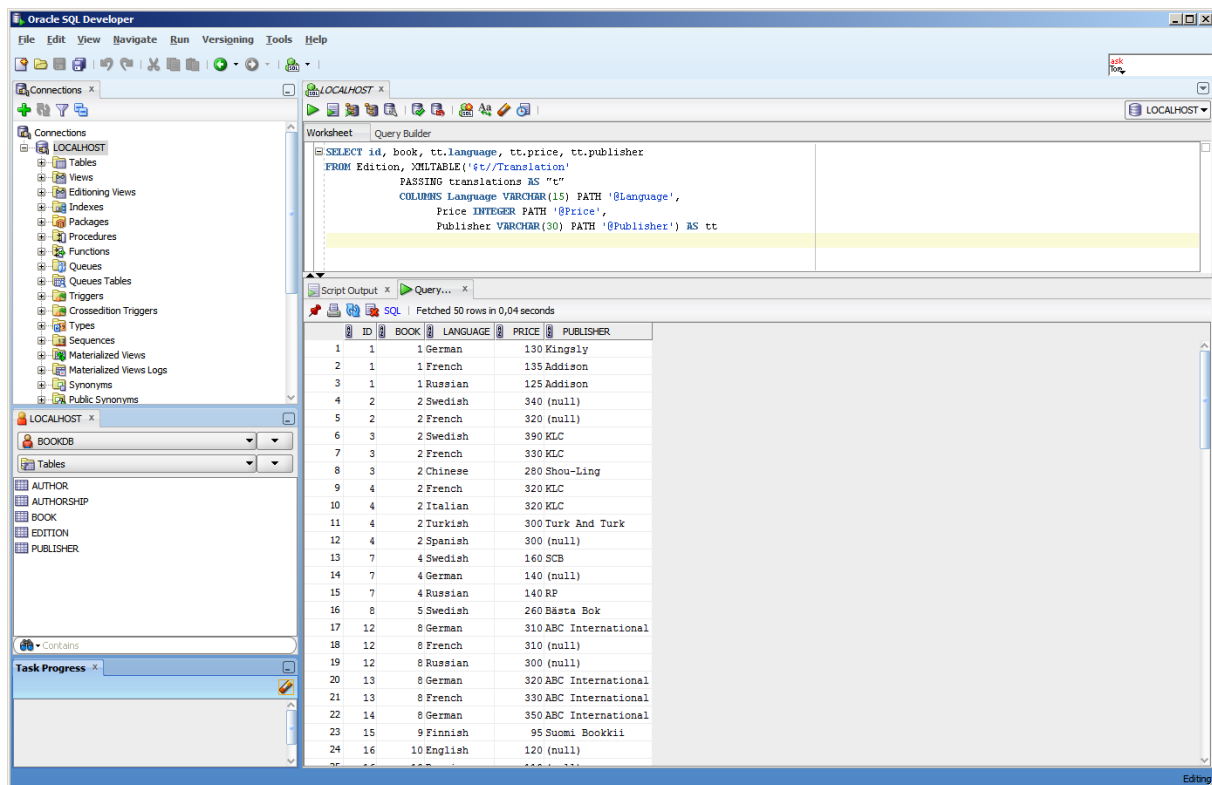
Just as with XMLQUERY the result of the XMLTABLE function is also wrapped inside a document node. This can be illustrated with the following example, where in order to access the Language attribute, we must go from the root (the document node) to the Translation element node, to the attribute node:

```
SELECT id, book, XMLQUERY('/Translation/@Language'
                        PASSING tt.column_value RETURNING CONTENT)
FROM Edition, XMLTABLE('$t//Translation' PASSING translations AS "t") AS tt
```

Using the keyword COLUMNS could also break down this 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
Oceans on Earth
Contact
Music Now and Before
Musical Instruments
Le chateau de mon pere

4.6 Method/Function Extract and function ExtractValue

Oracle's method/function Extract and function ExtractValue can be used with XML objects (values of the data type XMLTYPE) to retrieve XML fragments or values. They are deprecated and the SQL/XML function XMLQUERY should be used instead. Here are some examples anyway.

If we want to get the country of each author we could use any of the following:

```
SELECT name, a.info.extract('//Country/text()'), Extract(info, '//Country/text()'),  
ExtractValue(info, '//Country')  
FROM author a
```

The extract method and the Extract function return XML, so it is the exact node that is returned. The ExtractValue function returns the value of the node and not the node itself. The extract method (and any other XMLTYPE method) requires that the column containing the XML object be qualified with an alias. Both of the following will therefore return an error (even though at plain sight they appear to be correct).

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

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

Another important thing to remember is that the result of extract (method or function) will be a new XML document with a document node as its root. This is the same behaviour as for XMLQUERY which we discussed earlier.

4.7 Method/Function ExistsNode

The function ExistsNode and the corresponding XMLTYPE method existsNode can be used to check the existence of a node for a specific XPath expression. They return 1 if the result is not empty and 0 if the result is empty. We could for example find all the authors from Sweden. Any one of the two conditions is enough.

```
SELECT name  
FROM author a  
WHERE ExistsNode(info, '//Country[. = "Sweden"]') = 1  
OR a.info.existsNode('//Country[. = "Sweden"]') = 1
```

The result is the following:

```
Jakob Hanson  
Marie Franksson
```

This function/method is deprecated and the SQL/XML function XMLEXISTS should be used instead.

4.8 XMLColAttVal

XMLColAttVal is a function that transforms one or more columns to an XML fragment. For each column an element "column" is created and the value becomes the content. The column's name is stored as the value of the attribute "name". The same result could of course be produced with the standard publishing functions of SQL/XML. Here is an example:

```
SELECT XMLCOLATTVAL(name, country, city)
FROM Publisher
```

This produces the following result:

```
<column name = "NAME">ABC International</column>
<column name = "LAND">Germany</column>
<column name = "CITY">Berlin</column>

<column name = "NAME">Addison</column>
<column name = "LAND">France</column>
<column name = "CITY">Toulouse</column>
...
```

We could of course add a root element with XMLELEMENT. The following statements will have the same result.

```
SELECT XMLELEMENT(NAME "Publisher", XMLCOLATTVAL(name, country, city))
FROM Publisher
```

```
SELECT XMLELEMENT(NAME "Publisher",
    XMLELEMENT(NAME "column", XMLATTRIBUTES('NAME' AS "name"), name),
    XMLELEMENT(NAME "column", XMLATTRIBUTES('COUNTRY' AS "name"), country),
    XMLELEMENT(NAME "column", XMLATTRIBUTES('CITY' AS "name"), city))
FROM Publisher
```

4.9 DML for XML

Oracle provides several functions for manipulating XML with operations similar to SQL INSERT, UPDATE and DELETE. There is one function for update called UpdateXML, one function for delete called DeleteXML and several functions for insert called InsertChildXML, InsertChildXMLBefore, InsertChildXMLAfter, InsertXMLBefore, InsertXMLAfter and AppendChildXML. All these functions work based on the same principal. They take an XML value as a parameter and return a changed version of it. The original XML value is not affected. That means that the column containing the original XML value has to be updated with SQL UPDATE if the change is to become permanent. In this section we look at some examples. For more details on these functions refer to the documentation.

4.9.1 UpdateXML

The function UpdateXML is fairly simple. It takes three parameters: the original XML value, an XPath expression identifying the node whose value is to be changed, and the new value. If the XPath expression matches more nodes, then all of them will be updated. If the XPath expression does not match any nodes, the result will be identical to the original XML value.

Let's say we want to change the e-mail address of the author Carl Sagan. We can use the following UPDATE statement:

```
UPDATE author
SET info = UPDATEXML(info, '//Email/text()', 'carl@sagan.info')
WHERE name = 'Carl Sagan'
```

This statement identifies the correct row in the author table and replaces the value of the column info with a new value generated by the function UpdateXML. The function takes the current value of the column info and replaces the text node with the new value. UpdateXML always replaces the entire node so UPDATEXML(info, '//Email', 'carl@sagan.info') would instead have removed the element node and created a text node.

The third parameter can be a string value or XML. If the node to be updated is an attribute node, then the third parameter provides the new value for the node, but the node itself is not replaced, just its value.

Here is another way to achieve the same result as with the previous statement:

```
UPDATE author
SET info = UPDATEXML(info, '//Email', XMLELEMENT(NAME "Email", 'carl@sagan.info'))
WHERE name = 'Carl Sagan'
```

This is obviously unnecessarily complex, since it recreates the entire element node instead of just switching the text node.

4.9.2 DeleteXML

Removing a node is done with the function DeleteXML. It deletes any nodes matching the specified XPath expression. We could for example remove the Email element node from Carl Sagan's info XML (which would violate the XML Schema, but we can ignore that right now).

```
UPDATE author
SET info = DELETXML(info, '//Email')
WHERE name = 'Carl Sagan'
```

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

4.9.3 Insert and Append functions

The reason there are many functions for adding nodes, is that the relative position of the added nodes needs to be specified. You may want to add a node before another node or after another node, or perhaps as the last child node. Let's look at some examples. If we want to add a Website element for Carl Sagan, we may use the function AppendChildXML:

```
UPDATE author
SET info = APPENDCHILDXML(info, '//Info',
                          XMLTYPE('<Website>www.carlsagan.com</Website>'))
WHERE name = 'Carl Sagan'
```

This statement adds the new element node as the last child of the node matching the XPath expression specified in the second parameter. In the previous statement we created an XMLTYPE value from a string representation. Another way would be to use the XMLELEMENT function:

```
UPDATE author
SET info = APPENDCHILDXML(info, '//Info',
                          XMLELEMENT(NAME "Website", 'www.carlsagan.com'))
WHERE name = 'Carl Sagan'
```

If we would prefer to add the Website element directly after the Email element, we can use the function InsertXMLAfter:

```
UPDATE author
SET info = INSERTXMLAFTER(info, '//Email',
                          XMLELEMENT(NAME "Website", 'www.carlsagan.com'))
WHERE name = 'Carl Sagan'
```

The created node becomes the next sibling to the node specified by the XPath expression. If that XPath expression matches several nodes, then a new node will be added after each of them.

If we want to add an attribute node, the function InsertChildXML may be the best choice. Let's say that we want to add an attribute Launched to the Website element that we created before and specify that Carl Sagan's website was launched in 1997. We could do that with the following statement:

```
UPDATE author
SET info = INSERTCHILDXML(info, '//Website', '@Launched', 1997)
WHERE name = 'Carl Sagan'
```

The third parameter specifies the name of the node to be created. The at sign (@) indicates that the node to be created shall be an attribute node. The fourth parameter specifies the value of the new node. It can be of any type and it will be adapted to XML. If it is a date, time or decimal, the current locale may affect the resulting layout.

4.10 Other XMLTYPE methods

Oracle has a number of extra methods that can be used on XMLTYPE objects. We have already discussed some of them in previous sections. The methods getStringVal, getBLOBVal and getCLOBVal are basically serialization methods that return the XMLTYPE object as a String, BLOB and CLOB respectively. The method getNumberVal returns the value of the object as a number. The object must have a value that is possible to convert to a number. The XMLTYPE object must be a text node or attribute node. Here is a simple example:

```
SELECT XMLQUERY('99' RETURNING CONTENT).getNumberVal() + 1
FROM DUAL
```

The result is 100. XMLQUERY returns 99 as an XMLTYPE object and the method retrieves its value as a number. The following does not work because the first part of the plus operation is not a number:

```
SELECT XMLQUERY('99' RETURNING CONTENT) + 1
FROM DUAL
```

There are also some methods on the XMLTYPE that can return information about the XML object. The method getRootElement returns the name of the root element unless the XML object is a fragment and then the result is NULL. The method getSchemaURL returns the URL of the XML Schema associated with the XML object. The method isFragment can be used to check if an XML object is an XML fragment or an XML document. The method returns 1 or 0. The method isSchemaValid can be used to validate the XML object given an XML Schema. The method isSchemaBased checks if the object is associated with an XML Schema. The method isSchemaValidated checks if the object has already been validated based on its associated XML Schema. It does not distinguish between not being valid and not having been validated. The following statement uses some of these methods:

```
SELECT a.info.isSchemaBased(), a.info.isSchemaValidated(),
       a.info.getRootElement(), a.info.isFragment()
FROM author a
WHERE id = 1
```

The result is 0,0,'Info',0, which means that the XML object is not schema based, it has not been validated, its root element is Info and it is not a fragment (it is an XML document).

There is also a method called transform, which can be used to apply an XSLT to the XML object. This method is similar to the function XMLTransform, so they are both described in a separate section.

4.11 XMLTransform

If we want to use XSLT to transform XML objects, we have two options. There is a function XMLTransform and a method transform. Both have the same result. The function requires that the XML value to be transformed is specified as a parameter, while the method operates on a specific XML object. 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 XMLTRANSFORM(info,
'<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 and returns the XML value serialized. The method is a little less flexible. It requires that the XSLT is provided as an XMLTYPE value, which is quite easy to do. The following statement produces the same result as the one using the function.

```
SELECT a.info.transform(
XMLTYPE('<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 a
WHERE name = 'Carl Sagan'
```

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.

4.12 XQuery function ora:view

In some cases, we may want to access relational data from within XQuery. The function ora:view makes this possible. It takes the name of a table or view as a parameter and returns the content as an XML fragment with one ROW element per row and one subelement for each column. The element names will be in upper case by default. We could, for example, access all the countries of publishers (in an XQuery statement) using the following statement:

```
SELECT XMLQUERY('for $c in distinct-values(ora:view("publisher"))//COUNTRY)
           return element Country {$c}'
       RETURNING CONTENT)
FROM DUAL
```

The result is an XML fragment with one Country element for each unique country:

```
<Country>Austria</Country>
<Country>Belgium</Country>
<Country>China</Country>
<Country>England</Country>
...
```

5 Epilogue

Oracle has been moving closer to the SQL standard with each new version. Many of the Oracle specific functions have been deprecated and replaced by standard constructs. It is therefore essential to follow the release information of each version. Many of the Oracle specific features described here will probably be replaced in the years to come. The XQuery Update Facility will probably be available in future version. In the examples in the previous chapter we looked at some of the features that are available in Oracle 11g R2. 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