

Mastering XML Generation in Oracle Database 10g Release 2

An Oracle White Paper
March 2005

Mastering XML Generation in Oracle Database 10g Release 2

Introduction.....	1
Generating XML Using XML SQL Utility (XSU).....	1
Generating XML Using the XSU Java API.....	2
Generating XML Using the XSU PL/SQL API.....	2
Generating XML Using The DBMS_XMLGEN Package	4
Generating XML Using SYS_XMLGEN and SYS_XMLAGG Functions.....	5
SYS_XMLGEN() Function	5
SYS_XMLAGG() Function.....	6
Generating XML Using SQL/XML Functions	6
XMLElement() Function	7
XMLAttributes() Clause.....	7
XMLForest() Function.....	8
XMLConcat() Function	9
XMLAgg() Function.....	10
XMLPI() Function	11
XMLComment() Function	11
XMLRoot() Function.....	11
XMLSerialize() Function.....	12
XMLParse() Function	12
XMLSequence() Function	13
XMLColAttVal() Function.....	15
XMLType Views Based on SQL/XML Generation Functions	15
Conclusion	18

Mastering XML Generation in Oracle Database 10g Release 2

INTRODUCTION

To meet the growing demand for publishing and interchanging relational data in XML, Oracle database has continually evolved over the years to offer increasingly more effective and sophisticated XML generation capabilities based on industry standards. The XML generation approaches listed below reflect their progression through Oracle database releases.

- **XML-SQL Utility:** Introduced in Oracle8i, the Java implementation of XML-SQL Utility, along with its PL/SQL API — the DBMS_XMLQuery package, led the industry in meeting the needs of early XML applications.
- **DBMS_XMLGEN:** As the workload rapidly multiplied for these XML applications, Oracle soon released the high performance, scalable, and database-native DBMS_XMLGEN PL/SQL package in Oracle9i.
- **SYS_XMLGEN and SYS_XMLAGG SQL Functions:** These SQL functions were also introduced in Oracle9i to help generate well-formed XML documents from relational data.
- **SQL/XML:** Recognizing the important nature of XML technologies, Oracle launched the revolutionary XML DB product in Oracle 9i Release 2. As one of the pillars of XML DB, XML generation capabilities were based on the latest SQL 2003 Part 14 — the SQL/XML standard. Oracle continues to lead the industry with the best implementation of newly standardized and emerging SQL/XML capabilities in Oracle Database 10g and the upcoming Oracle Database 10g Release 2.

Oracle continues to lead the industry with the best implementation of newly standardized and emerging SQL/XML capabilities in Oracle Database 10g and the upcoming Oracle Database 10g Release 2.

In the following sections, we will explore each of these capabilities to explain their usage and underlying technology. We will also discuss the advantages and disadvantages of each approach, as well as the recommended usage guidelines.

GENERATING XML USING XML SQL UTILITY (XSU)

XML SQL Utility (XSU) is a collection of Java classes implementing essential services for transforming data retrieved from relational database tables or views into XML, and for inserting, updating, and deleting XML documents in the database. There are also PL/SQL packages mirroring the Java classes for developing PL/SQL programs. Since its introduction in Oracle8i, many XML applications have used JDBC and XSU Java classes in the mid-tier of their J2EE environment to access Oracle databases. PL/SQL programs utilize the Oracle8i database have also relied on the corresponding XSU PL/SQL packages to

process XML data. We will focus on the XML generation aspect of XSU in this section.

XSU is accessible through the following interfaces:

- The `OracleXMLQuery` Java class in the `oracle.xml.sql.query` package to generate XML from relational data.
- The PL/SQL package `DBMS_XMLQuery` mirrors the `OracleXMLQuery` Java class.

Generating XML Using the XSU Java API

The `OracleXMLQuery` class is the main class in XSU for generating XML from relational data. It works in conjunction with JDBC to connect to a database, issue an SQL query, and transform the results into XML data. The code fragment below shows the steps of using the `OracleXMLQuery` class to generate XML data from a specified SQL query.

```
// import the Oracle driver class
import oracle.jdbc.*;

// load the Oracle JDBC driver
DriverManager.registerDriver(new
oracle.jdbc.OracleDriver());

// create the connection
Connection conn = DriverManager.getConnection
("jdbc:oracle:oci:@", "hr", "hr");

// Create an OracleXMLQuery Class instance by passing
// a SQL query to the constructor.
OracleXMLQuery qry = new OracleXMLQuery (conn, "SELECT *
from EMPLOYEES");

// Invoking an OracleXMLQuery method to specify that
// only 20 rows should be included in the result set,
xmlQry.setMaxRows(20);

// Return a DOM object or string by invoking
// OracleXMLQuery methods.
XMLDocument domDoc = (XMLDocument)qry.getXMLDOM();

// Obtain a string object.
String xmlString = qry.getXMLString();

// Perform additional processing on the string
// or DOM as needed...
```

Generating XML Using the XSU PL/SQL API

`DBMS_XMLQuery` is the PL/SQL package that interfaces the underlying `OracleXMLQuery` Java class, which has been loaded into the Oracle database along with other Java packages to run on the integrated OracleJVM. XSU has

also supported the XMLType datatype since Oracle9i. Using XSU with XMLType is convenient if you have XMLType columns in tables.

DBMS_XMLQuery generates a CLOB that contains the XML document with or without a DTD or an XML schema. The following PL/SQL code shows the basic steps of using XMLQuery.

```
-- Declare a variable for the XML query context and a
-- variable for the generated XML.
DECLARE
  v_queryCtx  DBMS_XMLQuery.ctxType;
  v_result    CLOB;
BEGIN
  -- Obtain a context handle with a query.
  v_queryCtx = DBMS_XMLQuery.newContext
    ('SELECT * FROM employees WHERE
     employee_id =: EMPLOYEE_ID AND
     first_name =: FIRST_NAME');

  -- Bind values to the query.
  DBMS_XMLQuery.setBindValue (v_queryCtx,
                              'EMPLOYEE_ID', 20);
  DBMS_XMLQuery.setBindValue (v_queryCtx,
                              'FIRST_NAME', 'John');

  -- Set optional arguments
  DBMS_XMLQuery.setRowSetTag(v_queryCtx, 'EMPSET');

  -- Fetch the XML result as a CLOB using the
  -- getXML function,
  v_result := DBMS_XMLQuery.getXML(v_queryCtx);

  -- Process the results of the XML generation.
  v_xmlstr VARCHAR2(32767);
  v_line   VARCHAR2(2000);

  -- Print the CLOB stored in v_result.
  v_xmlstr := DBMS_LOB.SUBSTR(v_result, 32767);

  LOOP
    EXIT WHEN v_xmlstr IS NULL;
    v_line := substr(v_xmlstr, 1,
                   INSTR(v_xmlstr, CHR(10))-1);
    DBMS_OUTPUT.PUT_LINE('| ' || v_line);
    v_xmlstr := SUBSTR(v_xmlstr,
                      INSTR(v_xmlstr, CHR(10))+1);
  END LOOP;

  -- Close the context.
  DBMS_XMLQuery.closeContext(v_queryCtx);
END;
/
```

GENERATING XML USING THE DBMS_XMLGEN PACKAGE

To dramatically improve the performance and scalability of XML generation from SQL queries, the DBMS_XMLGEN PL/SQL package was introduced in Oracle9i to replace the DBMS_XMLQuery package. This package interfaces to a database-native XML generation functions implemented in the C language.

DBMS_XMLGEN creates XML documents from any SQL query by mapping the database query results into XML. It retrieves the XML document as a CLOB or XMLType. It also provides a fetch interface that allows specification of the maximum rows and rows to skip. This is useful for pagination requirements in Web applications. DBMS_XMLGEN also provides options for changing tag names for ROW, ROWSET, etc.

The result of call the `getXML()` procedure in the DBMS_XMLGen package is a CLOB. The default mapping is as follows:

- Every row of the query result maps to an XML element with the default tag name ROW.
- The entire result is enclosed in a ROWSET element. These names are both configurable, using the `setRowTagName()` and `setRowSetTagName()` procedures in DBMS_XMLGen.
- Each column in the SQL query result is mapped as a sub-element of the ROW element.
- Binary data is transformed to its hexadecimal representation.

The example below creates an XML document by selecting out the employee data from a relational table and putting the resulting CLOB into a table.

```
CREATE TABLE temp_clob_tab(result CLOB);

DECLARE
    qryCtx DBMS_XMLGEN.ctxHandle;
    result CLOB;
BEGIN
    qryCtx := dbms_xmlgen.newContext('SELECT * from
scott.emp');

    -- set the row header to be EMPLOYEE
    DBMS_XMLGEN.setRowTag(qryCtx, 'EMPLOYEE');

    -- now get the result
    result := DBMS_XMLGEN.getXML(qryCtx);

    INSERT INTO temp_clob_tab VALUES(result);

    --close context
    DBMS_XMLGEN.closeContext(qryCtx);
END;
/
```

This query example generates the following XML:

```
SELECT * FROM temp_clob_tab;
```

RESULT

```
-----<br><?xml version="1.0"?><br><ROWSET><br>  <EMPLOYEE><br>    <EMPNO>7369</EMPNO><br>    <ENAME>SMITH</ENAME><br>    <JOB>CLERK</JOB><br>    <MGR>7902</MGR><br>    <HIREDATE>17-DEC-80</HIREDATE><br>    <SAL>800</SAL><br>    <DEPTNO>20</DEPTNO><br>  </EMPLOYEE><br>  <EMPLOYEE><br>    <EMPNO>7499</EMPNO><br>    <ENAME>ALLEN</ENAME><br>    <JOB>SALESMAN</JOB><br>    <MGR>7698</MGR><br>    <HIREDATE>20-FEB-81</HIREDATE><br>    <SAL>1600</SAL><br>    <COMM>300</COMM><br>    <DEPTNO>30</DEPTNO><br>  </EMPLOYEE><br>  ...<br></ROWSET>
```

GENERATING XML USING SYS_XMLGEN AND SYS_XMLAGG FUNCTIONS

Along with DBMS_XMLGEN package, SYS_XMLGEN () and SYS_XMLAGG () functions were also introduced in Oracle9i to help generate well-formed XML documents from relational data.

SYS_XMLGEN() Function

This Oracle Database-specific SQL function is similar to the XMLElement () except that it takes a single argument and converts the result to XML. Unlike the other XML generation functions, SYS_XMLGEN () always returns a well-formed XML document. Unlike DBMS_XMLGEN which operates at a query level, SYS_XMLGEN () operates at the row level returning a XML document for each row.

SYS_XMLGEN () is powerful for the following reasons:

- You can create and query XML instances *within* SQL queries.
- Using the relational database infrastructure, you can create complex and nested XML instances from simple relational tables. For example, when you use an XMLType view that is a SYS_XMLGEN () on top of an object type, Oracle XML DB rewrites these queries when possible.

SYS_XMLGEN () creates an XML document from either of the following:

- A user-defined type (UDT) instance
- A scalar value passed
- XML

and returns an XMLType instance contained in the document.

`SYS_XMLGEN()` also optionally inputs a `XMLFormat` object type through which you can customize the SQL results. A `NULL` format object implies that the default mapping action is to be used.

`SYS_XMLGEN()` creates and queries XML instances in SQL queries, as follows:

```
SELECT SYS_XMLGEN(employee_id) AS "result"  
       FROM employees WHERE fname LIKE 'John%';
```

The resulting XML document is:

```
result  
-----  
<?xml version="1.0"?>  
<EMPLOYEE_ID>1001</EMPLOYEE_ID>
```

1 row selected.

SYS_XMLAGG() Function

`SYS_XMLAGG()` function aggregates all XML documents or fragments represented by `expr` and produces a single XML document. It adds a new enclosing element with a default name, `ROWSET`. To format the XML document differently, use the `'fmt'` parameter.

The following example uses the `SYS_XMLGen` function to generate an XML document for each row of the sample table `employees` where the employee's last name begins with the letter R, and then aggregates all of the rows into a single XML document in the default enclosing element `ROWSET`:

```
SELECT SYS_XMLAGG(SYS_XMLGEN(last_name))  
       FROM employees  
       WHERE last_name LIKE 'R%';
```

```
SYS_XMLAGG(SYS_XMLGEN(LAST_NAME))
```

```
-----  
<ROWSET>  
  <LAST_NAME>Raphaely</LAST_NAME>  
  <LAST_NAME>Rogers</LAST_NAME>  
  <LAST_NAME>Rajs</LAST_NAME>  
  <LAST_NAME>Russell</LAST_NAME>  
</ROWSET>
```

GENERATING XML USING SQL/XML FUNCTIONS

As industry-wide XML adoption took off, Oracle spearheaded the effort with other database vendors in the SQL standard committee to create a new SQL/XML standard, which is now included in the SQL 2003 and SQL 2005 standards as Part 14. The SQL/XML standard defines how SQL can be used in conjunction with XML in a database, including detailed definition of a new XML type, the values of an XML type, mappings between SQL constructs and XML constructs, and functions for generating XML from SQL data. Since Oracle Database 9i Release 2, SQL/XML features have been supported as an integral part of the Oracle SQL engine, which also provides additional SQL extensions to support querying, updating, and transformation of XML data. As a

founding member, Oracle continues to drive the SQL/XML committee efforts to define essential new features for the upcoming SQL 2005 standard.

With the addition of the SQL/XML functions to Oracle's exceptional SQL engine, Oracle continues to lead the industry with the best implementation of newly standardized and emerging SQL/XML capabilities in Oracle 9i Release 2, Oracle Database 10g, and the new Oracle Database 10g Release 2.

Using SQL/XML function is the most efficient and flexible method of generating XML from relational data. A number of SQL/XML functions for XML generation are supported in Oracle XML DB: `XMLRoot()`, `XMLPI()`, `XMLComment()`, `XMLElement()`, `XMLForest()`, `XMLConcat()`, `XMLAgg()`, and `XMLSerialize()`. There is also an Oracle Database extension function for XML generation: `XMLColAttVal()`.

XMLElement() Function

`XMLElement()` function takes an element name, an optional collection of attributes for the element, and zero or more arguments that make up the element content and returns an instance of `XMLType`.

`XMLElement()` is primarily used to construct XML instances from relational data. Since not every SQL identifier is an acceptable XML Name, it is necessary to define a mapping of SQL identifiers to XML Names.

As part of generating a valid XML element name from a SQL identifier, characters that are disallowed in an XML element name are escaped. With *partial escaping*, the SQL identifiers other than the ":" sign that are not representable in XML are preceded by an escape character using the # sign followed by the unicode representation of that character in hexadecimal format. This can be used to specify namespace prefixes for the elements being generated. The *fully escaped* mapping escapes all non-XML characters in the SQL identifier name, including the ":" character.

XMLAttributes() Clause

`XMLElement()` also takes an optional `XMLAttributes()` clause, which specifies the attributes of that element. This can be followed by a list of values that make up the children of the newly created element. In the `XMLAttributes()` clause, the value expressions are evaluated to get the values for the attributes.

The list of values that follow the `XMLAttributes()` clause are converted to XML format, and are made as children of the top-level element. If the expression evaluates to NULL, then no element is created for that expression.

The example below produces an `Emp` element for each employee, with an `id` and `name` attribute:

```
SELECT XMLELEMENT ("Emp", XMLATTRIBUTES (
                                e.employee_id as "ID",
```

```

        e.fname || ' ' || e.lname AS "name"))
AS "result"
FROM employees e
WHERE employee_id > 200;

```

This query produces the following typical XML result fragment:

```

result
-----
<Emp ID="1001" name="John Smith"/>
<Emp ID="1206" name="Mary Martin"/>

```

The next example illustrates the use of namespaces to create an XML schema-based document. Assuming that an XML schema "http://www.oracle.com/Employee.xsd" exists and has no target namespace, then the following query creates an XMLType instance conforming to that schema:

```

SELECT XMLElement("Employee", XMLAttributes
  ('http://www.w3.org/2001/XMLSchema' AS "xmlns:xsi",
   'http://www.oracle.com/Employee.xsd' AS
    "xsi:nonamespaceSchemaLocation"),
  XMLForest(employee_id, last_name, salary)) AS "RESULT"
FROM hr.employees
WHERE department_id = 10;

```

This creates the following XML document that conforms to XML schema Employee.xsd.

```

RESULT
-----
<Employee xmlns:xsi=http://www.w3.org/2001/XMLSchema
xsi:nonamespaceSchemaLocation="http://www.oracle.com/Emp
loyee.xsd">
  <EMPLOYEE_ID>200</EMPLOYEE_ID>
  <LAST_NAME>Whalen</LAST_NAME>
  <SALARY>4400</SALARY>
</Employee>

```

1 row selected.

XMLForest() Function

XMLForest() function produces a forest of XML elements from the given list of arguments. The arguments may be value expressions with optional aliases.

The list of value expressions are converted to XML format. For a given expression, if the AS clause is omitted, then the fully escaped form of the column name is used as the name of the enclosing tag of the element.

For an object type or collection, the AS clause is mandatory. For other types, the AS clause can be optionally specified. If the AS clause is specified, then the partially escaped form of the alias is used as the name of the enclosing tag. If the expression evaluates to NULL, then no element is created for that expression.

The example below generates an `Emp` element for each employee, with a `name` attribute and elements with the employee's start date and department as the content.

```
SELECT XMLELEMENT("Emp", XMLATTRIBUTES (e.fname || ' ' ||
e.lname AS "name"),
      XMLForest (e.hire, e.department AS "department"))
AS "result"
FROM employees e;
```

This query might produce the following XML result:

```
result
-----
<Emp name="John Smith">
  <HIRE>24-MAY-00</HIRE>
  <department>Accounting</department>
</Emp>
<Emp name="Mary Martin">
  <HIRE>FEB-01-96</HIRE>
  <department>Shipping</department>
</Emp>
```

2 rows selected.

XMLConcat() Function

`XMLConcat()` function concatenates all the arguments passed in to create a XML fragment. The `XMLConcat()` syntax. `XMLConcat()` has two forms:

- The first form generates an `XMLType` containing a document fragment by taking an `XMLSequenceType`, which is a `VARRAY` of `XMLType`, and returns a single `XMLType` instance that is the concatenation of all of the elements of the varray. This form is useful to collapse lists of `XMLTypes` into a single instance.
- The second form takes an arbitrary number of `XMLType` values and concatenates them together. If one of the value is null, then it is ignored in the result. If all the values are NULL, then the result is NULL. This form is used to concatenate arbitrary number of `XMLType` instances in the same row. `XMLAgg()` can be used to concatenate `XMLType` instances across rows.

The following example shows `XMLConcat()` returning the concatenation of `XMLTypes` from the `XMLSequenceType`:

```
SELECT XMLConcat(XMLSequenceType (
xmltype ('<PartNo>1236</PartNo>'),
xmltype ('<PartName>Widget</PartName>'),
xmltype ('<PartPrice>29.99</PartPrice>'))).getClobVal ()
AS "result"
FROM dual;
```

returns a single fragment of the form:

```
result
-----
```

```
<PartNo>1236</PartNo>
<PartName>Widget</PartName>
<PartPrice>29.99</PartPrice>
```

1 row selected.

XMLAgg() Function

XMLAgg () is an aggregate function that produces a forest of XML elements from a collection of XML elements. As with XMLConcat (), any arguments that are null are dropped from the result. This function can be used to concatenate XMLType instances across multiple rows. It also allows an optional ORDER BY clause to order the XML values being aggregated.

XMLAgg () is an aggregation function and hence produces one aggregated XML result for each group. If there is no group by specified in the query, then it returns a single aggregated XML result for all the rows of the query.

The following example produces a Department element containing Employee elements with employee job ID and last name as the contents of the elements. It also orders the employee XML elements in the department by their last name.

```
SELECT
XMLELEMENT ("Department", XMLAGG (XMLELEMENT ("Employee",
      e.job||' '||e.ename) ORDER BY e.ename))
  AS "Dept_list"
FROM scott.emp e
WHERE e.deptno = 10;
```

```
Dept_list
-----
<Department>
  <Employee>MANAGER CLARK</Employee>
  <Employee>PRESIDENT KING</Employee>
  <Employee>CLERK MILLER</Employee>
</Department>
```

1 row selected.

The result is a single row, because XMLAgg () aggregates the rows. You can use the GROUP BY clause to group the returned set of rows into multiple groups:

```
SELECT XMLELEMENT ("Department",
      XMLAttributes (deptno AS "deptno"), XMLAgg (
      XMLElement ("Employee", e.job||' '||e.ename)))
  AS "Dept_list"
FROM scott.emp e
GROUP BY e.deptno;
```

```
Dept_list
-----
<Department deptno="10">
  <Employee>MANAGER CLARK</Employee>
  <Employee>PRESIDENT KING</Employee>
  <Employee>CLERK MILLER</Employee>
```

```

</Department>

<Department deptno="20">
  <Employee>CLERK SMITH</Employee>
  <Employee>ANALYST FORD</Employee>
  <Employee>CLERK ADAMS</Employee>
  <Employee>ANALYST SCOTT</Employee>
  <Employee>MANAGER JONES</Employee>
</Department>

<Department deptno="30">
  <Employee>SALESMAN ALLEN</Employee>
  <Employee>MANAGER BLAKE</Employee>
  <Employee>SALESMAN MARTIN</Employee>
  <Employee>SALESMAN TURNER</Employee>
  <Employee>CLERK JAMES</Employee>
  <Employee>SALESMAN WARD</Employee>
</Department>

```

3 rows selected.

XMLPI() Function

SQL/XML Function XMLPI () returns an instance of XMLType containing XML processing instructions. In the following example,

```

SELECT XMLPI (NAME "OrderAnalysisComp", 'imported,
reconfigured, disassembled')
  AS pi FROM DUAL;

```

This results in the following output :

```

PI
-----
<?OrderAnalysisComp imported, reconfigured,
disassembled?>

```

1 row selected.

XMLComment() Function

You use SQL function XMLComment to generate XML comments. Function XMLComment returns an instance of XMLType. If string-result is NULL, then the function returns NULL.

```

SELECT XMLComment('This is a comment') AS cmnt FROM
DUAL;

```

This query results in the following output:

```

CMNT
-----
<!--This is a comment-->

```

XMLRoot() Function

The SQL/XML function XMLRoot () can be used to add a VERSION property, and optionally a STANDALONE property, to the root information item of an

XML document. Typically, this is done to ensure data-model compliance. This function returns an instance of XMLType.

The example below generates an XML document with a version property and a STANDALONE property.

```
SELECT XMLRoot(XMLType('<poid>143598</poid>'), VERSION
'1.0', STANDALONE YES)
AS xmlroot FROM DUAL;
```

This results in the following output :

```
XMLROOT
-----
<?xml version="1.0" standalone="yes"?>
<poid>143598</poid>
```

1 row selected.

XMLSerialize() Function

The SQL/XML function XMLSerialize() can be used to obtain a string or a LOB representation of XML data. The following example shows how this function generates a string containing the contents of an XML value.

```
SELECT XMLSerialize(DOCUMENT
XMLType('<poid>143598</poid>') AS CLOB)
AS xmlserialize_doc FROM DUAL;
```

This results in the following output:

```
XMLSERIALIZE_DOC
-----
<poid>143598</poid>
```

XMLParse() Function

The SQL/XML function XMLParse() can be used to parse a string containing XML data and generate a corresponding value of XMLType.

The WELLFORMED keyword is an Oracle XML DB extension to the SQL/XML standard. When you specify WELLFORMED, you are informing the parser that the XML value expression argument is well-formed, so that Oracle XML DB does not check again whether the the XML value is well-formed.

Here is an example of using the XMLParse() function.

```
SELECT XMLParse(CONTENT
'124 <purchaseOrder poNo="12435">
      <customerName> Acme Enterprises</customerName>
      <itemNo>32987457</itemNo>
    </purchaseOrder>'
WELLFORMED)
AS po FROM DUAL d;
```

This results in the following output :

```
PO
```

```
-----  
124 <purchaseOrder poNo="12435">  
<customerName>Acme Enterprises</customerName>  
<itemNo>32987457</itemNo>  
</purchaseOrder>
```

XMLSequence() Function

XMLSequence () function returns a sequence of XMLType. The function returns an XMLSequenceType which is a VARRAY of XMLType instances. Because this function returns a collection, it can be used in the FROM clause of SQL queries. XMLSequence () is essential for effective SQL queries involving XMLTypes. This function is also an Oracle Database extension to the SQL/XML standard functions.

The XMLSequence () function has two forms:

- The first form inputs an XMLType instance and returns a VARRAY of top-level nodes. This form can be used to shred XML fragments into multiple rows.
- The second form takes as input a REFCURSOR argument, with an optional instance of the XMLFormat object and returns the VARRAY of XMLTypes corresponding to each row of the cursor. This form can be used to construct XMLType instances from arbitrary SQL queries.

Here is an example of the first form. Suppose you had the following XML document containing employee information:

```
<EMPLOYEES>  
<EMP>  
  <EMPNO>112</EMPNO>  
  <EMPNAME>Joe</EMPNAME>  
  <SALARY>50000</SALARY>  
</EMP>  
<EMP>  
  <EMPNO>217</EMPNO>  
  <EMPNAME>Jane</EMPNAME>  
  <SALARY>60000</SALARY>  
</EMP>  
<EMP>  
  <EMPNO>412</EMPNO>  
  <EMPNAME>Jack</EMPNAME>  
  <SALARY>40000</SALARY>  
</EMP>  
</EMPLOYEES>
```

To create a new XML document containing only employees who make \$50,000 or more a year, you can use the following syntax:

```
SELECT SYS_XMLAGG(value(e), xmlformat('EMPLOYEES'))  
  FROM TABLE(XMLSequence(Extract(doc, '/EMPLOYEES/EMP'))) e  
 WHERE EXTRACTVALUE(value(e), '/EMP/SALARY') >= 50000;
```

This returns the following XML document:

```
<EMPLOYEES>  
<EMP>  
  <EMPNO>112</EMPNO>  
  <EMPNAME>Joe</EMPNAME>
```

```

        <SALARY>50000</SALARY>
    </EMP>
    <EMP>
        <EMPNO>217</EMPNO>
        <EMPNAME>Jane</EMPNAME>
        <SALARY>60000</SALARY>
    </EMP>
</EMPLOYEES>

```

2 rows selected.

Notice how `extract()` was used to extract out all the employees:

- `Extract()` returns a fragment of EMP elements.
- `XMLSequence()` creates a collection of these top level elements into `XMLType` instances and returns that.
- The `TABLE` function was then used to makes the collection into a table value which can be used in the `FROM` clause of queries.

An example of the second form of `XMLSequence()` is shown below that creates an XML document for each row of the cursor expression and returns the value as an `XMLSequenceType`. The `XMLFormat` object can be used to influence the structure of the resulting XML documents. For example, a call such as:

```

SELECT value(e).getClobVal() AS "xmltype"
FROM TABLE(XMLSequence(Cursor(SELECT * FROM scott.emp))) e;

```

might return the following XML:

```

xmltype
-----
<ROW>
  <EMPNO>7369</EMPNO>
  <ENAME>SMITH</ENAME>
  <JOB>CLERK</JOB>
  <MGR>7902</MGR>
  <HIREDATE>17-DEC-80</HIREDATE>
  <SAL>800</SAL>
  <DEPTNO>20</DEPTNO>
</ROW>

<ROW>
  <EMPNO>7499</EMPNO>
  <ENAME>ALLEN</ENAME>
  <JOB>SALESMAN</JOB>
  <MGR>7698</MGR>
  <HIREDATE>20-FEB-81</HIREDATE>
  <SAL>1600</SAL>
  <DEPTNO>30</DEPTNO>
</ROW>
...

```

14 rows selected.

The `ROW` tag used for each row can be changed using the `XMLFormat` object.

XMLColAttVal() Function

XMLColAttVal() function generates a forest of XML column elements containing the value of the arguments passed in. This function is also an Oracle Database extension to the SQL/XML standard functions.

The name of the arguments are put in the name attribute of the column element. Unlike the XMLForest() function, the name of the element is not escaped in any way and hence this function can be used to transport SQL columns and values without escaped names.

The following example generates an Emp element for each employee, with a name attribute and elements with the employee's start date and department as the content.

```
SELECT XMLELEMENT("Emp",
  XMLATTRIBUTES(e.fname || ' ' || e.lname AS "name" ),
  XMLCOLATTVAL(e.hire, e.department AS department"))
AS "result"
FROM employees e;
```

This query might produce the following XML result:

```
result
-----
<Emp name="John Smith">
  <column name="HIRE">24-MAY-00</column>
  <column name="department">Accounting</column>
</Emp>
<Emp name="Mary Martin">
  <column name="HIRE">01-FEB-96</column>
  <column name="department">Shipping</column>
</Emp>
```

2 rows selected.

Because the name associated with each XMLColAttVal() argument is used to populate an attribute value, neither the fully escaped mapping nor the partially escaped mapping is used.

XMLTYPE VIEWS BASED ON SQL/XML GENERATION FUNCTIONS

Another powerful mechanism to generate XML from relational data is to use XMLType views. XMLType views can be created by taking advantage the rich set of SQL/XML generation functions provided by Oracle XML DB. This enables construction of XMLType view from the underlying relational tables directly without physically migrating those relational legacy data into XML.

Each row of an XMLType view corresponds to an XMLType instance. The object identifier for uniquely identifying each row in the view can be created using a function such as extract with getNumberVal() applied to the XMLType result. Once created, an XMLType view can be reused in queries to return XML data for specific needs of an application.

The example below creates an XMLType view from a complex query with an inner select by using a number of SQL/XML generation functions, namely, XMLElement(), XMLAttributes(), XMLConcat(), XMLRoot(), XMLPI() and XMLComment() functions.

```

create or replace view DEPARTMENT_XML of xmltype
with object id
(
  substr(extractValue(sys_nc_rowinfo$, '/Department/Name'),1,30)
)
as
select xmlroot(xmlConcat(xmlPI("pi", 'valid, well-formed'),
  xmlElement
  (
    "Department",
    xmlAttributes( d.DEPARTMENT_ID as "DepartmentId"),
    xmlElement("Name", d.DEPARTMENT_NAME),
    xmlElement
    (
      "Location",
      xmlForest
      (
        STREET_ADDRESS as "Address",
        CITY as "City",
        STATE_PROVINCE as "State",
        POSTAL_CODE as "Zip",
        COUNTRY_NAME as "Country"
      )
    ),
  ),
  xmlElement
  (
    "EmployeeList",
    xmlcomment('Generated from an inner select call'),
    (
      select xmlAgg
      (
        xmlElement
        (
          "Employee",
          xmlAttributes (
            e.EMPLOYEE_ID as "employeeNumber" ),
          xmlForest
          (
            e.FIRST_NAME as "FirstName",
            e.LAST_NAME as "LastName",
            e.EMAIL as "EmailAddress",
            e.PHONE_NUMBER as "Telephone",
            e.HIRE_DATE as "StartDate",
            j.JOB_TITLE as "JobTitle",
            e.SALARY as "Salary",
            m.FIRST_NAME || ' ' || m.LAST_NAME
          as "Manager"
          ),
          xmlElement (
            "Commission", e.COMMISSION_PCT )
          )
        )
      )
    from HR.EMPLOYEES e, HR.EMPLOYEES m, HR.JOBS j
    where e.DEPARTMENT_ID = d.DEPARTMENT_ID
    and j.JOB_ID = e.JOB_ID
    and m.EMPLOYEE_ID = e.MANAGER_ID
  )
)
)
)

```

```

), VERSION '1.0') as XML
from HR.DEPARTMENTS d, HR.COUNTRIES c, HR.LOCATIONS l
where d.LOCATION_ID = l.LOCATION_ID
and l.COUNTRY_ID = c.COUNTRY_ID

```

This XMLType view will generate rows of XMLType instances as the following:

```
SQL> select object_value from department_xml;
```

```
OBJECT_VALUE
```

```

-----
<?xml version="1.0"?>
<?pi valid, well-formed?>
<Department DepartmentId="10">
  <Name>Administration</Name>
  <Location>
    <Address>2004 Charade Rd</Address>
    <City>Seattle</City>
    <State>Washington</State>
    <Zip>98199</Zip>
    <Country>United States of America</Country>
  <EmployeeList>
    <!--Generated from an inner select call-->
    <Employee employeeNumber="200">
      <FirstName>Jennifer</FirstName>
      <LastName>Whalen</LastName>
      <EmailAddress>JWHALEN</EmailAddress>
      <Telephone>515.123.4444</Telephone>
      <StartDate>17-SEP-87</StartDate>
      <JobTitle>Administration Assistant</JobTitle>
      <Salary>4400</Salary>
      <Manager>Neena Kochhar</Manager>
      <Commission/>
    </Employee>
  </EmployeeList>
</Department>

<?xml version="1.0"?>
<?pi valid, well-formed?>
<Department DepartmentId="20">
  <Name>Marketing</Name>
  <Location>
    <Address>147 Spadina Ave</Address>
    <City>Toronto</City>
    <State>Ontario</State>
    <Zip>M5V 2L7</Zip>
    <Country>Canada</Country>
  </Location>
  <EmployeeList>
    <!--Generated from an inner select call-->
    <Employee employeeNumber="201">
      <FirstName>Michael</FirstName>
      <LastName>Hartstein</LastName>
      <EmailAddress>MHARTSTE</EmailAddress>
      <Telephone>515.123.5555</Telephone>
      <StartDate>17-FEB-96</StartDate>
      <JobTitle>Marketing Manager</JobTitle>
      <Salary>13000</Salary>
      <Manager>Steven King</Manager>
      <Commission/>
    </Employee>
    <Employee employeeNumber="202">
      <FirstName>Pat</FirstName>
      <LastName>Fay</LastName>
      <EmailAddress>PFAY</EmailAddress>

```

```

        <Telephone>603.123.6666</Telephone>
        <StartDate>17-AUG-97</StartDate>
        <JobTitle>Marketing Representative</JobTitle>
        <Salary>6000</Salary>
        <Manager>Michael Hartstein</Manager>
        <Commission/>
    </Employee>
</EmployeeList>
</Department>
...
27 rows selected.

```

We can query this XMLType view with other query criteria to further tailor XML generation. Continue with the same example, the query below will only retrieve information about the 'Human Resources' department.

```

SQL> select value(d).extract('/') from DEPARTMENT_XML d
      where existsNode(object_value,
        '/Department[Name=" Human Resources "]') = 1
/

```

```

VALUE(D).EXTRACT('/')
-----
<Department DepartmentId="40">
  <Name>Human Resources</Name>
  <Location>
    <Address>8204 Arthur St</Address>
    <City>London</City>
    <Country>United Kingdom</Country>
  </Location>
  <EmployeeList>
    <!--Generated from an inner select call-->
    <Employee employeeNumber="203">
      <FirstName>Susan</FirstName>
      <LastName>Mavris</LastName>
      <EmailAddress>SMAVRIS</EmailAddress>
      <Telephone>515.123.7777</Telephone>
      <StartDate> 1994-06-07</StartDate>
      <JobTitle>Human Resources Representative</JobTitle>
      <Salary>6500</Salary>
      <Manager>Neena Kochhar</Manager>
      <Commission/>
    </Employee>
  </EmployeeList>
</Department>

```

CONCLUSION

XML generation from relational data has a vital role in XML applications for publishing and interchanging XML data. From XSU in the early days, to the subsequent DBMS_XMLGEN package, and finally the SQL/XML functions, we have shown how Oracle database has continually evolved over the years to offer increasingly more effective, sophisticated, and standard-based XML generation capabilities. For new XML applications, the best choice for XML generation is to use the high performance, scalable, and standards-based SQL/XML functions. As XML adoption accelerates to become mainstream, Oracle plans to introduce diverse, versatile, and essential new capabilities for XML generation based on SQL/XML standard in future releases.

For new XML applications, the best choice for XML generation is to use the high performance, scalable, and standards-based SQL/XML functions.



Mastering XML Generation in Oracle Database 10g Release 2
March 2005
Author: Geoff Lee

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200
www.oracle.com

Oracle Corporation provides the software
that powers the internet.

Oracle is a registered trademark of Oracle Corporation. Various
product and service names referenced herein may be trademarks
of Oracle Corporation. All other product and service names
mentioned may be trademarks of their respective owners.

Copyright © 2005 Oracle Corporation
All rights reserved.