Introduction to Oracle & XML

Version 1.4.0
January 2025

nikos dimitrakas

Table of contents

1 INTRODUCTION 3

1.1 ORACLE 3
1.2 PREREQUISITES 3
1.3 STRUCTURE 3

2 ORACLE DATABASE 23AI 3

2.1 INSTALLATION 3
2.1.1 Services 5
2.2 SQL DEVELOPER 6
2.2.1 XML in SQL Developer 7

3 SAMPLE DATA 8

3.1 XML DATA TYPE 9

4 EXAMPLES 9

4.1 XMLELEMENT, XMLFOREST, XMLATTRIBUTES 9
4.1.1 EVALNAME 11
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 XQuery transform 17
4.9.2 DML functions 18
4.9.3 insert 18
4.9.4 delete 19
4.9.5 rename 19
4.9.6 replace 21
4.9.7 UpdateXML (deprecated) 21
4.9.8 DeleteXML (deprecated) 22
4.9.9 Insert and Append functions (deprecated) 22
4.10 OTHER XMLTYPE METHODS 24
4.11 XMLTRANSFORM 25
4.12 XQUERY FUNCTION ORA:VIEW 26
4.13 NATIVE XQUERY 27

5 EPILOGUE 29

1 Introduction

This compendium gives a short introduction to Oracle Database 23ai and its facilities for database administration. We discuss installing Oracle Database 23ai 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 Database 23.6 for Windows on Windows 11, but they should work in a similar manner on any platform. It is recommended that you use Oracle Database 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 Database is one of the major DBMSs 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 23) 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 Database 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 Database (like SQL Plus or TOAD) but the recommended tool is SQL Developer.

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.

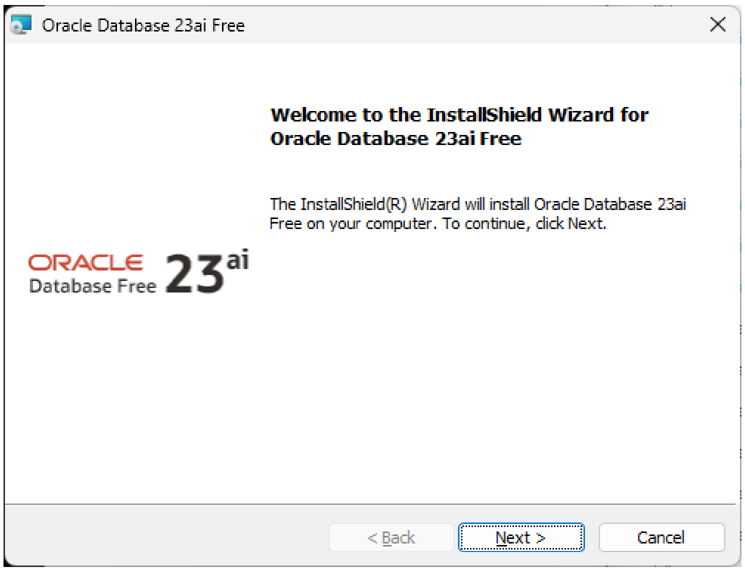
2 Oracle Database 23ai

Oracle Database 23ai is available for free by Oracle for non-commercial use. The installation file is a zip-file 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 23.6 for Windows x64. In order to download the installation files, you may need to create a free account.

Unzip the file in a folder prior to initiating the installation. Make sure the folder name does not contain spaces or other special characters. Run setup.exe to start the installation. Wait a while until the following window appears:

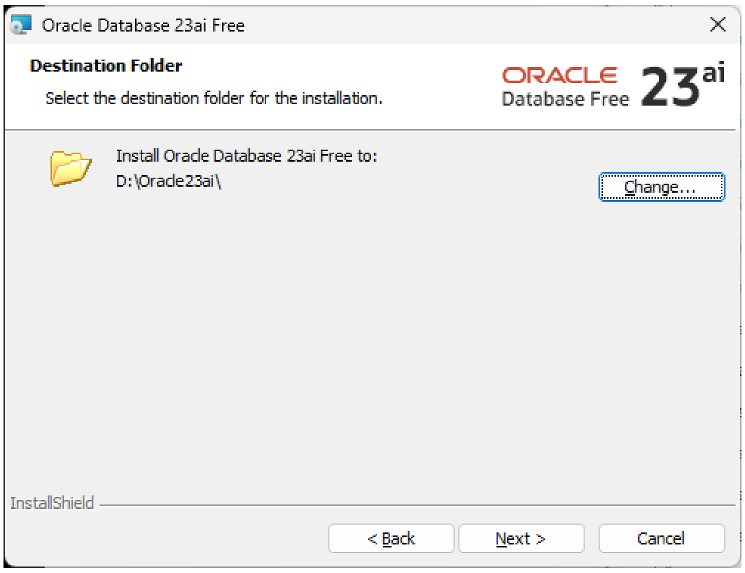

Oracle Database 23ai Free

Welcome to the InstallShield Wizard for Oracle Database 23ai Free

The InstallShield(R) Wizard will install Oracle Database 23ai Free on your computer. To continue, click Next.

Press "Next", accept the license agreement and press "Next" again!

Select a suitable destination folder and press "Next"!

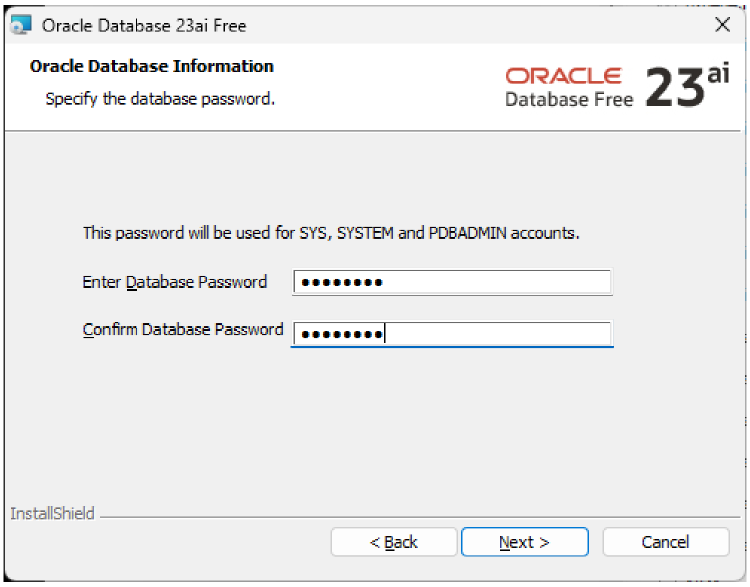

Oracle Database 23ai Free

Destination Folder

Select the destination folder for the installation.

Install Oracle Database 23ai Free to:
D:\Oracle23ai\

Specify a system password and press "Next"! You will need to use this password later, to connect to the database.


Oracle Database 23ai Free
Oracle Database Information
Specify the database password.
ORACLE Database Free 23ai

This password will be used for SYS, SYSTEM and PDBADMIN accounts.

Enter Database Password ••••••••

Confirm Database Password ••••••••

InstallShield
< Back
Next >
Cancel

Press "Install" and wait for the installation to complete! Then just press "Finish"!

At this point it can be a good idea to restart Windows, even though it is not supposed to be required.

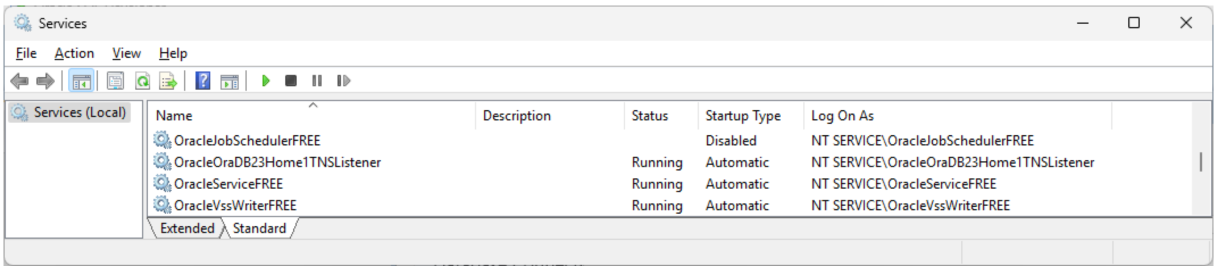
In the start menu, you will find several shortcuts to different Oracle tools. We will use SQL Developer. SQL Developer can be downloaded from Oracle's website and since it requires java, it is available in two versions, with and without Java. Download the latest SQL Developer with JDK (the version without JDK does not always work). The recommended version as of January 2025 is 24.3.1. Some older versions (18.x) have problems displaying XML and many versions (17.x – 24.x) have a bug making Local connections impossible. TNS-connections can be used instead.

SQL Developer does not require any installation. Just extract the files of the downloaded zip-file in a directory and run the sqldeveloper.exe. We will see in section 2.2 how to create a connection and work inside SQL Developer.

The version used in this introduction is SQL Developer 24.3.0 with JDK 17 included.

2.1.1 Services

During the installation several Windows services were created:



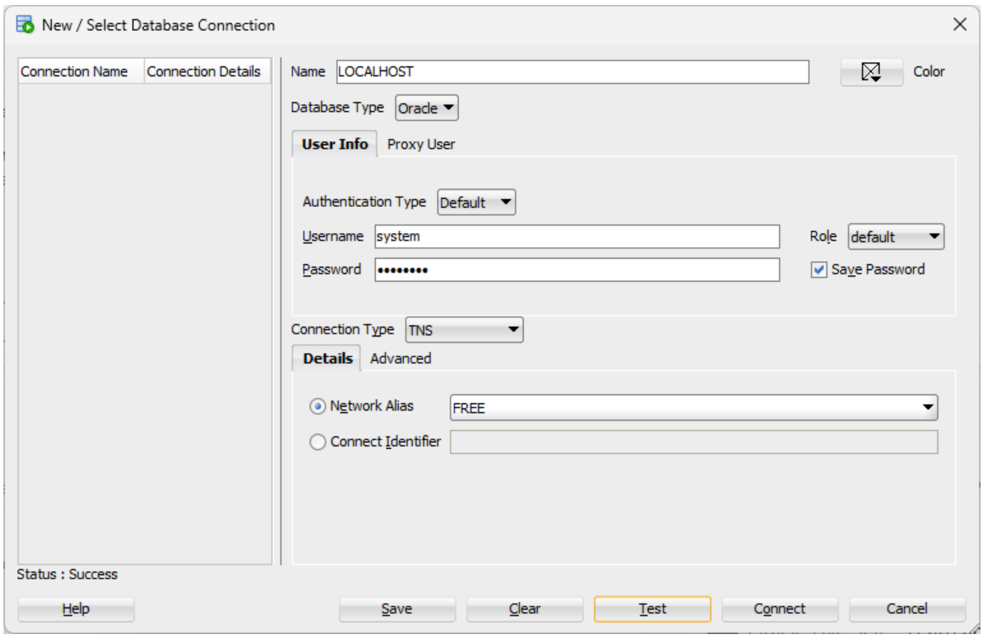
Name	Description	Status	Startup Type	Log On As
OracleJobSchedulerFREE			Disabled	NT SERVICE\OracleJobSchedulerFREE
OracleOraDB23Home1TNSListener		Running	Automatic	NT SERVICE\OracleOraDB23Home1TNSListener
OracleServiceFREE		Running	Automatic	NT SERVICE\OracleServiceFREE
OracleVssWriterFREE		Running	Automatic	NT SERVICE\OracleVssWriterFREE

The one called OracleServiceFREE 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.

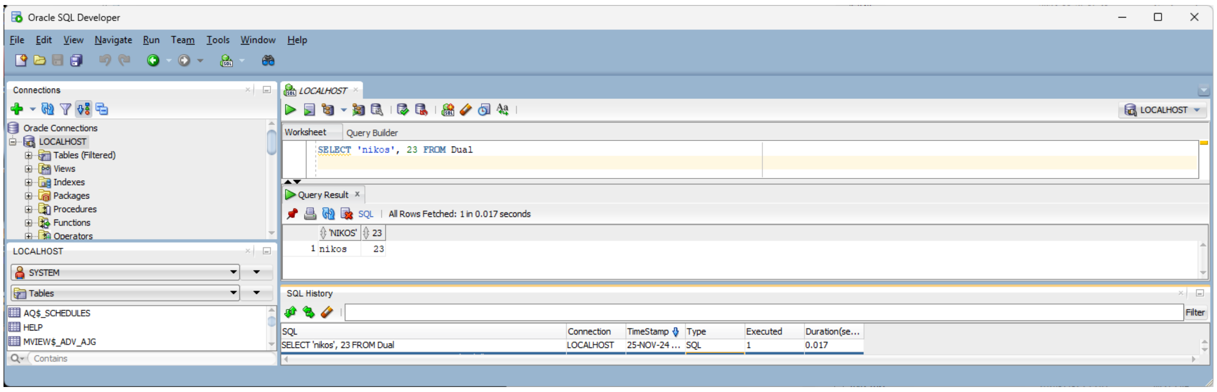
When you start SQL Developer, you need to create a connection or use an existing one. If the Welcome page is shown, a connection can be created by selecting "Create a Connection Manually". Otherwise, press the green plus sign on the left side under "Connections". There are many types of connections (Basic, Local, TNS and more). Create a connection of type TNS (Basic and Local do not always work)! Give the connection a name, specify the authentication type as Default, the username as system, the password to the password specified during the installation, the role as default and select the network alias FREE! You may choose to save the password, otherwise you will need to specify it every time you establish the connection. The settings should look something like this:


New / Select Database Connection

Name: LOCALHOST
Database Type: Oracle
Authentication Type: Default
Username: system
Role: default
Password: ••••••••
Save Password: ✓
Connection Type: TNS
Network Alias: FREE
Status : Success

Press "Save" and the connection will be added to the connection list on the left side! Press "Connect" to establish a connection and return to the main window!

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:



```
SELECT 'nikos', 23 FROM Dual
```

All Rows Fetched: 1 in 0.017 seconds

```
  'NIKOS' 23
1 nikos   23
```

SQL	Connection	TimeStamp	Type	Executed	Duration(se…
SELECT 'nikos', 23 FROM Dual	LOCALHOST	25-NOV-24 …	SQL	1	0.017

Different panes can be shown/hidden with the menus under View. Panes can be moved around with drag and drop.

There are many things that can be configured in SQL Developer under Tools > Preferences. One thing that may be important to fix is the date format. In the Preferences window under Database > NLS, things like Date Format, Decimal Separator, etc. can be configured. It is recommended you set the Date Format to YYYY-MM-DD.

When running scripts or XQuery in SQL Developer, the result is presented in text form. How wide the result is and how much of each value is shown depends on different settings. To configure appropriate values, use the following commands and adjust the numbers:

```
SET LONG 1000;
SET LONGCHUNKSIZE 100;
SET PAGESIZE 100;
```

The first one decides the number of characters per value to show in the result.
The second one decides the length of each line in the result.
The third one decides the number of rows in the result before repeating the headings.

At this point you may also want to change the password expiration policy of Oracle. The default is that all passwords expire after six months. Run the following command in order to remove the password expiration:

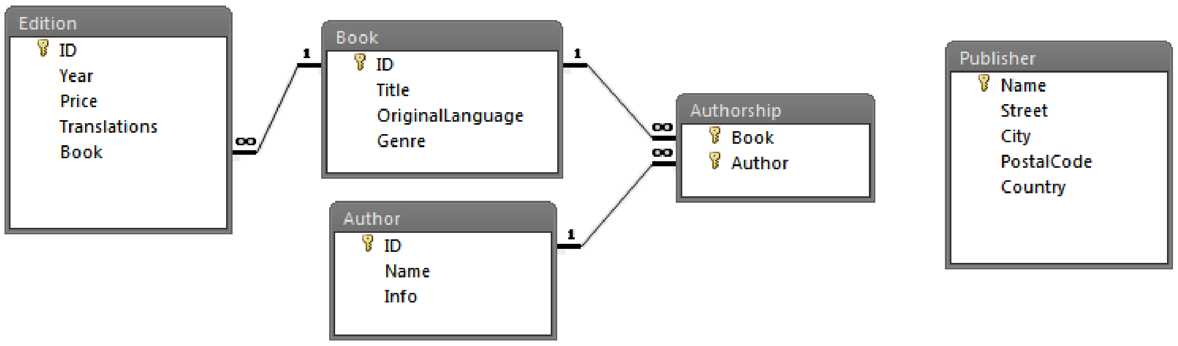
```
ALTER PROFILE DEFAULT LIMIT PASSWORD_LIFE_TIME UNLIMITED;
```

2.2.1 XML in SQL Developer

SQL Developer has some built-in functionality for XML, for example to indent and color-code XML. Whenever an XML value is part of a result, it will be possible to open each XML value in a separate editor, format it and get some syntax checking.

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 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 Database 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 "Name"), info)
FROM Author
```

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

```
<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>
```

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>
```

4.1.1 EVALNAME

Oracle allows for the element names and attribute names to be dynamically generated from values. That is done by using the keyword EVALNAME instead of NAME in the function XMLELEMENT or by adding the keyword EVALNAME after AS in the functions XMLATTRIBUTES and XMLFOREST. The following example would create elements based on the value of the column country:

```
SELECT XMLELEMENT(EVALNAME country, name) FROM Publisher
```

The result would look like this:

```
<Germany>ABC International</Germany>
<Germany>Deutche Ferlage</Germany>
<France>Addison</France>
<Italy>Aurora Publ.</Italy>
<England>Benton Inc</England>
<England>McManus Publishing</England>
<Sweden>Bästa Bok</Sweden>
…
```

Unfortunately, Oracle does not check that the values used as element names are valid as element names. So, it is up to us to make sure that special characters are eliminated or replaced. The previous statement returns the following, syntactically incorrect, XML:

```
<United States>Marco Polo</United States>
```

Using EVALNAME with our sample database may seem inappropriate, but if we had a database of a higher abstraction, EVALNAME could be very useful.

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"/>
      <Publisher Name="McManus Publishing" City="London"/>
   </Country>
   <Country Name="Finland">
      <Publisher Name="Suomi Bookkii" City="Helsinki"/>
   </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 Database, 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 SQL Developer may 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 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 support by Oracle Database according to the documentation, but appears to work) would let the Country element be the root node.

If SQL Developer shows (XMLTYPE) instead of the serialized version, it may be necessary to serialize the result with XMLCAST or if it’s a text node, by using the XPath function string().

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.

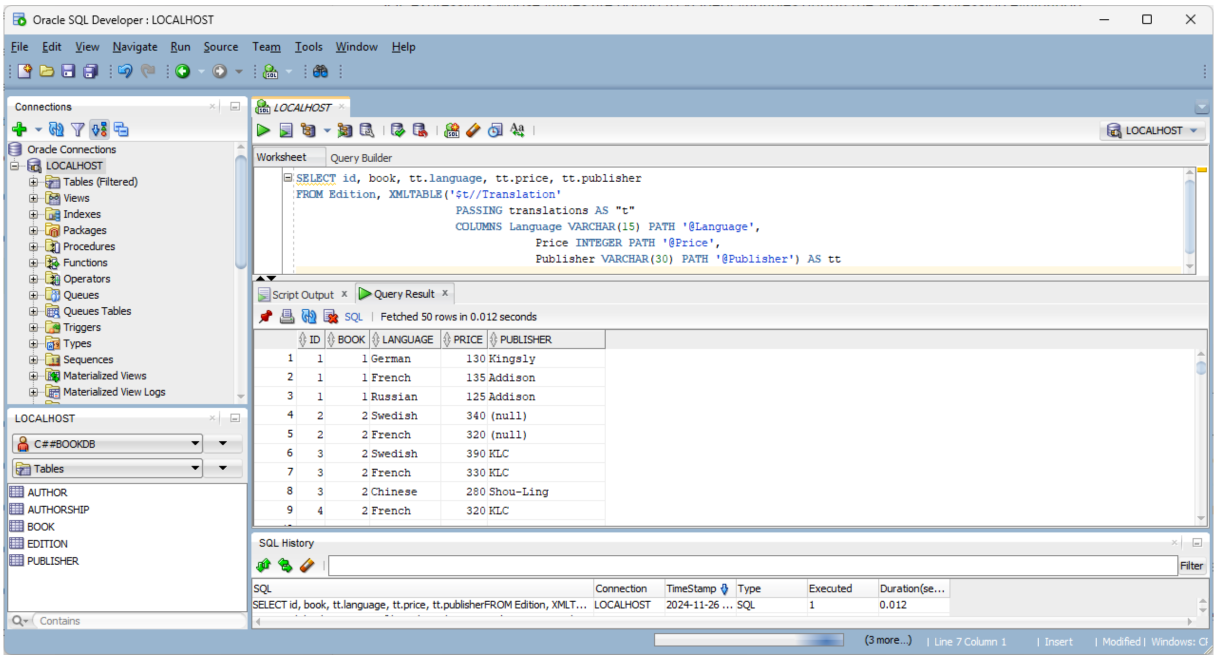
Just as with XMLQUERY the result of the XMLTABLE function is also wrapped inside a document node. This can be illustrated with the following example, where in order to access the Language attribute, we must go from the root (the document node) to the Translation element node, to the attribute node:

```
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:



```
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
```

Fetched 50 rows in 0.012 seconds

	ID	BOOK	LANGUAGE	PRICE	PUBLISHER
1	1	1	German	130	Kingsly
2	1	1	French	135	Addison
3	1	1	Russian	125	Addison
4	2	2	Swedish	340	(null)
5	2	2	French	320	(null)
6	3	2	Swedish	390	KLC
7	3	2	French	330	KLC
8	3	2	Chinese	280	Shou-Ling
9	4	2	French	320	KLC

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:

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

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 method extract and the function Extract return XML, so it is the exact node that is returned. The function ExtractValue returns the value of the node and not the node itself. The method extract (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 Database 12 was the first version of Oracle Database to support (in part) the XQuery Update Facility. In earlier versions of Oracle Database, XML could be manipulated only with Oracle specific functions. In this section we look at some examples using both techniques. But first a general introduction to both techniques. Starting with version 12 the Oracle specific functions have been deprecated and are likely to be removed in some future version.

4.9.1 XQuery transform

The XQuery 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 RETURNING CONTENT)
WHERE …
```

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

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 our variable. The modify-expression can be any of the following expressions: delete, insert, rename, or replace. In the following sections we will look at some examples that use the different modify expressions.

4.9.2 DML functions

Oracle Database 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.

If we compare these functions with the XQuery Update Facility, UpdateXML corresponds to replace, DeleteXML corresponds to delete, Insert* and AppendChildXML correspond to insert, and nothing corresponds to rename (we must instead delete the node and insert a new one).

The XML DML functions are deprecated since Oracle Database 12 and will likely be removed in a future version.

4.9.3 insert

When using a transform statement to add nodes to an XML value, you need to use an "insert node" expression. The placement of the new node will be based on an XPath expression and 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('copy $res := $i
                     modify insert node element Website {"www.carlsagan.com"}
                                  as last into $res/Info
                     return $res'
          PASSING info AS "i" RETURNING CONTENT)
WHERE name = 'Carl Sagan'
```

Note that when using an "insert node" expression, the node specified as a position reference for the new node must exist and must exist exactly once. So, exactly one matching node for $res/Info must exist, or an error will be raised.

4.9.4 delete

If we want to remove a node, then we use the "delete node" 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('copy $res := $i
                     modify delete node $res/Info/Email
                     return $res'
          PASSING info AS "i" RETURNING CONTENT)
WHERE name = 'Carl Sagan'
```

If the XPath expression specified after "delete node" 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('copy $res := $i
                     modify insert node element Email {"carlsagan@nasa.gov"}
                                  as first into $res/Info
                     return $res'
          PASSING info AS "i" RETURNING CONTENT)
WHERE name = 'Carl Sagan'
```

4.9.5 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('copy $res := $i
                     modify rename node $res/Info/Country as "BirthCountry"
                     return $res'
           PASSING info AS "i" RETURNING CONTENT)
```

The XPath expression specified after "rename node" 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('copy $res := $trans
                             modify for $t in $res//Translation
                                    return rename node $t as "Version"
                             return $res'
                   PASSING translations AS "trans" RETURNING CONTENT)
```

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

```
UPDATE Author
SET info = XMLQUERY('copy $res := $i
                     modify rename node $res/Info/BirthCountry as "Country"
                     return $res'
           PASSING info AS "i" RETURNING CONTENT)

UPDATE Edition
SET translations = XMLQUERY('copy $res := $trans
                             modify for $t in $res//Version
                                    return rename node $t as "Translation"
                             return $res'
                   PASSING translations AS "trans" RETURNING CONTENT)
```

Note that when using a "rename node" expression, the node to be renamed must exist. This can be easily checked with a WHERE clause in the SQL UPDATE statement.

4.9.6 replace

It is also possible to replace a node with another node or sequence of nodes. A "replace node" 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('copy $res := $info
                     modify replace node $res//Email
                                    with element Skype {"carl.sagan.author"}
                     return $res'
          PASSING info AS "info" RETURNING CONTENT)
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('copy $res := $i
                     modify replace value of node $res/Info/YearOfBirth with 1914
                     return $res'
          PASSING info AS "i" RETURNING CONTENT)
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'
```

As with "insert node" and "rename node", "replace node" expressions may not specify an XPath expression to a node that does not exist or that matches multiple nodes.

4.9.7 UpdateXML (deprecated)

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 table Author 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.

4.9.8 DeleteXML (deprecated)

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 = DELETEXML(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'
```

4.9.9 Insert and Append functions (deprecated)

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 earlier 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 (Dual is a special dummy table with one row that can be used to run the SELECT clause once, without using any real tables):

```
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 the schema name as an optional 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>
…
```

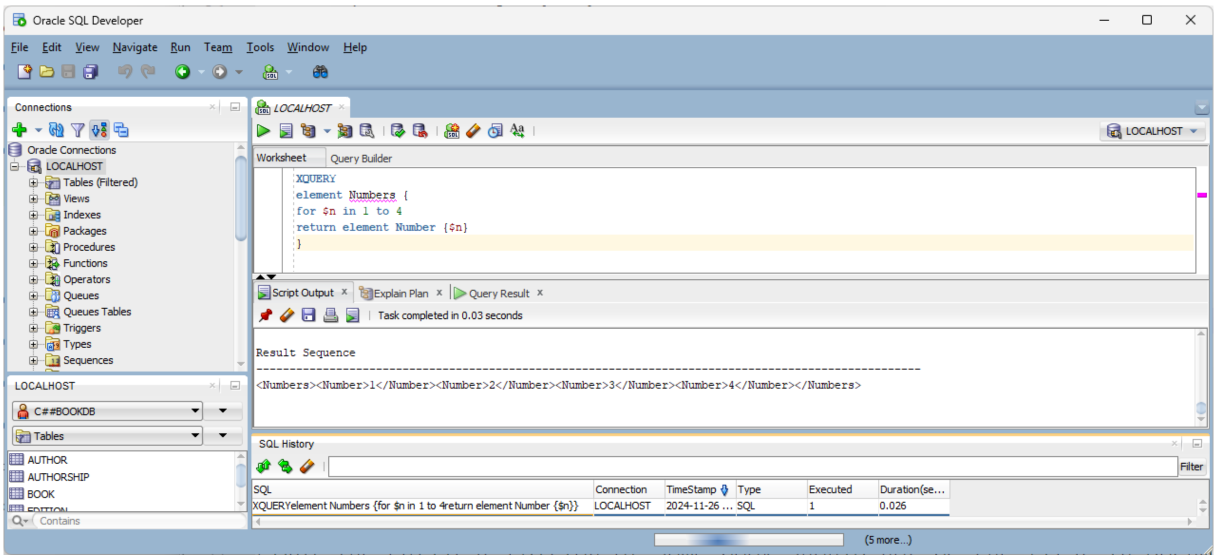
The function ora:view is deprecated. Oracle suggests using fn:collection instead. Its syntax is a little different, but the result is the same. So ora:view(“publisher”) can be replaced by fn:collection("oradb:/C##BOOKDB/PUBLISHER"). Here the table name must be qualified with the schema name and both the schema name and table name are case sensitive. And the prefix oradb: must be used. PUBLIC can be used to refer to the current schema.

4.13 Native XQuery

In many of the previous sections we have seen XQuery being used inside SQL functions used in SQL statements. But it is also possible to execute XQuery statements directly. This is done with the keyword XQUERY. Anything appearing after that keyword will be interpreted as XQuery. For example, a simple XQuery statement like the following could be executed like this:

```
XQUERY
element Numbers {
for $n in 1 to 4
return element Number {$n}
}
```

SQL Developer will always run XQuery as a script and the result will not be shown as a grid:



```
XQUERY
element Numbers {
for $n in 1 to 4
return element Number {$n}
}
```

Task completed in 0.03 seconds

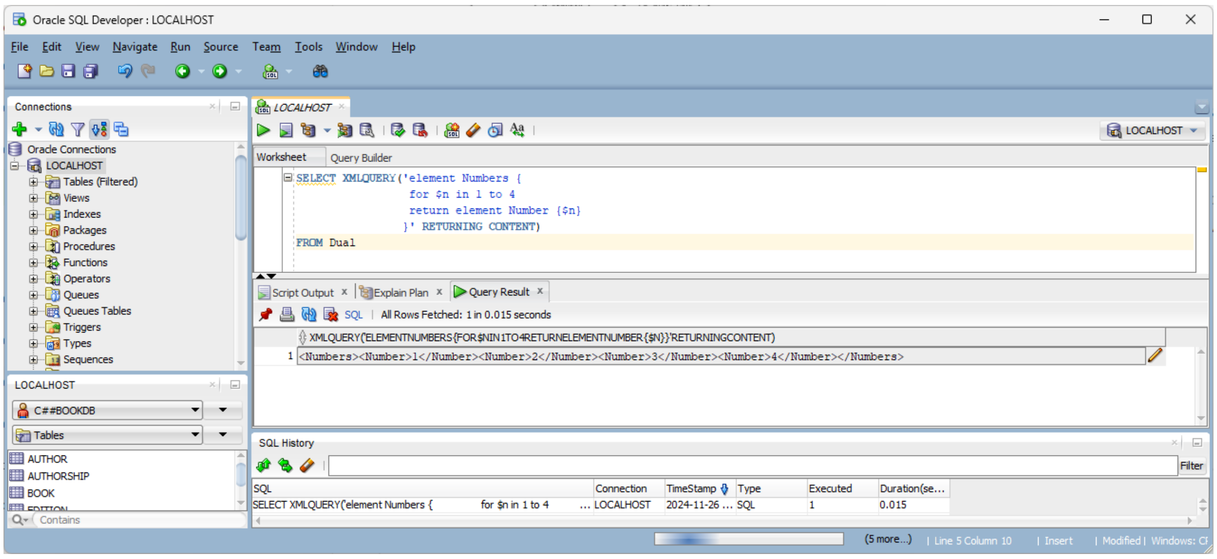
```
Result Sequence
----------------------------------------------------------------------------------------------------
<Numbers><Number>1</Number><Number>2</Number><Number>3</Number><Number>4</Number></Numbers>
```

SQL	Connection	TimeStamp	Type	Executed	Duration(se...
XQUERYelement Numbers {for $n in 1 to 4return element Number {$n}}	LOCALHOST	2024-11-26 ...	SQL	1	0.026

In many cases it can be better to run XQuery with the help of an SQL statement. The following is an SQL statement that runs the same XQuery statement as above:

```
SELECT XMLQUERY('element Numbers {
                    for $n in 1 to 4
                    return element Number {$n}
                    }' RETURNING CONTENT)
FROM Dual
```

Since this is an SQL statement, the result will be a table with one row and one column:



```
SELECT XMLQUERY('element Numbers {
                 for $n in 1 to 4
                 return element Number {$n}
               }' RETURNING CONTENT)
FROM Dual
```

All Rows Fetched: 1 in 0.015 seconds

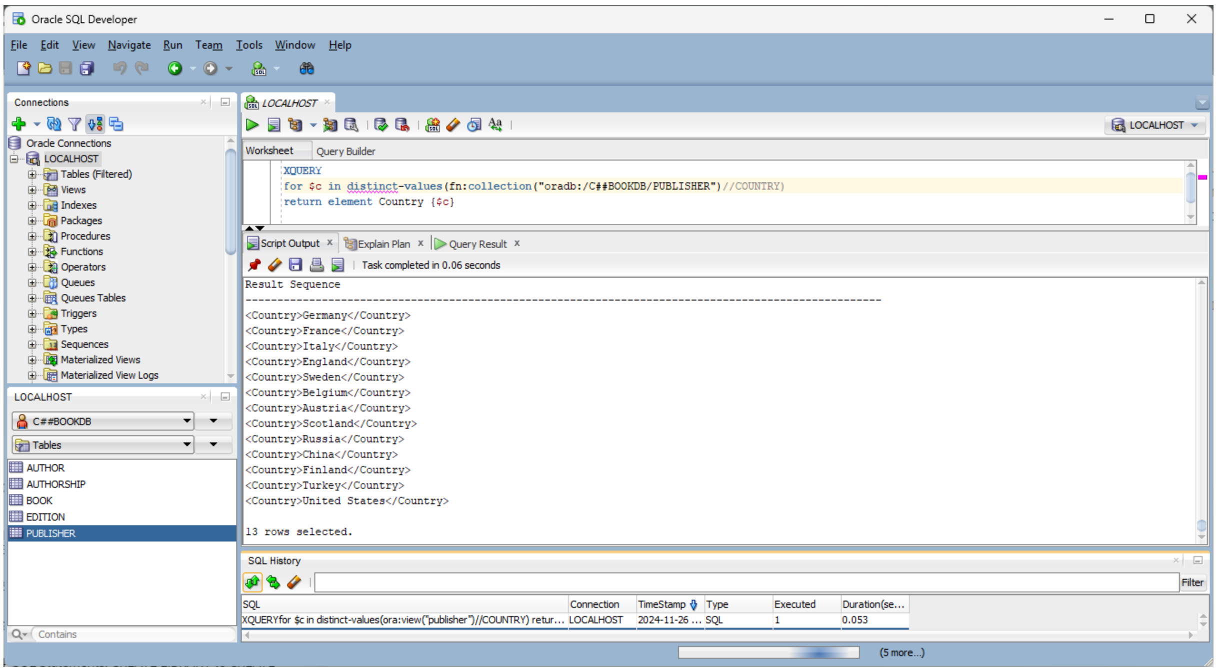
```
   XMLQUERY('ELEMENTNUMBERS{FOR$NIN1TO4RETURNELEMENTNUMBER{$N}}'RETURNINGCONTENT)
1  <Numbers><Number>1</Number><Number>2</Number><Number>3</Number><Number>4</Number></Numbers>
```

SQL	Connection	TimeStamp	Type	Executed	Duration(se...
SELECT XMLQUERY('element Numbers { for $n in 1 to 4 ...	LOCALHOST	2024-11-26 ...	SQL	1	0.015

When we use native XQuery, we may still want to access some data from our tables. The functions discussed in section 4.12 can be used, for example to get all the countries with publishers:

```
XQUERY
for $c in distinct-values(fn:collection("oradb:/C##BOOKDB/PUBLISHER")//COUNTRY)
return element Country {$c}
```

Since this is XQuery and not SQL, the result will be shown as text (the element node sequence serialized):



```
XQUERY
for $c in distinct-values(fn:collection("oradb:/C##BOOKDB/PUBLISHER")//COUNTRY)
return element Country {$c}
```

Task completed in 0.06 seconds

```
Result Sequence
----------------------------------------------------------------------------------------------------
<Country>Germany</Country>
<Country>France</Country>
<Country>Italy</Country>
<Country>England</Country>
<Country>Sweden</Country>
<Country>Belgium</Country>
<Country>Austria</Country>
<Country>Scotland</Country>
<Country>Russia</Country>
<Country>China</Country>
<Country>Finland</Country>
<Country>Turkey</Country>
<Country>United States</Country>

13 rows selected.
```

SQL	Connection	TimeStamp	Type	Executed	Duration(se...
XQUERYfor $c in distinct-values(ora:view("publisher")//COUNTRY) retur...	LOCALHOST	2024-11-26 ...	SQL	1	0.053

In some rare cases, native XQuery in Oracle will not produce the correct result, even though the exact same XQuery statement will, if executed as part of an SQL statement (with XMLQUERY and Dual).

5 Epilogue

Oracle Database 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. Some of the Oracle specific features described here will probably be replaced in the years to come. The XQuery Update Facility was the latest new feature to be implemented by Oracle. In the examples in the previous chapter, we looked at some of the features that are available in Oracle 23ai. There are many more details. But it has not been the goal of this introduction to cover everything.

Oracle Database has also support for JSON, including features that are part of SQL/JSON. Many of them are similar to the equivalent XML functionality.

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