This chapter contains the following sections:

- What are the Oracle9i Native XML Database Features?
- XMLType Datatype
  - When to use XMLType
  - XMLType Storage in the Database
  - XMLType Functions
  - Manipulating XML Data in XMLType Columns
  - Selecting and Querying XML Data
- Indexing XMLType columns
- Java Access to XMLType (oracle.xdb.XMLType)
- DBMS_XMLGEN
- SYS_XMLGEN
- SYS_XMLAGG
- TABLE Functions
- Frequently Asked Questions (FAQs): XMLType
What are the Oracle9i Native XML Database Features?

Oracle9i supports XMLType, a new system defined object type. XMLType has built-in member functions that offer a powerful mechanism to create, extract and index XML data. Users can also generate XML documents as XMLType instances dynamically using the SQL functions, SYS_XMLGEN and SYS_XMLAGG, and the PL/SQL package DBMS_XMLGEN.

Table 5–1 summarizes the new XML features natively supported in Oracle9i.

Table 5–1 Oracle9i Native XML Support Feature Summary

<table>
<thead>
<tr>
<th>XML Feature</th>
<th>Description</th>
</tr>
</thead>
</table>
| XMLType         | (new) XMLType is a system defined datatype with predefined member functions to access XML data. You can perform the following tasks with XMLType:  
■ Create columns of XMLType and use XMLType member functions on instances of the type. See “XMLType Datatype” on page 5-3.  
■ Create PL/SQL functions and procedures, with XMLType as argument and return parameters. See, “When to use XMLType” on page 5-8.  
■ Store, index, and manipulate XML data in XMLType columns. Refer to “XMLType Datatype” on page 5-3. |
| DBMS_XMLGEN     | (new) DBMS_XMLGEN is a PL/SQL package that converts the results of SQL queries to canonical XML format, returning it as XMLType or CLOB. DBMS_XMLGEN is implemented in C, and compiled in the database kernel. DBMS_XMLGEN is similar in functionality to DBMS_XMLQUERY package. Refer to “DBMS_XMLGEN” on page 5-31. |
XMLType Datatype

XMLType is a new server datatype that can be used as columns in tables, and views. Variables of XMLType can be used in PL/SQL stored procedures as parameters, return values, and so on. You can use XMLType inside the server, in PL/SQL, SQL and Java. It is not currently supported through OCI.

XMLType includes useful built-in member functions that operate on XML content. For example, function Extract extracts a specific node(s) from an XMLType instance.

New SQL functions such as SYS_XMLGEN that return XML documents return them as XMLType. This provides strong typing in SQL statements. You can use

### Table 5–1 Oracle9i Native XML Support Feature Summary (Cont.)

<table>
<thead>
<tr>
<th>XML Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYS_XMLGEN</td>
<td>(new) SYS_XMLGEN is a SQL function, which generates XML within SQL queries. DBMS_XMLGEN and other packages operate at a query level, giving aggregated results for the entire query. SYS_XMLGEN operates on a single argument inside a SQL query and converts the result to XML. SYS_XMLGEN takes in a scalar value, object type, or a XMLType instance to be converted to an XML document. It also takes an optional XMLGenFormatType object to specify formatting options for the result. SYS_XMLGEN returns a XMLType. Refer to &quot;SYS_XMLGEN&quot; on page 5-63.</td>
</tr>
<tr>
<td>SYS_XMLAGG</td>
<td>(new) SYS_XMLAGG is an aggregate function, which aggregates over a set of XMLType's. SYS_XMLAGG aggregates all the input XML documents/fragments and produces a single XML document by concatenating XML fragments, and adding a top-level tag. Refer to &quot;SYS_XMLAGG&quot; on page 5-72.</td>
</tr>
<tr>
<td>UriTypes</td>
<td>(new) The UriType family of types can store and query Uri-refs in the database. SYS.UriType is an abstract object type which provides functions to access the data pointed to by the URL. SYS.HttpUriType and SYS.DBUriType are subtypes of UriType. The HttpUriType can store HTTP URLs and the DBUriType can store intra-database references. You can also define your own subtypes of SYS.UriType to handle different URL protocols.</td>
</tr>
</tbody>
</table>
XMLType in SQL queries in the same way as any other user-defined datatypes in the system.

**Note:** In this release, XMLType is only supported in the server in SQL, PL/SQL, and Java. To use XMLType on the client side, use functions such as getClobVal() or other functions that retrieve the complete XML document.

### How to use XMLType

XMLType can be used to create table columns. The createXML() static function in the XMLType can be used to create XMLType instances for insertion. By storing your XML documents as XMLType, XML content can be readily searched using standard SQL queries.

We will show some simple examples on how to create an XMLType column and use it in a SQL statement.

#### Example of Creating XMLType columns

The XMLType column can be created like any other user-defined type column,

```sql
CREATE TABLE warehouses(
    warehouse_id NUMBER(3),
    warehouse_spec SYS.XMLTYPE,
    warehouse_name VARCHAR2(35),
    location_id NUMBER(4));
```

#### Example of Inserting values into an XMLType column

To insert values into the XMLType column, you would need to bind an XMLType instance. An XMLType instance can be easily created from a VARCHAR or a CLOB by using the XMLType’s createXML() static function.

```sql
INSERT into warehouses (warehouse_id, warehouse_spec)
VALUES (1001, sys.XMLType.createXML(
    '<Warehouse whNo="100">
    <Building>Owned</Building>
    </Warehouse>'));
```

This example created an XMLType instance from a string literal. The input to createXML could be any expression that returns a VARCHAR2 or CLOB.
The createXML function also checks that the input XML is well-formed. It does not check for validity of the XML.

#### Example of Using XMLType in a SQL Statement

The following simple SELECT statement shows how you can use XMLType in an SQL statement:

```sql
SELECT
    w.warehouse_spec.extract('/Warehouse/Building/text()').getStringVal()
    "Building"
FROM warehouses w

where warehouse_spec is an XMLType column operated on by member function Extract(). The result of this simple query is a string (varchar2):

```
Building
```

See Also:  "How to use XMLType" on page 5-4.

#### Example of Updating an XMLType column

In this release, an XML document in an XMLType is stored packed in a CLOB. Consequently updates, have to replace the document in place. We do not support piece-wise update of the XML for this release.

To update an XML document, you would fire a standard SQL update statement, except that you would bind an XMLType instance.

```sql
UPDATE warehouses SET warehouse_spec =
    sys.XMLType.createXML(
        '<Warehouse whono="200">
            <Building>Leased</Building>
        </Warehouse>'));
```

In this example, we are creating an XMLType instance from a string literal and updating the warehouse_spec column with the new value. Note that any triggers would get fired on the update statement and you can see and modify the XML value inside the triggers.

#### Example of Deleting a row containing an XMLType column

Deleting a row containing an XMLType column is no different from any other datatype.
You can use the Extract and ExistsNode functions to identify rows to delete as well. For example to delete all warehouse rows for which the warehouse building is Leased, we can write a statement such as,

```
DELETE FROM warehouses e
WHERE e.warehouse_spec.extract('//Building/text()').getStringVal()
    = 'Leased';
```

## Guidelines for Using XMLType Columns

The following are guidelines for storing XML data in XMLType columns:

- **Define column XMLType.** First, define a column of XMLType. You can include optional storage characteristics with the column definition.

- **Create an XMLType instance.** Use the XMLType constructor to create the XMLType instance before inserting into the column. You can also use the SYS_XMLGEN and SYS_XMLAGG functions to directly create instances of XMLType. See "SYS_XMLGEN Example 3: Converting XMLType Instance" on page 5-68 and "SYS_XMLAGG Example 2: Aggregating XMLType Instances Stored in Tables" on page 5-74.

- **Select or extract a particular XMLType instance.** You can select out the XMLType instance from the column. XMLType offers a choice of member functions, such as, `extract()` and `existsNode()`, to extract a particular node or check to see if a node exists, respectively. See Table 5-3, "XMLType Member and Static Functions".

  **See Also:**

  - "XMLType Query Example 6 — Extract fragments from XMLType" on page 28
  - "XMLType Query Example 1 — Retrieve an XML Document as a CLOB" on page 5-20

- **You can define an Oracle Text index.** You can define an Oracle Text (interMedia Text) index on XMLType columns. This enables you to use CONTAINS, HASPATH, INPATH, and other text operators on the column. All the text operators and index functions that operate on LOB columns, also work on XMLType columns.
Benefits of XMLType

Using XMLType has the following advantages:

- It brings the worlds of XML and SQL together as it enables:
  - SQL operations on XML content
  - XML operations on SQL content
- It is convenient to implement as it includes built-in functions, indexing support, navigation, and so on.

XMLType Interacts with Other SQL Constructs

You can use XMLType in SQL statements combined with other columns and datatypes. For example, you can query XMLType columns, join the result of the extraction with a relational column, and then Oracle can determine an optimal way to execute these queries.

You Can Select a Tree or Serialized Format for Your Resulting XML

XMLType is optimized to not materialize the XML data into a tree structure unless needed. Hence when SQL selects XMLType instances inside queries, only a serialized form is exchanged across function boundaries. These are exploded into tree format only when operations such as extract() and existsNode() are performed. The internal structure of XMLType is also an optimized DOM-like tree structure.

You Can Create functional indexes and Text indexes on XMLType

Oracle9i text index has been enhanced to support XMLType columns as well. You can create functional indexes on ExistsNode and Extract functions as well to speed up query evaluation.

See Also: Chapter 8, "Searching XML Data with Oracle Text"

See Also:

- "Indexing XMLType columns" on page 31
- Chapter 8, "Searching XML Data with Oracle Text"
- Oracle9i Application Developer's Guide - Large Objects (LOBs)
When to use XMLType

Use XMLType in the following cases:

- You need to store XML as a whole in the database and retrieve it.
- You need SQL queriability on some or the whole document. The functions ExistsNode and Extract provide the necessary SQL queriability over XML documents.
- You need strong typing inside SQL statements and PL/SQL functions. Strong typing implies that you ensure that the values passed in are XML values and not any arbitrary text string.
- You need the XPath functionality provided by Extract and ExistsNode functions to work on your XML document. Note that the XMLType uses the built-in C XML parser and processor and hence would provide better performance and scalability when used inside the server.
- You need indexing on XPath searches on documents. XMLType provides member functions that can be used to create functional indexes to optimize searches.
- You do not need piecewise updates of the document.
- Shield applications from storage models - In future releases, the XMLType would support different storage alternatives. Using XMLType instead of CLOBs or relational storage, allows applications to gracefully move to various storage alternatives later, without affecting any of the query or DML statements in the application.
- Preparing for future optimizations - All new functionality related to XML will only support the XMLType. Since the server is natively aware that the XMLType can only store XML data, better optimizations and indexing techniques can be done in the future. By writing applications to use XMLType, these optimizations and enhancements can be easily achieved in the future without rewriting the application.

XMLType Storage in the Database

In this release, XMLType offers a single CLOB storage option. In future releases, Oracle may provide other storage options, such as BLOBs, NCLOBs, and so on.

When you create an XMLType column, a hidden CLOB column is automatically created to store the XML data. The XMLType column itself becomes a virtual column over this hidden CLOB column. It is not possible to directly access the
XMLType Storage in the Database

CLOB column, however, you can set the storage characteristics for the column using the XMLType storage clause.

You cannot create VARRAYs of XMLType and store it in the database since VARRAYs do not support CLOBs when stored in tables.

---

**Note:** You cannot create columns of VARRAY types which contain XMLType. This is because Oracle does not support LOB locators inside VARRAYs, and XMLType (currently) always stores the XML data in a CLOB.

---

**XMLType Creation Example 1 — Creating XMLType Columns**

As explained earlier, you can create XMLType columns by simply using the XMLType as the datatype.

The following statement creates a purchase order document column of XMLType.

```sql
CREATE TABLE po_xml_tab(
   poid number,
   poDoc SYS.XMLTYPE);
```

**XMLType Creation Example 2 — Adding XMLType Columns**

You can alter tables to add XMLType columns as well. This is similar to any other datatype.

The following statement adds a new customer document column to the table,

```sql
ALTER TABLE po_xml_tab add (custDoc sys.XMLType);
```

**XMLType Creation Example 3 — Dropping XMLType Columns**

You can alter tables to drop XMLType columns, similar to any other datatype.

The following statement drops the custDoc column.

```sql
ALTER TABLE po_xml_tab drop (custDoc sys.XMLType);
```
Specifying Storage Characteristics on XMLType Columns

As previously mentioned, the XML data in a XMLType column is stored as a CLOB column. You can also specify LOB storage characteristics for the CLOB column. In the previous example, the warehouse spec column is an XMLType column.

Figure 5–1 illustrates the XMLType storage clause syntax.

Figure 5–1  XMLType Storage Clause Syntax

Table 5–2 explains the XMLType storage clause syntax.
You can specify storage characteristics on this column when creating the table as follows:

```sql
CREATE TABLE po_xml_tab(
    poid NUMBER(10),
    poDoc SYS.XMLTYPE
)
```

```sql
XMLType COLUMN poDoc
STORE AS CLOB {
    TABLESPACE lob_seg_ts
```
XMLType Storage in the Database

STORAGE (INITIAL 4096 NEXT 4096)
CHUNK 4096 NOCACHE LOGGING
);

The storage clause is also supported while adding columns to the table. If you want to add a new XMLType column to this table and specify the storage clause for that you can do the following:-

ALTER TABLE po_xml_tab  add(
    custDoc SYS.XMLTYPE
)
XMLType COLUMN custDoc
STORE AS CLOB {
    TABLESPACE lob_seg_ts
    STORAGE (INITIAL 4096 NEXT 4096)
    CHUNK 4096 NOCACHE LOGGING
};

See also: Oracle9i Application Developer’s Guide - Large Objects (LOBs) for more information about LOB storage options.

Specifying Constraints on XMLType Columns

You can specify the NOT NULL constraint on a XMLType column. For example:

CREATE TABLE po_xml_tab {
    poid number(10),
    poDoc sys.XMLType NOT NULL
};

prevents inserts such as:

INSERT INTO po_xml_tab (poDoc) VALUES (null);

You can also use the ALTER TABLE statement to change the NOT NULL information of a XMLType column, in the same way you would for other column types:

ALTER TABLE po_xml_tab MODIFY (poDoc NULL);
ALTER TABLE po_xml_tab MODIFY (poDoc NOT NULL);

Default values and other check constraints are not supported on this datatype.
XMLType Functions

Oracle9i has introduced two new SQL functions ExistsNode and Extract that operate on XMLType values.

existsNode() function uses the XMLType.existsNode() member function for its implementation. The syntax of ExistsNode function is:

```sql
existsNode ("XMLType_instance" IN sys.XMLType,
            "XPath_string" IN VARCHAR2) RETURN NUMBER
```

extract() function applies an XPath expression and returns an XMLType containing the resultant XML fragment. The syntax is:

```sql
extract ("XMLType_instance" IN sys.XMLType,
         "XPath_string" IN VARCHAR2) RETURN sys.XMLType;
```

**Note:** In this release, existsNode() and extract() SQL functions only use the functional implementation. In future releases, these functions will use new indices and be further optimized.

Table 5–3 lists all the XMLType and member functions, their syntax and descriptions.

You can use the SQL functions instead of the member functions ExistsNode and Extract inside any SQL statement. All the XMLType functions use the built-in C parser and processor to parse the XML data, validate it and apply XPath expressions over it. It also uses an optimized in-memory DOM tree to do processing (such as Extract).
### XMLType Functions

#### Table 5–3  XMLType Member and Static Functions

<table>
<thead>
<tr>
<th>XMLType Function</th>
<th>Syntax Summary</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>createXML()</strong></td>
<td>STATIC FUNCTION createXML(xmlval IN varchar2) RETURN sys.XMLType deterministic</td>
<td>Static function to create the XMLType instance from a string. Checks for well-formed XML value. Parameters: xmlval (IN) - string containing the XML document. Returns: A XMLType instance. String must contain a well-formed XML document. See also &quot;XMLType Query Example 6 — Extract fragments from XMLType&quot; on page 5-28, and other examples in this chapter.</td>
</tr>
<tr>
<td><strong>createXML()</strong></td>
<td>STATIC FUNCTION createXML(xmlval IN clob) RETURN sys.XMLType deterministic</td>
<td>Static function to create the XMLType instance from a CLOB. Checks for well-formed XML value. Parameters: xmlval (IN) - CLOB containing the XML document Returns: A XMLType instance. CLOB must contain a well-formed XML document. See &quot;XMLType Query Example 2 — Using extract() and existsNode()&quot; on page 5-24 and other examples in this chapter.</td>
</tr>
<tr>
<td><strong>existsNode()</strong></td>
<td>MEMBER FUNCTION existsNode(xpath IN varchar2) RETURN number deterministic</td>
<td>Given an XPath expression, checks if the XPath applied over the document can return any valid nodes. Parameters: xpath (IN) - the XPath expression to test Returns: 0 if the XPath expression does not return any nodes else 1. If the XPath string is null or the document is empty, then a value of 0 is returned. See also &quot;XMLType Query Example 2 — Using extract() and existsNode()&quot; on page 5-24.</td>
</tr>
<tr>
<td><strong>extract()</strong></td>
<td>MEMBER FUNCTION extract(xpath IN varchar2) RETURN sys.XMLType deterministic</td>
<td>Given an XPath expression, applies the XPath to the document and returns the fragment as a XMLType instance containing the result node(s). If the XPath does not result in any nodes, then the result is NULL. See also &quot;XMLType Query Example 6 — Extract fragments from XMLType&quot; on page 5-28.</td>
</tr>
</tbody>
</table>
### Table 5–3  XMLType Member and Static Functions (Cont.)

<table>
<thead>
<tr>
<th>XMLType Function</th>
<th>Syntax Summary</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>isFragment()</code></td>
<td><code>MEMBER FUNCTION</code></td>
<td>Checks if the document is really a fragment. A fragment might be present, if an EXTRACT or other operation was done on an XML document that may have resulted in many nodes. RETURNS: A numerical value 1 or 0 indicating if the XMLType instance contains a fragment or a well-formed document. See also &quot;XMLType Query Example 6 — Extract fragments from XMLType&quot; on page 5-28.</td>
</tr>
<tr>
<td></td>
<td><code>isFragment()</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>RETURN number</code></td>
<td></td>
</tr>
<tr>
<td><code>getClobVal()</code></td>
<td><code>MEMBER FUNCTION</code></td>
<td>Gets the document as a CLOB. RETURNS: A CLOB containing the serialized XML representation. Free the temporary CLOB after use. See also: &quot;XMLType Query Example 1 — Retrieve an XML Document as a CLOB&quot; on page 5-55.</td>
</tr>
<tr>
<td></td>
<td><code>getClobVal()</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>RETURN clob deterministic</code></td>
<td></td>
</tr>
<tr>
<td><code>getStringVal()</code></td>
<td><code>MEMBER FUNCTION</code></td>
<td>Gets the XML value as a string. RETURNS: A string containing the serialized XML representation, or in case of text nodes, the text itself. If the XML document is bigger than the maximum size of VARCHAR2, (4000 bytes), an error is raised at run time. See&quot;XMLType Delete Example 1 — Deleting Rows Using extract&quot; and also, &quot;How to use XMLType&quot; on page 5-4.</td>
</tr>
<tr>
<td></td>
<td><code>getStringVal()</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>RETURN varchar2 deterministic</code></td>
<td></td>
</tr>
<tr>
<td><code>getNumberVal()</code></td>
<td><code>MEMBER FUNCTION</code></td>
<td>Gets the numeric value pointed to by the XMLType as a number RETURNS: A number formatted from the text value pointed to by the XMLType instance. The XMLType must point to a valid text node that contains a numeric value. See also: &quot;XMLType Query Example 2 — Using extract() and existsNode()&quot; on page 5-24.</td>
</tr>
<tr>
<td></td>
<td><code>getNumberVal()</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>RETURN number deterministic</code></td>
<td></td>
</tr>
</tbody>
</table>
Manipulating XML Data in XMLType Columns

Since XMLType is a user-defined data type with functions defined on it, you can invoke functions on XMLType and obtain results. You can use XMLType wherever you use a user-defined type. This includes columns of tables, views, trigger body, type definitions, and so on.

You can perform the following manipulations (DML) on XML data in XMLType columns:

- Insert XML data
- Update XML data
- Delete XML data

Inserting XML Data into XMLType Columns

You can insert data into XMLType columns in the following ways:

- By using the INSERT statement (in SQL, PL/SQL, and Java)
- By using SQL*Loader

The XMLType columns can only store well-formed XML documents. Fragments and other non-well formed XML cannot be stored in such columns.

Using INSERT Statements

If you use the INSERT statement to insert XML data into XMLType, you need to first create XML documents to perform the insert with. You can create the insertable XML documents as follows:

1. Using XMLType constructors, SYS.XMLType.createXML(). This can be done in SQL, PL/SQL, and Java.
2. Using SYS_XMLGEN and SYS_XMLAGG SQL functions. This can be done in PL/SQL, Java.
**XMLType Insert Example 1- Using createXML() with CLOB**

The following examples use INSERT...SELECT and the XMLType.createXML() construct to first create an XML document and then insert the document into XMLType columns.

For example, if the po_clob_tab is a table containing a CLOB that stores an XML document,

```sql
CREATE TABLE po_clob_tab
(
    poid number,
    poClob CLOB
);

-- some value is present in the po_clob_tab
INSERT INTO po_clob_tab
VALUES(100, '<?xml version="1.0"?>
    <PO pono="1">
        <PNAME>Po_1</PNAME>
        <CUSTNAME>John</CUSTNAME>
        <SHIPADDR>
            <STREET>1033, Main Street</STREET>
            <CITY>Sunnyvalue</CITY>
            <STATE>CA</STATE>
        </SHIPADDR>
    </PO>');</
```

Now you can insert a purchase order XML document into table, po_xml_tab by simply creating an XML instance from the CLOB data stored in the other po_clob_tab,

```sql
INSERT INTO po_xml_tab
SELECT poid, sys.XMLType.createXML(poClob)
FROM po_clob_tab;
```

Note that we could have gotten the clob value from any expression including functions which can create temporary CLOBs or select out CLOBs from other table or views.

**XMLType Insert Example 2 - Using createXML() with string**

This example inserts a purchase order into table, po_tab using the createXML() construct.

```sql
insert into po_xml_tab
```
Manipulating XML Data in XMLType Columns

VALUES(100, sys.XMLType.createXML('<?xml version="1.0"?><PO pono="1"><PNAME>Po_1</PNAME><CUSTNAME>John</CUSTNAME><SHIPADDR><STREET>1033, Main Street</STREET><CITY>Sunnyvale</CITY><STATE>CA</STATE></SHIPADDR></PO>'));

Here the XMLType was created using the createXML function by passing in a string literal.

**XMLType Insert Example 3 - Using SYS_XMLGEN()**

This example inserts PurchaseOrder into table, po_xml_tab by generating it using the SYS_XMLGEN() SQL function which is explained later in this chapter. Assume that the PO is an object view that contains a purchase order object. The whole definition of the PO view is given in "DBMS_XMLGEN Example 5: Generating a Purchase Order From the Database in XML Format" on page 5-55.

```sql
INSERT into po_xml_tab
SELECT SYS_XMLGEN(value(p),
          sys.xmlgenformatType.createFormat('PO'))
FROM po p
WHERE p.pono=2001;
```

The SYS_XMLGEN creates an XMLType from the purchase order object which is then inserted into the po_xml_tab table.

**Updating XML Data in XMLType Columns**

You can only update the whole XML document. You can perform the update in SQL, PL/SQL, or Java.

See also "XMLType Java Example 4: Updating an Element in XMLType Column" on page 5-36, for updating XMLType through Java.

**XMLType Update Example 1 — Updating Using createXML()**

This example updates the XMLType using the createXML() construct. It updates only those documents whose purchase order number is 2001.

```sql
update po_xml_tab e
```
set e.poDoc = sys.XMLType.createXML(''"<?xml version="1.0"?>
<PO pono="2">
  <PNAME>Po_2</PNAME>
  <CUSTNAME>Nance</CUSTNAME>
  <SHIPADDR>
    <STREET>2 Avocet Drive</STREET>
    <CITY>Redwood Shores</CITY>
    <STATE>CA</STATE>
  </SHIPADDR>
</PO>'
WHERE e.po.extract('/PO/PONO/text()').getNumberVal() = 2001;

---

**Deleting XML Data**

DELETEs on the row containing the XMLType column are handled in the same way as for any other datatype.

**XMLType Delete Example 1 — Deleting Rows Using extract**

For example, to delete all purchase order rows with a purchase order name of “Po_2”, you can execute a statement such as:

```
DELETE from po_xml_tab e
WHERE e.poDoc.extract('/PO/PNAME/text()').getStringVal() = 'Po_2';
```

**Using XMLType Inside Triggers**

You can use the NEW and OLD binds inside triggers to read and modify the XMLType column values. In the case of INSERTs and UPDATE statements, you can modify the NEW value to change the value being inserted.

**XMLType Trigger Example 1 -**

For instance, you can write a trigger to change the purchase order to be a different one if it does not contain a shipping address.

```
CREATE OR REPLACE TRIGGER po_trigger
    BEFORE insert or update on po_xml_tab for each row
```
pono Number;
begin

if INSERTING then
  if :NEW.poDoc.existsnode('//SHIPADDR') = 0 then
    :NEW.poDoc := sys.xmltype.createXML('<PO>INVALID_PO</PO>'); end if;
  end if;

  -- when updating, if the old poDoc has purchase order number
  -- different from the new one then make it an invalid PO.
  if UPDATING then
    if :OLD.poDoc.extract('//PONO/text()').getNumberVal() !=
      :NEW.poDoc.extract('//PONO/text()').getNumberVal() then
      :NEW.poDoc := sys.xmltype.createXML('<PO>INVALID_PO</PO>');
    end if;
  end if;
end;
/

This example is only for illustration purposes. You can use the XMLType value to perform useful operations inside the trigger, such as validation of business logic or rules that the XML document should conform to, auditing, and so on.

Selecting and Querying XML Data

You can query XML Data from XMLType columns in the following ways:

- By selecting XMLType columns through SQL, PL/SQL, or Java.
- By querying XMLType columns directly and using extract() and/or existsNode().
- By using Text operators to query the XML content

Selecting XML data

You can select the XMLType data through PL/SQL or Java. You can also use the getClobVal(), getStringVal() or getNumberVal() functions to get out the XML as a CLOB, varchar or a number respectively.

XMLType Query Example 1 — Retrieve an XML Document as a CLOB

This example shows how to select an XMLType column through SQL*Plus.
set long 2000

select e.poDoc.getClobval() AS poXML
from po_xml_tab e;

POXML
---------------------
<?xml version="1.0"?>
<PO pono="2">
  <PNAME>Po_2</PNAME>
  <CUSTNAME>Nance</CUSTNAME>
  <SHIPADDR>
    <STREET>2 Avocet Drive</STREET>
    <CITY>Redwood Shores</CITY>
    <STATE>CA</STATE>
  </SHIPADDR>
</PO>

**Querying XML data**

We can query the XMLType data and extract portions of it using the ExistsNode and Extract functions. Both these functions use a limited set of the W3C standard XPath to navigate the document.

**Using XPath Expressions for Searching**

XPath is a W3C standard way to navigate XML documents. XPath models the XML document as a tree of nodes. It provides a rich set of operations to “walk” the tree and to apply predicates and node test functions. Applying an XPath expression to an XML document can result in a set of nodes. For instance, 

\[
/PO/PONO
\]

selects out all the “PONO” child elements under the “PO” root element of the document.

Here are some of the common constructs used in XPath:-

- The “/” denotes the root of the tree in an XPath expression. e.g. /PO refers to the child of the root node whose name is “PO”.

- The “/” is also used as a path separator to identify the children node of any given node. e.g. /PO/PNAME identifies the purchase order name element which is a child of the root element.

- The “/” is used to identify all descendants of the current node. e.g. PO//ZIP matches any zip code element under the “PO” element.
The "*" is used as a wildcard to match any child node. e.g. /PO/*/STREET would match any street element that is a grandchild of the “PO” element.

The [ ] are used to denote predicate expressions. XPath supports a rich list of binary operators such as OR, AND and NOT. e.g. /PO[PONO=20 and PNAME="PO_2'']/SHIPADDR selects out the shipping address element of all purchase orders, whose purchase order number is 20 and whose purchase order name is “PO_2”

The [ ] is also used for denoting an index into a list. For instance, /PO/PONO[2] identifies the second purchase order number element under the "PO" root element.

**Supported XPath constructs**

Oracle’s Extract and ExistsNode functions support a limited set of XPath expressions. XPath constructs supported in this release are:

- Child traversals /PO/SHIPADDR/STREET
- Attribute traversals /PO/@PONO
- Index access /PO/PONO[2]
- Wild card searches /PO/*/STREET
- Descendant searches PO//STREET
- Node text functions - /PO/PNAME/text()

Only the non-order dependant axes such as the child, descendant axes are supported. No sibling or parent axes are supported.

---

**Note:** Extract and ExistsNode functions do not yet support multi-byte character sets.

---

**Predicates are Not Supported** For this release, XMLType does not support any predicates. If you need predicate support, you can rewrite the function into multiple functions with the predicates expressed in SQL, when possible.

Finally, the XPath must identify a single or a set of element, text or attribute nodes. The result of the XPath cannot be a boolean expression.

**existsNode() Function with XPath**

The `existsNode` function on XMLType, checks if the given XPath evaluation results in at least a single XML element or text node. If so, it returns the numeric value 1 otherwise it returns a 0. For example, consider an XML document such as:
An XPath expression such as /PO/PNAME, results in a single node and hence the ExistsNode will return true for that XPath. This is the same with /PO/PNAME/text() which results in a single text node. The XPath, /PO/@pono also returns a value.

An XPath expression such as, /PO/POTYPE does not return any nodes and hence an ExistsNode on this would return the value 0.

Hence, the ExistsNode() member function can be directly used in the following ways:

- In queries as will be shown in the next few examples
- To create functional indexes to speed up evaluation of queries

Extract Function with XPath

The Extract function on XMLType, extracts the node or a set of nodes from the document identified by the XPath expression. The extracted nodes may be elements, attributes or text nodes. When extracted out all text nodes are collapsed into a single text node value.

The XMLType resulting from applying an XPath through Extract need not be a well-formed XML document but can contain a set of nodes or simple scalar data in some cases. You can use the getStringVal() or getNumberVal() methods on XMLType to extract this scalar data.

For example, the XPath expression /PO/PNAME identifies the PNAME element inside the XML document shown above. The expression /PO/PNAME/text() on the other hand refers to the text node of the PNAME element. Note that the latter is still considered an XMLType. In other words, EXTRACT(poDoc,'/PO/PNAME/text()') still returns an XMLType instance though the instance may actually contain only text. You can use the getStringVal() to get the text value out as a VARCHAR2 result.
Use the text() node test function to identify text nodes in elements before using the
getStringVal() or getNumberVal() to convert them to SQL data. Not having the
text() node would produce an XML fragment. For example, the XPath expression
/PO/PNAME identifies the fragment <PNAME>PO_1</PNAME> whereas the
expression /PO/PNAME/text() identifies the text value “PO_1”.

You can use the index mechanism to identify individual elements in case of
repeated elements in an XML document. For example if we had an XML document
such as,

```xml
<PO>
  <PONO>100</PONO>
  <PONO>200</PONO>
</PO>
```

you can use //PONO[1] to identify the first “PONO” element (with value 100) and

The result of the extract is always an XMLType. If applying the XPath produces an
empty set, then the Extract returns a NULL value.

Hence, the extract() member function can be used in a number of ways,
including the following:

- Extracting numerical values on which functional indexes can be created
to speed up processing
- Extracting collection expressions to be used in the FROM clause of SQL
  statements
- Extracting fragments to be later aggregated to produce different documents

**XMLType Query Example 2 — Using extract() and existsNode()**

Assume the po_xml_tab table which contains the purchase order id and the
purchase order XML columns - and assume that the following values are inserted
into the table,

```sql
INSERT INTO po_xml_tab values (100,
  sys.xmltype.createxml('<?xml version="1.0"?>
  <PO>
    <PONO>221</PONO>
    <PNAME>PO_2</PNAME>
  </PO>'));
```

```
INSERT INTO po_xml_tab values (200,
  sys.xmltype.createxml('<?xml version="1.0"?>
  <PO>
```

5-24  Oracle9i Application Developer’s Guide - XML, Release 1 (9.0.1)
Now we can extract the numerical values for the purchase order numbers using the EXTRACT function.

```sql
SELECT e.poDoc.extract('//PONO/text()').getNumberVal() as pono
FROM po_xml_tab e
WHERE e.podoc.existsnode('/PO/PONO') = 1 AND poid > 1;
```

Here `extract()` extracts the contents of tag, purchase order number, “PONO”. `existsnode()` finds those nodes where there exists “PONO” as a child of “PO”.

Note the use of the `text()` function to return only the text nodes. The `getNumberVal()` function can convert only text values into numerical quantity.

**See Also:** "XMLType Functions" on page 5-13.

**XMLType Query Example 3 — Querying Transient XMLType data**

The following example shows how you can select the XML data and query it inside PL/SQL.

```sql
-- create a transient instance from the purchase order table and then
-- perform some extraction on it...
declare
  poxml SYS.XMLType;
  cust SYS.XMLType;
  val VARCHAR2;
begin

  -- select the adt instance
  select poDoc into poxml
  from po_xml_tab p where p.poid = 100;
  
  -- do some traversals and print the output
  cust := poxml.extract('//SHIPADDR');
  
  -- do something with the customer XML fragment
  val := cust.getStringVal();
  dbms_output.put_line(' The customer XML value is ' || val);

end;
/
XMLType Query Example 4 — Extracting data from XML

The following example shows how you can extract out data from a purchase order XML and insert it into a SQL relation table.

Assume the following relational tables,

```sql
CREATE TABLE cust_tab
(
  custid number primary key,
  custname varchar2(20)
);
insert into cust_tab values (1001, "John Nike");

CREATE table po_rel_tab
(
  pono number,
  pname varchar2(100),
  custid number references cust_tab
  shipstreet varchar2(100),
  shipcity varchar2(30),
  shipzip varchar2(20)
);
```

You can write a simple PL/SQL block to transform any XML of the form,

```xml
<?xml version = '1.0'?><PO><PONO>2001</PONO><CUSTOMER CUSTNAME="John Nike"/><SHIPADDR><STREET>323 College Drive</STREET><CITY>Edison</CITY><STATE>NJ</STATE><ZIP>08820</ZIP></SHIPADDR></PO>
```

into the relational tables, using the Extract functions.

Here is a SQL example, (assuming that the XML described above is present in the po_xml_tab) -

```sql
insert into po_rel_tab as
select p.poDoc.extract('/PO/PONO/text()').getnumberval(),
  p.poDoc.extract('/PO/PNAME/text()').getstringval(),
```

```sql
```
-- get the customer id corresponding to the customer name
( SELECT custid
FROM cust_tab c
WHERE c.custname =
  p.poDoc.extract('/PO/CUSTOMER/@CUSTNAME').getstringval()
),
p.poDoc.extract('/PO/SHIPADDR/STREET/text()').getstringval(),
p.poDoc.extract('//CITY/text()').getstringval(),
p.poDoc.extract('//ZIP/text()').getstringval(),
from po_xml_tab p;

The po_tab would now have the following values,

<table>
<thead>
<tr>
<th>PONO</th>
<th>PNAME</th>
<th>CUSTID</th>
<th>SHIPSTREET</th>
<th>SHIPCITY</th>
<th>SHIPZIP</th>
</tr>
</thead>
<tbody>
<tr>
<td>2001</td>
<td>1001</td>
<td>323 College Drive</td>
<td>Edison</td>
<td>08820</td>
<td></td>
</tr>
</tbody>
</table>

Note how the PNAME is null, since the input XML document did not have the element called PNAME under PO. Also, note that we have used the //CITY to search for the city element at any depth.

We can do the same in an equivalent fashion inside a PL/SQL block-

```
declar
  poxml SYS.XMLType;
  cname varchar2(200);
  pono number;
  pname varchar2(100);
  shipstreet varchar2(100);
  shipcity varchar2(30);
  shipzip varchar2(20);
begin
  -- select the adt instance
  select poDoc into poxml from po_xml_tab p;
  cname := poxml.extract('/CUSTOMER/@CUSTNAME').getstringval();
  pono := poxml.extract('/PO/PONO/text()').getnumberval(),
  pname := poxml.extract('/PO/PNAME/text()').getstringval(),
  shipstreet := poxml.extract('/PO/SHIPADDR/STREET/text()').getstringval(),
  shipcity := poxml.extract('//CITY/text()').getstringval(),
```
shipzip := po迅猛.extract('//ZIP/text()').getstringval(),

insert into po_rel_tab
values (pono, pname,
       (select custid from cust_tab c where custname = cname),
       shipstreet, shipcity, shipzip);
end;
/

XMLType Query Example 5 — Using extract() to Search
Using Extract, Existsnode functions, you can perform a variety of operations on the
column, as follows:

select e.poDoc.extract('/PO/PNAME/text()').getStringVal() PNAME
from po_xml_tab e
where e.poDoc.existsNode('/PO/SHIPADDR') = 1 and
  e.poDoc.extract('/PO/PONO/text()').getNumberVal() = 300 and
  e.poDoc.extract('/@CUSTNAME').getStringVal() like '%John%';

This SQL statement extracts the purchase order name “PNAME” from the purchase
order element PO, from all the documents which contain a shipping address and
whose purchase order number is 300 and the customer name “CUSTNAME”
contains the string “John”.

XMLType Query Example 6 — Extract fragments from XMLType
The extract() member function extracts the nodes identified by the XPath
expression and returns an XMLType containing the fragment. Here, the result of the
traversal may be a set of nodes, a singleton node, or a text value. You can check if
the result is a fragment by using the isFragment() function on the XMLType. For
example:

select e.po.extract('/PO/SHIPADDR/STATE').isFragment()
  from po_xml_tab e;
The previous SQL statement would return 0, since the extraction 
/PO/SHIPADDR/STATE returns a singleton well formed node which is not a 
fragment.

On the other hand, an XPath such as, /PO/SHIPADDR/STATE/text() would be 
considered a fragment, since it is not a well-formed XML document.

**Querying XMLType Data using Text Operators**

Oracle Text index works on CLOB and VARCHAR columns. It has been extended 
in Oracle9i to work on XMLType columns as well. The default behavior of Oracle 
Text index is to automatically create XML sections, when defined over XMLType 
columns. It also provides the CONTAINS operator which has been extended to 
support XPath.

**Creating Text index over XMLType columns**

Text index can be created by using the CREATE INDEX with the INDEXTYPE 
 specification as with other CLOB or VARCHAR columns. However, since the 
XMLType is implemented as a virtual column, the text index is created using the 
functional index mechanism.

This requires that to create and use the text index in queries, in additional to having 
the privileges to create indexes and the privileges necessary to create text indexes, 
you would need to also need to have the privileges and settings necessary to create 
functional indexes. This includes

- **QUERY_REWRITE privilege** - You must have this privilege granted to create 
text indexes on XMLType columns in your own schema. If you need to create 
text indexes on XMLType columns in other schemas or on tables residing in 
other schemas, you must have the GLOBAL_QUERY_REWRITE privilege 
granted.

- **QUERY_REWRITE_ENABLED parameter** must be set to true.

---

**Note:** You cannot insert fragments into XMLType columns. You 
can use the SYS_XMLGEN function to convert a fragment into a 
well formed document by adding an enclosing tag. See "SYS_ 
XMLGEN" on page 5-63. You can, however, query further on the 
fragment using the various XMLType functions. 

---
QUERY_REWRITE_INTEGRITY must be set to trusted for the queries to be rewritten to use the text index.

See Also:
- Chapter 8, "Searching XML Data with Oracle Text"
- Oracle Text Reference
- Oracle Text Application Developer's Guide

Differences between CONTAINS and ExistsNode/Extract
There are certain differences with regard to the XPath support inside CONTAINS and that supported through the ExistsNode and Extract functions.

- In this release, the XPath supported by the Oracle Text index is more powerful than the functional implementation, as it can satisfy certain equality predicates as well.
- Since Oracle Text index ignores spaces, the XPath expression may not yield accurate results when spaces are significant.
- Oracle Text index also supports certain predicate expressions with string equality, but cannot support numerical and range comparisons.
- One other limitation is that the Oracle Text index may give wrong result if the XML document only has tag names and attribute names without any text. For example in the case of the following document,

```xml
<A>
  <B>
    <C>
    </C>
  </B>
  <D>
    <E>
    </E>
  </D>
</A>
```

the XPath expression - A/B/E will falsely match the above XML document.

- Both the functional and the Oracle Text index support navigation. Thus you can use the text index as a primary filter, to filter out all the documents that can potentially match the criterion in an efficient manner, and then apply secondary
filters such as `existsNode()` or `extract()` operations on the remainder of the documents.

**Indexing XMLType columns**

We can create the following indexes using `XMLType` to speed up query evaluation.

- **Functional Indexes with XMLType.** Queries can be speeded up by building functional indexes on the `ExistsNode` or the `Extracted` portions of an XML document.

  **Example of Functional Indexes on Extract operation:**

  For instance to speed up the search on the query,

  ```sql
  SELECT * FROM po_xml_tab e
  WHERE e.poDoc.extract('//PONO/text()').getNumberVal() = 100;
  ```

  we can create a functional index on the `Extract` function as:

  ```sql
  CREATE INDEX city_index ON po_xml_tab
  (poDoc.extract('//PONO/text()').getNumberVal());
  ```

  With this index, the SQL query would use the functional index to evaluate the predicate instead of parsing the XML document per row and evaluating the XPath expression.

  **Example of Functional indexes on ExistsNode:**

  We can also create bitmapped functional indexes to speed up the evaluation of the operators. In particular, the `ExistsNode` is best suited, since it returns a value of 1 or 0 depending on whether the XPath is satisfied in the document or not.

  For instance to speed up the query, that searches for whether the XML document contains an element called `Shipping address` at any level -

  ```sql
  SELECT * FROM po_xml_tab e
  WHERE e.poDoc.existsNode('//SHIPADDR') = 1;
  ```

  we can create a bitmapped functional index on the `ExistsNode` function as:

  ```sql
  CREATE INDEX po_index ON po_xml_tab
  (poDoc.existsNode('//SHIPADDR'));
  ```

  to speed up the query processing.
Creating Text Indexes on XMLType Columns. As explained earlier, you can create text indexes on the XMLType column. The index uses the PATH SECTION GROUP as the default section group when indexing XMLType columns. This is the default and can be overridden during index creation.

CREATE INDEX po_text_index ON po_xml_tab(poDoc) indextype is ctxsys.context;

You can do text operations such as CONTAINS and SCORE.. on this XMLType column. In Oracle9i CONTAINS function has been enhanced to support XPath using two new operators, INPATH and HASPATH.

INPATH checks if the given word appears within the path specified and HASPATH checks if the given XPath is present in the document.

SELECT * FROM po_xml_doc w
WHERE CONTAINS(w.poDoc,
 'haspath(/PO[./@CUSTNAME="John Nike"]') > 0;

Java Access to XMLType (oracle.xdb.XMLType)

XMLType can be accessed through the oracle.xdb.XMLType class in Java. The class provides functions similar to the XMLType in PL/SQL. The oracle.xdb.XMLType is a subclass of Oracle JDBC’s oracle.sql.OPAQUE class.

Since the XMLType in the server is implemented as an opaque type, you would need to use the getOPAQUE call in JDBC to get the opaque type instance from the JDBC ResultSet and then create an XMLType instance out of it. In future releases, JDBC would instantiate XMLTypes automatically.

You can bind XMLType to any XML data instance in JDBC using the setObject call in the java.sql.PreparedStatement interface.

The functions defined in the oracle.xdb.XMLType class help to retrieve the XML data in Java. Use the SQL functions to perform any queries etc.

XMLType Java Example 1: Selecting XMLType data in Java

You can select the XMLType in java in one of 2 ways,

- Use the getClobVal() or getStringVal() in SQL and get the result as a oracle.sql.CLOB or java.lang.String in Java. Here is a snippet of Java code that shows how to use this,
DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());

Connection conn =
    DriverManager.getConnection("jdbc:oracle:oci8:@", "scott", "tiger");

OraclePreparedStatement stmt =
    (OraclePreparedStatement) conn.prepareStatement(
        "select e.poDoc.getClobVal() poDoc, "
        +
        e.poDoc.getStringVal() poString "+
        " from po_xml_tab e");

ResultSet rset = stmt.executeQuery();
OracleResultSet orset = (OracleResultSet) rset;

// the first argument is a CLOB
oracle.sql.CLOB clb = orset.getCLOB(1);

// the second argument is a string..
String poString = orset.getString(2);

// now use the CLOB inside the program..

Use the getOPAQUE() call in the PreparedStatement to get the whole XMLType instance and use the XMLType constructor to construct an oracle.xdb.XMLType class out of it. Then you can use the Java functions on the XMLType class to access the data.

import oracle.xdb.XMLType;
...

OraclePreparedStatement stmt =
    (OraclePreparedStatement) conn.prepareStatement(
        "select e.poDoc from po_xml_tab e");

ResultSet rset = stmt.executeQuery();
OracleResultSet orset = (OracleResultSet) rset;

// get the XMLType
XMLType poxml = XMLType(orset.getOPAQUE(1));

// get the XML as a string...
String poString = poxml.getStringVal();
XMLType Java Example 2: Updating XMLType Data in Java

You can insert an XMLType in java in one of 2 ways,

- Bind a CLOB or a string to an insert/update/delete statement and use the createXML() constructor inside SQL to construct the XML instance,

```java
OraclePreparedStatement stmt =
    (OraclePreparedStatement) conn.prepareStatement(
        "update po_xml_tab set poDoc = sys.XMLType.createXML(?) ");

// the second argument is a string..
String poString = "<PO><PONO>200</PONO><PNAME>PO_2</PNAME></PO>";

// now bind the string..
stmt.setString(1,poString);
stmt.execute();
```

- Use the setObject() (or setOPAQUE()) call in the PreparedStatement to set the whole XMLType instance.

```java
import oracle.xdb.XMLType;
...
OraclePreparedStatement stmt =
    (OraclePreparedStatement) conn.prepareStatement(
        "update po_xml_tab set poDoc = ? ");

// the second argument is a string..
String poString = "<PO><PONO>200</PONO><PNAME>PO_2</PNAME></PO>";
XMLType poXML = XMLType.createXML(conn, poString);

// now bind the string..
stmt setObject(1,poXML);
stmt execute();
```

XMLType Java Example 3: Getting Metadata on XMLType

When selecting out XMLType values, JDBC describes the column as an OPAQUE type. You can select the column type name out and compare it with “XMLTYPE” to check if you are dealing with an XMLType,

```java
import oracle.sql.*;
import oracle.jdbc.*;
...
OraclePreparedStatement stmt =
    (OraclePreparedStatement) conn.prepareStatement(
```
"select poDoc from po_xml_tab";  

OracleResultSet rset = (OracleResultSet)stmt.executeQuery();

// Now, we can get the resultset metadata  
OracleResultSetMetaData mdata =  
    (OracleResultSetMetaData)rset.getMetaData();

// Describe the column = the column type comes out as OPAQUE  
// and column type name comes out as SYS.XMLTYPE  
if (mdata.getColumnType(1) == OracleTypes.OPAQUE &&  
    mdata.getColumnTypeName(1).compareTo("SYS.XMLTYPE") == 0)
{
    // we know it is an XMLtype..
}
XMLType Java Example 4: Updating an Element in XMLType Column

This example updates the “DISCOUNT” element inside PurchaseOrder stored in a XMLType column. It uses Java (JDBC) and the oracle.xdb.XMLType class. This example also shows you how to insert/update/delete XMLTypes using Java (JDBC).
It uses the parser to update an in-memory DOM tree and write the updated XML value to the column.

```java
-- create po_xml_hist table to store old PurchaseOrders
create table po_xml_hist (
  xpo sys.xmltype
);

/*
   DESCRIPTION
   Example for oracle.xdb.XMLType

   NOTES
   Have classes12.zip, xmlparserv2.jar, and oraxdb.jar in CLASSPATH
*/

import java.sql.*;
import java.io.*;
import oracle.xml.parser.v2.*;
import org.xml.sax.*;
import org.w3c.dom.*;
import oracle.jdbc.driver.*;
import oracle.sql.*;
import oracle.xdb.XMLType;

public class tkxmtpje
{
  static String conStr = "jdbc:oracle:oci8:@";
  static String user = "scott";
  static String pass = "tiger";
  static String qryStr =
    "SELECT x.poDoc from po_xml_tab x "+
    "WHERE x.poDoc.extract('/PO/PONO/text()').getNumberVal()=200";

  static String updateXML(String xmlTypeStr)
  {
    System.out.println("\n=================================");
    System.out.println("xmlType.getStringVal():");
    System.out.println(xmlTypeStr);
  }

  public static void main(String[] args)
  {
    // Update XML value
    updateXML("<xmlType>
      <PO>
        <PONO>1000</PONO>
      </PO>
    </xmlType>");
  }
}
```

Database Support for XML  5-37
```
System.out.println("================================");
String outXML = null;
try{
    DOMParser parser = new DOMParser();
    parser.setValidationMode(false);
    parser.setPreserveWhitespace(true);
    parser.parse(new StringReader(xmlTypeStr));
    System.out.println("xmlType.getStringVal(): xml String is well-formed");
    XMLDocument doc = parser.getDocument();
    NodeList nl = doc.getElementsByTagName("DISCOUNT");
    for(int i=0;i<nl.getLength();i++){
        XMLElement discount = (XMLElement)nl.item(i);
        XMLNode textNode = (XMLNode)discount.getFirstChild();
        textNode.setNodeValue(10);
    }
    StringWriter sw = new StringWriter();
    doc.print(new PrintWriter(sw));
    outXML = sw.toString();
    //print modified xml
    System.out.println("\n================================");
    System.out.println("Updated PurchaseOrder:");
    System.out.println(outXML);
    System.out.println("================================");
} catch (Exception e) {
    e.printStackTrace(System.out);
    return outXML;
}
```

```java
public static void main(String args[]) throws Exception {
    try{
        System.out.println("qryStr=": qryStr);
        DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
    }
    System.out.println("qryStr=": qryStr);
    DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
}
Connection conn =
    DriverManager.getConnection("jdbc:oracle:oci8:@", user, pass);

Statement s = conn.createStatement();
OraclePreparedStatement stmt;

ResultSet rset = s.executeQuery(qryStr);
OracleResultSet orset = (OracleResultSet) rset;
while(orset.next()){  
    //retrieve PurchaseOrder xml document from database
    XMLType xt = XMLType.createXML(orset.getOPAQUE(1));

    //store this PurchaseOrder in po_xml_hist table
    stmt = (OraclePreparedStatement)conn.prepareStatement(
        "insert into po_xml_hist values(?)");
    stmt.setObject(1,xt);  // bind the XMLType instance
    stmt.execute();

    //update "DISCOUNT" element
    String newXML = updateXML(xt.getStringVal());
    // create a new instance of an XMLtype from the updated value
    xt = XMLType.createXML(conn,newXML);

    // update PurchaseOrder xml document in database
    stmt = (OraclePreparedStatement)conn.prepareStatement(
        "update po_xml_tab x set x.poDoc =? where x.poDoc.extract('/PO/PONO/text()').getNumberVal()=200");
    stmt.setObject(1,xt);  // bind the XMLType instance
    stmt.execute();
    conn.commit();
    System.out.println("PurchaseOrder 200 Updated!");
}

//delete PurchaseOrder 1001
s.execute("delete from po_xml x " +
"where x.xpo.extract(?'PurchaseOrder/PONO/text()').getNumberVal()==1001");
System.out.println("PurchaseOrder 1001 deleted!");
}
catch( Exception e )
{
    e.printStackTrace(System.out);
}

----------------------
-- list PurchaseOrders
----------------------

set long 20000
set pages 100
select x.xpo.getClobVal()
from po_xml x;

Here is the resulting updated purchase order in XML:

<?xml version = '1.0'?>
<PurchaseOrder>
  <PONO>200</PONO>
  <CUSTOMER>
    <CUSTNO>2</CUSTNO>
    <CUSTNAME>John Nike</CUSTNAME>
    <ADDRESS>
      <STREET>323 College Drive</STREET>
      <CITY>Edison</CITY>
      <STATE>NJ</STATE>
      <ZIP>08820</ZIP>
    </ADDRESS>
    <PHONELIST>
      <VARCHAR2>609-555-1212</VARCHAR2>
      <VARCHAR2>201-555-1212</VARCHAR2>
    </PHONELIST>
  </CUSTOMER>
  <ORDERDATE>20-APR-97</ORDERDATE>
  <SHIPDATE>20-MAY-97 12.00.00.000000 AM</SHIPDATE>
  <LINEITEMS>
    <LINEITEM_TYP LineItemNo="1">
      <ITEM StockNo="1004">
        <PRICE>6750</PRICE>
        <TAXRATE>2</TAXRATE>
      </ITEM>
    </LINEITEM_TYP>
  </LINEITEMS>
</PurchaseOrder>
Installing and Using oracle.xdb.XMLType Class

The oracle.xdb.XMLType is available in the xdb_g.jar file in the ORACLE_HOME/rdbms/jlib where ORACLE_HOME refers to the Oracle home directory.

Using oracle.xdb.XMLType inside JServer:

This class is pre-loaded into the JServer and is available in the SYS schema.

It is not loaded however, if you have upgraded your database from an earlier version. If you need to upload the class into the JServer, you would need to run the initxdbi.sql file located in the ORACLE_HOME/rdbms/admin directory, while connected as SYS.

Using oracle.xdb.XMLType on the client:

If you need to use the oracle.xdb.XMLType class on the client side, then ensure that the xdb_g.jar file is listed in your CLASSPATH environment variable.
Native XML Generation

Oracle9i supports native XML generation with the following packages and functions:

- **DBMS.XMLGEN** PL/SQL supplied package. Gets XML from SQL queries. This is written in C and linked to the server for enhanced performance.

- SQL functions for getting XML from SQL queries are:
  - **SYS.XMLGEN** operates on rows, generating XML documents
  - **SYS.XMLAGG** operates on groups of rows, aggregating several XML documents into one

**DBMS.XMLGEN**

DBMS.XMLGEN creates XML documents from any SQL query by mapping the database query results into XML. It gets the XML document as a CLOB. It provides a “fetch” interface whereby you can specify 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, and so on.

The parameters of the package can restrict the number of rows retrieved, the enclosing tag names. To summarize, DBMS.XMLGEN PL/SQL package allows you:

- To create an XML document instance from any SQL query and get the document as a CLOB
- A “fetch” interface with maximum rows and rows to skip. For example, the first fetch could retrieve a maximum of 10 rows, skipping the first four. This is useful for pagination in web-based applications.
- Options for changing tag names for ROW, ROWSET, and so on.

**See Also:** “Generating XML with XSU’s OracleXMLQuery” on page 7-2, in Chapter 7, "XML SQL Utility (XSU)", and compare the functionality of OracleXMLQuery with DBMS.XMLGEN.

Sample Query Result

The following shows a sample result from executing the “select * from scott.emp” query on a database:

```xml
<?xml version="1.0"?>
```
The result of the getXML() using 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, maps as a subelement of the ROW element.
- All datatypes other than CURSOR expressions are supported by DBMS_XMLGen. Binary data is transformed to its hexadecimal representation.

As the document is in a CLOB, it has the same encoding as the database character set. If the database character set is SHIFTJIS, then the XML document is SHIFTJIS.

**DBMS_XMLGEN Calling Sequence**

Figure 5–2 summarizes the DBMS_XMLGEN calling sequence.
Here is `DBMS_XMLGEN`'s calling sequence:

1. Get the context from the package by supplying a SQL query and calling the `newContext()` call.

2. Pass the context to all the procedures/functions in the package to set the various options. For example to set the ROW element’s name, use `setRowTag(ctx)`, where `ctx` is the context got from the previous `newContext()` call.

3. Get the XML result, using the `getXML()`. By setting the maximum rows to be retrieved per fetch using the `setMaxRows()` call, you can call this function repeatedly, getting the maximum number of row set per call. The function returns a null CLOB if there are no rows left in the query.

   `getXML()` always returns an XML document, even if there were no rows to retrieve. If you want to know if there were any rows retrieved, use the function `getNumRowsProcessed()`.
4. You can reset the query to start again and repeat step 3.
5. Close the `closeContext()` to free up any resource allocated inside.

Table 5–5 summarizes DBMS_XMLGEN functions and procedures.

<table>
<thead>
<tr>
<th>Function or Procedure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DBMS_XMLGEN Type definitions</strong></td>
<td>The context handle used by all functions. DTD or schema specifications:</td>
</tr>
<tr>
<td>SUBTYPE ctxHandle IS NUMBER</td>
<td></td>
</tr>
<tr>
<td>■ NONE CONSTANT NUMBER:= 0; -- supported for this release.</td>
<td></td>
</tr>
<tr>
<td>■ DTD CONSTANT NUMBER:= 1;</td>
<td></td>
</tr>
<tr>
<td>■ SCHEMA CONSTANT NUMBER:= 2;</td>
<td></td>
</tr>
<tr>
<td>Can be used in getXML function to specify whether to generate a DTD or XML Schema or none. Only the NONE specification is supported in the getXML functions for this release.</td>
<td></td>
</tr>
<tr>
<td><strong>FUNCTION PROTOTYPES</strong></td>
<td>Given a query string, generate a new context handle to be used in subsequent functions.</td>
</tr>
<tr>
<td>newContext()</td>
<td>Returns a new context</td>
</tr>
<tr>
<td>FUNCTION newContext(queryString IN VARCHAR2)</td>
<td>PARAMETERS: queryString (IN)- the query string, the result of which needs to be converted to XML</td>
</tr>
<tr>
<td></td>
<td>RETURNS: Context handle. Call this function first to obtain a handle that you can use in the getXML() and other functions to get the XML back from the result.</td>
</tr>
<tr>
<td>setRowTag()</td>
<td>Sets the name of the element separating all the rows. The default name is ROW.</td>
</tr>
<tr>
<td>PROCEDURE setRowTag(ctx IN ctxHandle, rowTag IN VARCHAR2);</td>
<td>PARAMETERS:</td>
</tr>
<tr>
<td></td>
<td>ctx (IN) - the context handle obtained from the newContext call,</td>
</tr>
<tr>
<td></td>
<td>rowTag (IN) - the name of the ROW element. NULL indicates that you do not want the ROW element to be present. Call this function to set the name of the ROW element, if you do not want the default &quot;ROW&quot; name to show up. You can also set this to NULL to suppress the ROW element itself. Its an error if both the row and the rowset are null and there is more than one column or row in the output.</td>
</tr>
</tbody>
</table>
**dbms_xmlgen**

**Table 5–5  DBMS_XMLGEN Functions and Procedures (Cont.)**

<table>
<thead>
<tr>
<th>Function or Procedure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>setRowSetTag()</td>
<td>Sets the name of the document’s root element. The default name is “ROWSET”</td>
</tr>
</tbody>
</table>

**PROCEDURE**

```sql
setRowSetTag(ctx IN ctxHandle, rowSetTag IN VARCHAR2);
```

**PARAMETERS:**
- `ctx (IN)`: the context handle obtained from the `newContext` call,
- `rowsetTag (IN)`: the name of the document element.

NULL indicates that you do not want the `ROW` element to be present. Call this to set the name of the document root element, if you do not want the default “ROWSET” name in the output. You can also set this to NULL to suppress the printing of this element. However, this is an error if both the row and the rowset are null and there is more than one column or row in the output.

**getXML()**

**PROCEDURE**

```sql
getXML(ctx IN ctxHandle, clobval IN OUT NCOPY clob, dtdOrSchema IN number:= NONE);
```

**PARAMETERS:**
- `ctx (IN)`: The context handle obtained from the `newContext()` call,
- `clobval (IN/OUT)`: the clob to which the XML document is to be appended,
- `dtdOrSchema (IN)`: whether we should generate the DTD or Schema. This parameter is NOT supported.

Use this version of the `getXML` function, to avoid any extra CLOB copies and if you want to reuse the same CLOB for subsequent calls. This `getXML` call is more efficient than the next flavor, though this involves that you create the lob locator.

When generating the XML, the number of rows indicated by the `setSkipRows` call are skipped, then the maximum number of rows as specified by the `setMaxRows` call (or the entire result if not specified) is fetched and converted to XML. Use the `getNumRowsProcessed` function to check if any rows were retrieved or not.

**getXML()**

Generates the XML document and return it as a CLOB.
### Table 5–5  DBMS.XMLGEN Functions and Procedures (Cont.)

<table>
<thead>
<tr>
<th>Function or Procedure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FUNCTION getXML(ctx IN ctxHandle, dtdOrSchema IN number:= NONE) RETURN clob</strong></td>
<td>PARAMETERS: ctx (IN) - The context handle obtained from the newContext() call, [dtdOrSchema (IN) - whether we should generate the DTD or Schema. This parameter is NOT supported.] RETURNS: A temporary CLOB containing the document. Free the temporary CLOB obtained from this function using the <code>dbms_lob.freetemporary</code> call.</td>
</tr>
<tr>
<td><strong>FUNCTION getXMLType(ctx IN ctxHandle, dtdOrSchema IN number:= NONE) RETURN sys.XMLType</strong></td>
<td>PARAMETERS: ctx (IN) - The context handle obtained from the newContext() call, [dtdOrSchema (IN) - whether we should generate the DTD or Schema. This parameter is NOT supported.] RETURNS: An XMLType instance containing the document.</td>
</tr>
<tr>
<td><strong>FUNCTION getNumRowsProcessed()</strong></td>
<td>Gets the number of SQL rows processed when generating the XML using the getXML call. This count does not include the number of rows skipped before generating the XML.</td>
</tr>
<tr>
<td><strong>FUNCTION getNumRowsProcessed(ctx IN ctxHandle) RETURN number</strong></td>
<td>PARAMETERS: queryString (IN) - the query string, the result of which needs to be converted to XML. RETURNS: This gets the number of SQL rows that were processed in the last call to getXML. You can call this to find out if the end of the result set has been reached. This does not include the number of rows skipped. Use this function to determine the terminating condition if you are calling getXML in a loop. Note that getXML would always generate a XML document even if there are no rows present.</td>
</tr>
<tr>
<td><strong>PROCEDURE setMaxRows()</strong></td>
<td>Sets the maximum number of rows to fetch from the SQL query result for every invocation of the getXML call.</td>
</tr>
<tr>
<td><strong>PROCEDURE setMaxRows(ctx IN ctxHandle, maxRows IN NUMBER);</strong></td>
<td>PARAMETERS: ctx (IN) - the context handle corresponding to the query executed, [maxRows (IN) - the maximum number of rows to get per call to getXML.] The maxRows parameter can be used when generating paginated results using this utility. For instance when generating a page of XML or HTML data, you can restrict the number of rows converted to XML and then in subsequent calls, you can get the next set of rows and so on. This also can provide for faster response times.</td>
</tr>
</tbody>
</table>
### DBMS_XMLGEN

#### Table 5–5  DBMS_XMLGEN Functions and Procedures (Cont.)

<table>
<thead>
<tr>
<th>Function or Procedure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>setSkipRows()</td>
<td>Skips a given number of rows before generating the XML output for every call to the getXML routine.</td>
</tr>
</tbody>
</table>

**PROCEDURE**

**setSkipRows**

```sql
setSkipRows(ctx IN ctxHandle, skipRows IN NUMBER);
```

**PARAMETERS:**
- `ctx (IN)` - the context handle corresponding to the query executed,
- `skipRows (IN)` - the number of rows to skip per call to getXML.

The `skipRows` parameter can be used when generating paginated results for stateless web pages using this utility. For instance, when generating the first page of XML or HTML data, you can set `skipRows` to zero. For the next set, you can set the `skipRows` to the number of rows that you got in the first case.

| setConvertSpecialChars() | Sets whether special characters in the XML data need to be converted into their escaped XML equivalent or not. For example, the "<" sign is converted to &lt;. The default is to perform conversions. |

**PROCEDURE**

**setConvertSpecialChars**

```sql
setConvertSpecialChars(ctx IN ctxHandle, conv IN boolean);
```

**PARAMETERS:**
- `ctx (IN)` - the context handle to use,
- `conv (IN)` - true indicates that conversion is needed.

You can use this function to speed up the XML processing whenever you are sure that the input data cannot contain any special characters such as `<`, `>`, `"`, `'` etc. which need to be escaped. Note that it is expensive to actually scan the character data to replace the special characters, particularly if it involves a lot of data. So in cases when the data is XML-safe, then this function can be called to improve performance.

| useItemTagsForColl() | Sets the name of the collection elements. The default name for collection elements is the type name itself. You can override that to use the name of the column with the "_ITEM" tag appended to it using this function. |

**PROCEDURE**

**useItemTagsForColl**

```sql
useItemTagsForColl(ctx IN ctxHandle);
```

**PARAMETERS:**
- `ctx (IN)` - the context handle.

If you have a collection of NUMBER, say, the default tag name for the collection elements is NUMBER. You can override this behavior and generate the collection column name with the _ITEM tag appended to it, by calling this procedure.

| restartQuery() | Restarts the query and generate the XML from the first row again. |
DBMS_XMLGEN Example 1: Generating Simple XML

This example creates an XML document by selecting out the employee data from an object-relational table and puts 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
  closeContext(qryCtx);
END;
/
```

Here is the XML generated from this example:

```
select * from temp_clob_tab;
```
RESULT

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

DBMS_XMLGEN Example 2: Generating Simple XML with Pagination

Instead of getting the whole XML for all the rows, we can use the “fetch” interface that the DBMS_XMLGEN provides to retrieve a fixed number of rows each time. This speeds up the response time and also can help in scaling applications which would need to use a DOM API over the result XML - particularly if the number of rows is large.

The following example illustrates how to use DBMS_XMLGEN to retrieve results from the scott.emp table:

```sql
-- create a table to hold the results..
create table temp_clob_tab ( result clob);

declare
  qryCtx dbms_xmlgen.ctxHandle;
  result CLOB;
begin
```

-- get the query context;
qryCtx := dbms_xmlgen.newContext('select * from scott.emp');

-- set the maximum number of rows to be 5,
dbms_xmlgen.setMaxRows(qryCtx, 5);

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

  -- if there were no rows processed, then quit..!
  exit when dbms_xmlgen.getNumRowsProcessed(qryCtx) = 0;

  -- do some processing with the lob data..!
  -- Here, we are inserting the results
  -- into a table. You can print the lob out, output it to a stream,
  -- put it in a queue
  -- or do any other processing.
  insert into temp_clob_tab values(result);
end loop;
--close context
closeContext(qryCtx);
end;
/

Here, for each set of 5 rows, we would get an XML document.

**DBMS_XMLGEN Example 3: Generating Complex XML**

Complex XML can be generated using Object types to represent nested structures

```sql
CREATE TABLE new_departments (    
  department_id   NUMBER PRIMARY KEY,    
  department_name VARCHAR2(20) ) ;

CREATE TABLE new_employees (    
  employee_id    NUMBER PRIMARY KEY,    
  last_name   VARCHAR2(20),    
  department_id  NUMBER REFERENCES departments
)

CREATE TYPE emp_t AS OBJECT(    
  "@employee_id" NUMBER,
```
CREATE TYPE emplist_t AS TABLE OF emp_t;

CREATE TYPE dept_t AS OBJECT(
"@department_id" NUMBER,
  department_name VARCHAR2(20),
  emplist emplist_t
);

qryCtx := dbms_xmlgen.newContext
('SELECT dept_t(department_id, department_name,
  CAST(MULTISET
    (SELECT e.employee_id, e.last_name
     FROM new_employees e
     WHERE e.department_id = d.department_id)
    AS   emplist_t))  AS deptxml
FROM new_departments d');
DBMS_XMLGEN.setRowTag(qryCtx, NULL);
With relational data, the results are a flat non-nested XML document. To obtain nested XML structures, you can use object-relational data, where the mapping is as follows:

- **Object types** map as an XML element
- **Attributes of the type**, map to sub-elements of the parent element

---

**Note:** Complex structures can be obtained by using object types and creating object views or object tables. A canonical mapping is used to map object instances to XML.

The @ sign, when used in column or attribute names, is translated into an attribute of the enclosing XML element in the mapping.

---

**DBMS_XMLGEN Example 4: Generating Complex XML #2 - Inputting User Defined Types To Get Nesting in XML Documents**

When you input a user-defined type (UDT) value to DBMS_XMLGEN functions, the user-defined type gets mapped to an XML document using a canonical mapping. In the canonical mapping, user-defined type’s attributes are mapped to XML elements.

Any attributes with names starting with “@” are mapped to an attribute of the preceding element.

User-defined types can be used to get nesting within the result XML document.

For example, consider the two tables, EMP and DEPT:

```sql
CREATE TABLE DEPT
(
    deptno number primary key,
    dname varchar2(20)
);

CREATE TABLE EMP
(
    empno number primary key,
    ename varchar2(20),
    deptno number references dept
);
```

Now, to generate a hierarchical view of the data, that is, departments with employees in them, you can define suitable object types to create the structure inside the database as follows:
CREATE TYPE EMP_T AS OBJECT
(
    "@empno" number, -- empno defined as an attribute!
    ename varchar2(20),
); /
/
You have defined the empno with an @ sign in front, to denote that it must be
mapped as an attribute of the enclosing Employee element.

CREATE TYPE EMPLIST_T AS TABLE OF EMP_T;
/
CREATE TYPE DEPT_T AS OBJECT
(
    "@deptno" number,
    dname varchar2(20),
    emplist emplist_t
); /
/
Department type, DEPT_T, represents the department as containing a list of
employees. You can now query the employee and department tables and get the
result as an XML document, as follows:

    declare
    qryCtx dbms_xmlgen.ctxHandle;
    result CLOB;
    begin

    -- get the query context;
    qryCtx := dbms_xmlgen.newContext(
        'SELECT
            dept_t(deptno, dname,
                CAST(MULTISET(select empno, ename
                        from emp e
                        where e.deptno = d.deptno) AS emplist_t))
            AS deptxml
            FROM dept d');

    -- set the maximum number of rows to be 5,
    dbms_xmlgen.setMaxRows(qryCtx, 5);

    -- set no row tag for this result as we have a single ADT column
    dbms_xmlgen.setRowTag(qryCtx, null);

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

-- if there were no rows processed, then quit..!
exit when dbms_xmlgen.getNumRowsProcessed(qryCtx) = 0;

-- do whatever with the result..!
end loop;
end;
/

The MULTISET operator treats the result of the subset of employees working in the department as a list and the CAST around it, casts it to the appropriate collection type. You then create a department instance around it and call the DBMS_XMLGEN routines to create the XML for the object instance. The result is:

```xml
<?xml version="1.0"?>
<ROWSET>
  <DEPTXML deptno="10">
    <DNAME>Sports</DNAME>
    <EMPLIST>
    <EMP_T empno="200">
      <ENAME>John</ENAME>
    </EMP_T>
    <EMP_T empno="300">
      <ENAME>Jack</ENAME>
    </EMP_T>
    </EMPLIST>
  </DEPTXML>
  <DEPTXML deptno="20">
    <!-- .. other columns -->
  </DEPTXML>
</ROWSET>
```

The default name “ROW” is not present because you set that to NULL. The deptno and empno have become attributes of the enclosing element.

**DBMS_XMLGEN Example 5: Generating a Purchase Order From the Database in XML Format**

This example uses DBMS_XMLGEN.getXMLType() to generate PurchaseOrder in XML format from a relational database using object views.
-- Note: DBMS_XMLGEN Package maps UDT attributes names starting with '@' to xml attributes

-- Purchase Order Object View Model

-- PhoneList Varray object type
CREATE TYPE PhoneList_vartyp AS VARRAY(10) OF VARCHAR2(20)
/

-- Address object type
CREATE TYPE Address_typ AS OBJECT (Street VARCHAR2(200), City VARCHAR2(200), State CHAR(2), Zip VARCHAR2(20))
/

-- Customer object type
CREATE TYPE Customer_typ AS OBJECT (CustNo NUMBER, CustName VARCHAR2(200), Address Address_typ, PhoneList PhoneList_vartyp)
/

-- StockItem object type
CREATE TYPE StockItem_typ AS OBJECT (@StockNo NUMBER, Price NUMBER, TaxRate NUMBER)
/

-- LineItems object type
CREATE TYPE LineItem_typ AS OBJECT (@LineItemNo NUMBER, Item StockItem_typ, Quantity NUMBER, Discount NUMBER)
/

-- LineItems Nested table
CREATE TYPE LineItems_ntabtyp AS TABLE OF LineItem_typ
CREATE TYPE PO_typ AUTHID CURRENT_USER AS OBJECT (  
PONO NUMBER,  
Cust_ref REF Customer_typ,  
OrderDate DATE,  
ShipDate TIMESTAMP,  
LineItems_ntab LineItems_ntabtyp,  
ShipToAddr Address_typ  
);

CREATE TABLE Customer_tab(  
CustNo NUMBER NOT NULL,  
CustName VARCHAR2(200),  
Street VARCHAR2(200),  
City VARCHAR2(200),  
State CHAR(2),  
Zip VARCHAR2(20),  
Phone1 VARCHAR2(20),  
Phone2 VARCHAR2(20),  
Phone3 VARCHAR2(20),  
constraint cust_pk PRIMARY KEY (CustNo)  
);  
ORGANIZATION INDEX OVERFLOW;

CREATE TABLE po_tab (  
PONO NUMBER, /* purchase order no */  
CustNo NUMBER constraint po_cust_fk references Customer_tab, /* Foreign KEY referencing customer */  
OrderDate DATE, /* date of order */  
ShipDate TIMESTAMP, /* date to be shipped */  
ToStreet VARCHAR2(200), /* shipto address */  
ToCity VARCHAR2(200),  
ToState CHAR(2),  
ToZip VARCHAR2(20),  
constraint po_pk PRIMARY KEY(PONO)  
);

-- Stock Table
CREATE TABLE Stock_tab {
  StockNo    NUMBER constraint stock_uk UNIQUE,
  Price      NUMBER,
  TaxRate    NUMBER
};

-- Line Items Table
CREATE TABLE LineItems_tab(  
  LineItemNo  NUMBER,
  PONo        NUMBER constraint LI_PO_FK REFERENCES po_tab,
  StockNo     NUMBER ,
  Quantity    NUMBER,
  Discount    NUMBER,
  constraint LI_PK PRIMARY KEY (PONo, LineItemNo)
);

-- create Object Views

-- Customer Object View
CREATE OR REPLACE VIEW Customer OF Customer_typ  
WITH OBJECT IDENTIFIER(CustNo)  
AS SELECT c.Custno, C.custname,  
         Address_typ(C.Street, C.City, C.State, C.Zip),  
         PhoneList_vartyp(Phone1, Phone2, Phone3)  
FROM Customer_tab c;

-- Purchase order view
CREATE OR REPLACE VIEW PO OF PO_typ  
WITH OBJECT IDENTIFIER (PONO)  
AS SELECT P.PONo,  
         MAKE_REF(Customer, P.Custno),  
         P.OrderDate,  
         P.ShipDate,  
         CAST( MULTISET(  
               SELECT LineItem_typ( L.LineItemNo,  
                                   StockItem_typ(L.StockNo,S.Price,S.TaxRate),  
                                   L.Quantity, L.Discount)  
               FROM LineItems_tab L, Stock_tab S  
               WHERE L.PONo = P.PONo and S.StockNo=L.StockNo )  
         AS LineItems_ntabtyp),  
         Address_typ(P.ToStreet,P.ToCity, P.ToState, P.ToZip)  
FROM PO_tab P;

-- create table with XMLType column to store po in XML format
create table po_xml_tab(
poid number,
    poDoc SYS.XMLType /* purchase order in XML format */
) /

---------------------
-- Populate data
---------------------
-- Establish Inventory

INSERT INTO Stock_tab VALUES(1004, 6750.00, 2) ;
INSERT INTO Stock_tab VALUES(1011, 4500.23, 2) ;
INSERT INTO Stock_tab VALUES(1534, 2234.00, 2) ;
INSERT INTO Stock_tab VALUES(1535, 3456.23, 2) ;

-- Register Customers

INSERT INTO Customer_tab
    VALUES (1, 'Jean Nance', '2 Avocet Drive',
            'Redwood Shores', 'CA', '95054',
                    '415-555-1212', NULL, NULL) ;

INSERT INTO Customer_tab
    VALUES (2, 'John Nike', '323 College Drive',
            'Edison', 'NJ', '08820',
                    '609-555-1212', '201-555-1212', NULL) ;

-- Place Orders

INSERT INTO PO_tab
    VALUES (1001, 1, '10-APR-1997', '10-MAY-1997',
                NULL, NULL, NULL, NULL) ;

INSERT INTO PO_tab
                '55 Madison Ave', 'Madison', 'WI', '53715') ;

-- Detail Line Items

INSERT INTO LineItems_tab VALUES(01, 1001, 1534, 12, 0) ;
INSERT INTO LineItems_tab VALUES(02, 1001, 1535, 10, 10) ;
INSERT INTO LineItems_tab VALUES(01, 2001, 1004, 1, 0) ;
INSERT INTO LineItems_tab VALUES(02, 2001, 1011, 2, 1) ;
--- Use DBMS_XMLGEN Package to generate PO in XML format
--- and store SYS.XMLType in po_xml table
-------------------------------------------------------

declare
    qryCtx dbms_xmlgen.ctxHandle;
    pxml SYS.XMLType;
    cxml clob;
begin
    -- get the query context;
    qryCtx := dbms_xmlgen.newContext(''
        select pono,deref(cust_ref) customer,p.OrderDate,p.shipdate,
            lineitems_ntab lineitems,shiptoaddr
        from po p''
    );
    -- set the maximum number of rows to be 1,
    dbms_xmlgen.setMaxRows(qryCtx, 1);
    -- set rowset tag to null and row tag to PurchaseOrder
    dbms_xmlgen.setRowSetTag(qryCtx,null);
    dbms_xmlgen.setRowTag(qryCtx,'PurchaseOrder');
    loop
        -- now get the po in xml format
        pxml := dbms_xmlgen.getXMLType(qryCtx);
        -- if there were no rows processed, then quit..!
        exit when dbms_xmlgen.getNumRowsProcessed(qryCtx) = 0;
        -- Store SYS.XMLType po in po_xml table (get the pono out)
        insert into po_xml_tab (poid, poDoc)
            values(
                pxml.extract('//PONO/text()').getNumberVal(),
                pxml);
    end loop;
end;
/
-------------------------------------------------------

-- list xml PurchaseOrders
---------------------------

set long 100000

5-60 Oracle9i Application Developer's Guide - XML, Release 1 (9.0.1)
set pages 100
select x.xpo.getClobVal() xpo
from po_xml x;

PurchaseOrder 1001:

This produces the following purchase order XML document:

```xml
<?xml version="1.0"?>
<PurchaseOrder>
  <PONO>1001</PONO>
  <CUSTOMER>
    <CUSTNO>1</CUSTNO>
    <CUSTNAME>Jean Nance</CUSTNAME>
    <ADDRESS>
      <STREET>2 Avocet Drive</STREET>
      <CITY>Redwood Shores</CITY>
      <STATE>CA</STATE>
      <ZIP>95054</ZIP>
    </ADDRESS>
    <PHONELIST>
      <VARCHAR2>415-555-1212</VARCHAR2>
    </PHONELIST>
  </CUSTOMER>
  <ORDERDATE>10-APR-97</ORDERDATE>
  <SHIPDATE>10-MAY-97 12.00.00.000000 AM</SHIPDATE>
  <LINEITEMS>
    <LINEITEM_TYP LineItemNo="1">
      <ITEM StockNo="1534">
        <PRICE>2234</PRICE>
        <TAXRATE>2</TAXRATE>
      </ITEM>
      <QUANTITY>12</QUANTITY>
      <DISCOUNT>0</DISCOUNT>
    </LINEITEM_TYP>
    <LINEITEM_TYP LineItemNo="2">
      <ITEM StockNo="1535">
        <PRICE>3456.23</PRICE>
        <TAXRATE>2</TAXRATE>
      </ITEM>
      <QUANTITY>10</QUANTITY>
      <DISCOUNT>10</DISCOUNT>
    </LINEITEM_TYP>
  </LINEITEMS>
</PurchaseOrder>
```
<?xml version="1.0"?>
<PurchaseOrder>
  <PONO>2001</PONO>
  <CUSTOMER>
    <CUSTNO>2</CUSTNO>
    <CUSTNAME>John Nike</CUSTNAME>
    <ADDRESS>
      <STREET>323 College Drive</STREET>
      <CITY>Edison</CITY>
      <STATE>NJ</STATE>
      <ZIP>08820</ZIP>
    </ADDRESS>
    <PHONELIST>
      <VARCHAR2>609-555-1212</VARCHAR2>
      <VARCHAR2>201-555-1212</VARCHAR2>
    </PHONELIST>
  </CUSTOMER>
  <ORDERDATE>20-APR-97</ORDERDATE>
  <SHIPDATE>20-MAY-97 12.00.00.000000 AM</SHIPDATE>
  <LINEITEMS>
    <LINEITEM_TYP LineItemNo="1">
      <ITEM StockNo="1004">
        <PRICE>6750</PRICE>
        <TAXRATE>2</TAXRATE>
      </ITEM>
      <QUANTITY>1</QUANTITY>
      <DISCOUNT>0</DISCOUNT>
    </LINEITEM_TYP>
    <LINEITEM_TYP LineItemNo="2">
      <ITEM StockNo="1011">
        <PRICE>4500.23</PRICE>
        <TAXRATE>2</TAXRATE>
      </ITEM>
      <QUANTITY>2</QUANTITY>
      <DISCOUNT>1</DISCOUNT>
    </LINEITEM_TYP>
  </LINEITEMS>
  <SHIPTOADDR>
    <STREET>55 Madison Ave</STREET>
    <CITY>Madison</CITY>
    <STATE>WI</STATE>
  </SHIPTOADDR>
</PurchaseOrder>
Oracle9i introduces a new SQL function, `SYS_XMLGEN()`, that generates XML in SQL queries. `DBMS_XMLGEN` and other packages operate at the query level, giving aggregated results for the entire query. `SYS_XMLGEN` takes in a single argument in an SQL query and converts it (the result) to XML.

`SYS_XMLGEN` takes a scalar value, object type, or `XMLType` instance to be converted to an XML document. It also takes an optional `XMLGenFormatType` object that you can use to specify formatting options for the resulting XML document. `SYS_XMLGEN` returns an `XMLType`.

It is used to create and query XML instances within SQL queries, as follows:

```sql
SQL> SELECT SYS_XMLGEN(employee_id)
    2   FROM employees WHERE last_name LIKE 'Scott%';
```

The resulting XML document is:

```xml
<employee_id>60</employee_id>
```

**SYS_XMLGEN Syntax**

The `SYS_XMLGEN` function takes an expression that evaluates to a particular row and column of the database, and returns an instance of type `XMLType` containing an XML document. See Figure 5–3. The `expr` can be a scalar value, a user-defined type, or a `XMLType` instance.

- If `expr` is a scalar value, the function returns an XML element containing the scalar value.
- If `expr` is a type, the function maps the user-defined type attributes to XML elements.
- If `expr` is a `XMLType` instance, then the function encloses the document in an XML element whose default tag name is `ROW`.

By default the elements of the XML document match the elements of `expr`. For example, if `expr` resolves to a column name, the enclosing XML element will be the
same column name. If you want to format the XML document differently, specify `fmt`, which is an instance of the XMLGenFormatType object.

**Figure 5-3  SYS_XMLGEN Syntax**

The following example retrieves the employee email ID from the sample table `oe.employees` where the `employee_id` value is 205, and generates an instance of a XMLType containing an XML document with an EMAIL element.

```
SELECT SYS_XMLGEN(email).getStringVal()
FROM employees
WHERE employee_id = 205;
```

Why is SYS_XMLGEN so Powerful?

SYS_XMLGEN() is powerful for the following reasons:

- You can create and query XML instances within SQL queries.
- Using the object-relational infrastructure, you can create complex and nested XML instances from simple relational tables.

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 XMLGenFormatType object type through which you can customize the SQL results. A NULL format object implies that the default mapping behavior is to be used.
**XMLGenFormatType Object**

XMLGenFormatType can be used to specify formatting arguments to `SYS_XMLGEN` and `SYS_XMLAGG` functions. Table 5–6 lists the XMLGenFormatType attributes.

<table>
<thead>
<tr>
<th>XMLGenFormatType Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>enclTag</td>
<td>The name of the enclosing tag to use</td>
</tr>
<tr>
<td>schemaType</td>
<td>Not currently supported, but in future will be used to specify how the XMLSchema should be generated.</td>
</tr>
<tr>
<td>processingIns</td>
<td>Can specify processing instructions such as stylesheet instructions that need to be appended to the beginning of the document.</td>
</tr>
<tr>
<td>schemaName</td>
<td>Will be used for XMLSchema.</td>
</tr>
<tr>
<td>dburl</td>
<td>Not used currently.</td>
</tr>
<tr>
<td>targetNameSpace</td>
<td>Not used currently, but will be used for XMLSchema generation.</td>
</tr>
</tbody>
</table>
Creating a Formatting Object with createFormat

You can use the static member function `createFormat` to create a formatting object. This function has most of the values defaulted. For example:

```sql
-- function to create the formatting object..
STATIC MEMBER FUNCTION createFormat(
    enc1Tag IN varchar2 := null,
    schemaType IN varchar2 := 'NO_SCHEMA'
    schemaName IN varchar2 := null,
    targetNameSpace IN varchar2 := null,
    dburl IN varchar2 := null,
    processingIns IN varchar2 := null)
RETURN XMLGenFormatType;
```

SYS_XMLGEN Example 1: Converting a Scalar Value to an XML Document Element's Contents

When you input a scalar value to `SYS_XMLGEN()`, `SYS_XMLGEN()` converts the scalar value to an `element` containing the scalar value. For example:

```sql
select sys_xmlgen(empno) from scott.emp where rownum < 1;
```

returns an XML document that contains the empno value as an element, as follows:

```xml
<?xml version="1.0"?>
<EMPNO>30</EMPNO>
```

The enclosing element name, in this case EMPNO, is derived from the column name passed to the operator. Also, note that the result of the SELECT statement is a row containing a XMLType.

---

**Note:** Currently, SQL*Plus cannot display XMLType properly, so you need to extract the LOB value out and display that. Use the `getClobval()` function on the XMLType to retrieve the CLOB value. For example,

```sql
SELECT sys_xmlgen(empno).getclobval() FROM scott.emp WHERE rownum < 1;
```

---

In the last example, you used the column name “EMPNO” for the document. If the column name cannot be derived directly, then the default name “ROW” is used. For example, in the following case:
select sys_xmlgen(empno).getClobVal()
from scott.emp
where rownum < 1;

you get the following XML output:

```xml
<?xml version="1.0"?>
<ROW>60</ROW>
```
since the function cannot infer the name of the expression. You can override the
default ROW tag by supplying an XMLGenFormatType object to the first argument
of the operator.

For example, in the last case, if you wanted the result to have EMPNO as the tag
name, you can supply a formatting argument to the function, as follows:

```sql
select sys_xmlgen(empno *2,
    sys.xmlgenformattype.createformat('EMPNO')).getClobVal()
from dual;
```

This results in the following XML:

```xml
<?xml version="1.0"?>
<EMPNO>60</EMPNO>
```

---

**Note:** Currently, CURSOR expressions are not supported as input values.

---

**SYS_XMLGEN Example 2: Converting a User-Defined Type (UDT) to XML**

When you input a user-defined type (UDT) value to SYS_XMLGEN(), the
user-defined type gets mapped to an XML document using a canonical mapping. In
the canonical mapping the user-defined type’s attributes are mapped to XML
elements.

Any type attributes with names starting with “@” are mapped to an attribute of the
preceding element. User-defined types can be used to get nesting within the result
XML document.

Using the same example as given in the DBMS_XMLGEN section ("DBMS_XMLGEN
Example 4: Generating Complex XML #2 - Inputting User Defined Types To Get
Nesting in XML Documents” on page 5-53), you can generate a hierarchical XML
for the employee, department example as follows:
SELECT SYS_XMLGEN(  
  dept_t(deptno, dname,  
      CAST(MULTISET(  
        select empno, ename  
        from emp e  
        where e.deptno = d.deptno) AS emplist_t)))).getClobVal()  
  AS deptxml  
FROM dept d;

The MULTISET operator treats the result of the subset of employees working in the department as a list and the CAST around it, cast’s it to the appropriate collection type. You then create a department instance around it and call SYS_XMLGEN() to create the XML for the object instance.

The result is:

```xml
<?xml version="1.0"?>
<ROW DEPTNO="100">
  <DNAME>Sports</DNAME>
  <EMPLIST>
    <EMP_T EMPNO="200">
      <ENAME>John</ENAME>
    </EMP_T>
    <EMP_T>
      <ENAME>Jack</ENAME>
    </EMP_T>
  </EMPLIST>
</ROW>
```

per row of the department. The default name “ROW” is present because the function cannot deduce the name of the input operand directly.

The difference between the SYS_XMLGEN and the DBMS_XMLGEN is apparent from this example:

- SYS_XMLGEN works inside SQL queries and operates on the expressions and columns within the row
- DBMS_XMLGEN works on the entire result set

**SYS_XMLGEN Example 3: Converting XMLType Instance**

If you pass an XML document into SYS_XMLGEN(), SYS_XMLGEN encloses the document (or fragment) with an element, whose tag name is the default “ROW”, or the name passed in through the formatting object. This functionality can be used to turn document fragments into well formed documents.
For example, the extract operation on the following document, can return a fragment. If you extract out the EMPNO elements from the following document:

```xml
<DOCUMENT>
  <EMPLOYEE>
    <ENAME>John</ENAME>
    <EMPNO>200</EMPNO>
  </EMPLOYEE>
  <EMPLOYEE>
    <ENAME>Jack</ENAME>
    <EMPNO>400</EMPNO>
  </EMPLOYEE>
  <EMPLOYEE>
    <ENAME>Joseph</ENAME>
    <EMPNO>300</EMPNO>
  </EMPLOYEE>
</DOCUMENT>
```

Using the following statement:

```sql
select e.xmldoc.extract('/DOCUMENT/EMPLOYEE/ENAME')
from po_xml_tab e;
```

you get a document fragment such as the following:

```xml
<ENAME>John</ENAME>
<ENAME>Jack</ENAME>
<ENAME>Joseph</ENAME>
```

You can make this fragment a valid XML document, by calling `SYS_XMLGEN()` to put an enclosing element around the document, as follows:

```sql
select SYS_XMLGEN(e.xmldoc.extract('/DOCUMENT/EMPLOYEE/ENAME')).getclobval()
from po_xml_tab e;
```

This places an element “ROW” around the result, as follows:

```xml
<?xml version="1.0"?>
<ROW>
  <ENAME>John</ENAME>
  <ENAME>Jack</ENAME>
  <ENAME>Joseph</ENAME>
</ROW>
```
SYS_XMLGEN

--- Example 4: Using SYS_XMLGEN() with Object Views

```sql
-- create Purchase Order object type
CREATE OR REPLACE TYPE PO_typ AUTHID CURRENT_USER AS OBJECT (
    PONO NUMBER,
    Customer Customer_typ,
    OrderDate DATE,
    ShipDate TIMESTAMP,
    LineItems_ntab LineItems_ntabtyp,
    ShipToAddr Address_typ
);

-- Purchase order view
CREATE OR REPLACE VIEW PO OF PO_typ
WITH OBJECT IDENTIFIER (PONO)
AS SELECT P.PONo,
    Customer_typ(P.Custno,C.CustName,C.Address,C.PhoneList),
    P.OrderDate,
    P.ShipDate,
    CAST( MULTISET(
        SELECT LineItem_typ( L.LineItemNo,
            StockItem_typ(L.StockNo,S.Price,S.TaxRate),
            L.Quantity, L.Discount)
        FROM LineItems_tab L, Stock_tab S
        WHERE L.PONo = P.PONo and S.StockNo=L.StockNo )
    AS LineItems_ntabtyp),
    Address_typ(P.ToStreet,P.ToCity, P.ToState, P.ToZip)
FROM PO_tab P, Customer C
WHERE P.CustNo=C.custNo;

--- Use SYS_XMLGEN() to generate PO in XML format
set long 20000
set pages 100
SELECT SYS_XMLGEN(value(p),
    sys.xmlgenformatType.createFormat('PurchaseOrder')).getClobVal()
```

**Note:** If the input was a column, then the column name would have been used as default. You can override the enclosing element name using the formatting object that can be passed in as an additional argument to the function.

---
This returns the Purchase Order in XML format:

```xml
<?xml version="1.0"?>
<PurchaseOrder>
  <PONO>1001</PONO>
  <CUSTOMER>
    <CUSTNO>1</CUSTNO>
    <CUSTNAME>Jean Nance</CUSTNAME>
    <ADDRESS>
      <STREET>2 Avocet Drive</STREET>
      <CITY>Redwood Shores</CITY>
      <STATE>CA</STATE>
      <ZIP>95054</ZIP>
    </ADDRESS>
    <PHONELIST>
      <VARCHAR2>415-555-1212</VARCHAR2>
    </PHONELIST>
  </CUSTOMER>
  <ORDERDATE>10-APR-97</ORDERDATE>
  <SHIPDATE>10-MAY-97 12.00.00.000000 AM</SHIPDATE>
  <LINEITEMS_NTAB>
    <LINEITEM_TYP LineItemNo="1">
      <ITEM StockNo="1534">
        <PRICE>2234</PRICE>
        <TAXRATE>2</TAXRATE>
      </ITEM>
      <QUANTITY>12</QUANTITY>
      <DISCOUNT>0</DISCOUNT>
    </LINEITEM_TYP>
    <LINEITEM_TYP LineItemNo="2">
      <ITEM StockNo="1535">
        <PRICE>3456.23</PRICE>
        <TAXRATE>2</TAXRATE>
      </ITEM>
      <QUANTITY>10</QUANTITY>
      <DISCOUNT>10</DISCOUNT>
    </LINEITEM_TYP>
  </LINEITEMS_NTAB>
</PurchaseOrder>
```
SYS_XMLAGG

SYS_XMLAGG aggregates all input documents and produces a single XML document. It aggregates (concatenates) fragments. SYS_XMLAGG takes in an XMLType as argument, and aggregates (or concatenates) all XML documents in rows into a single document per group. Use SYS_XMLAGG() for either of the following tasks:

- Aggregate fragments together
- Aggregate related XML data

In Figure 5-4 shows how 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, specify fmt, which is an instance of the SYS.XMLGenFormatType object.

**Figure 5-4  SYS_XMLAGG Syntax**

For example:

```sql
SQL> SELECT SYS_XMLAGG(SYS_XMLGEN(last_name) ,
               SYS.XMLGENFORMATTYPE.createFormat
               ('EmployeeGroup')).getClobVal()
FROM employees
GROUP BY department_id;
```

This generates the following XML document:

```xml
<EmployeeGroup>
  <last_name>Scott</last_name>
  <last_name>Mary</last_name>
</EmployeeGroup >
<EmployeeGroup >
  <last_name>Jack</last_name>
  <last_name>John</last_name>
</EmployeeGroup >
```
SYS_XMLAGG Example 1: Aggregating XML from Relational Data

Consider the SELECT statement:

```sql
SELECT SYS_XMLAGG(SYS_XMLGEN(ename)).getClobVal() xml_val
FROM scott.emp
GROUP BY deptno;
```

This returns the following XML document:

```
xm_val
-------------
<ROWSET>
  <ENAME>John</ENAME>
  <ENAME>Jack</ENAME>
  <ENAME>Joseph</ENAME>
</ROWSET>

<ROWSET>
  <ENAME>Scott</ENAME>
  <ENAME>Adams</ENAME>
</ROWSET>

2 rows selected
```

Here, you grouped all the employees belonging to a particular department and aggregated all the XML documents produced by the `SYS_XMLGEN` function. In the previous SQL statement, if you wanted to enclose the result of each group in an `EMPLOYEE` tag, use the following statement:

```sql
select sys_xmlagg(sys_xmlgen(ename),
  sys.xmlgenformattype.createformat('EMPLOYEE')).getClobVal() xmlval
FROM scott.emp
GROUP BY deptno);
```

This returns the following:

```
xm_val
-------------
<EMPLOYEE>
  <ENAME>John</ENAME>
  <ENAME>Jack</ENAME>
  <ENAME>Joseph</ENAME>
</EMPLOYEE>

<EMPLOYEE>
  <ENAME>Scott</ENAME>
```

Database Support for XML  5-73
SYS_XMLAGG Example 2: Aggregating XMLType Instances Stored in Tables

You can also aggregate XMLType instances that are stored in tables or selected out from functions. Assuming that you have a table defined as follows:

```sql
CREATE TABLE po_tab
(
  pono number primary key,
  orderdate date,
  poxml sys.XMLType;
);
```

```sql
insert into po_Tab values (100,'10-11-2000',
  '<?xml version="1.0"?><PO pono="100"><PONAME>Po_1</PONAME></PO>');
insert into po_Tab values (200,'10-23-1999',
  '<?xml version="1.0"?><PO pono="200"><PONAME>Po_2</PONAME></PO>');
```

You can now aggregate the purchase orders into a single purchase order using the SYS_XMLAGG function as follows:

```sql
select SYS_XMLAGG(poxml,sys.xmlgenformattype.createformat('POSET'))
from po_Tab;
```

This produces a single XML document of the form:

```xml
<?xml version="1.0"?>
<POSET>
  <PO pono="100">
    <PONAME>Po_1</PONAME>
  </PO>
  <PO pono="200">
    <PONAME>Po_2</PONAME>
  </PO>
</POSET>
```

SYS_XMLAGG Example 3: Aggregating XMLType Fragments

The Extract() function in XMLType, allows you to extract fragments of XML documents. You can also aggregate these fragments together using the SYS_XMLAGG function. For example, from the previous example, if you extract out the
PONAME elements alone, you can aggregate those together using the `SYS_XMLAGG` function as follows:

```sql
select SYS_XMLAGG(p.po_xml.extract('//PONAME')).getclobval()
from po_tab p;
```

This produces the following XML document:

```xml
<?xml version="1.0"?>
<ROWSET>
    <PONAME>Po_1</PONAME>
    <PONAME>Po_2</PONAME>
</ROWSET>
```

**SYS_XMLAGG Example 4: Aggregating all Purchase Orders into One XML Document**

```sql
set long 20000
set pages 200
SELECT SYS_XMLAGG(SYS_XMLGEN(value(p),
        sys.xmlgenformatType.createFormat('PurchaseOrder'))).getClobVal()
PO
FROM po p;
```

This returns all Purchase Orders in one XML Document, namely that enclosed in the ROWSET element:

```xml
<?xml version="1.0"?>
<ROWSET>
    <PurchaseOrder>
        <PONO>1001</PONO>
        <CUSTOMER>
            <CUSTNO>1</CUSTNO>
            <CUSTNAME>Jean Nance</CUSTNAME>
            <ADDRESS>
                <STREET>2 Avocet Drive</STREET>
                <CITY>Redwood Shores</CITY>
                <STATE>CA</STATE>
                <ZIP>95054</ZIP>
            </ADDRESS>
            <PHONELIST>
                <VARCHAR2>415-555-1212</VARCHAR2>
            </PHONELIST>
        </CUSTOMER>
        <ORDERDATE>10-APR-97</ORDERDATE>
    </PurchaseOrder>
</ROWSET>
```
<SHIPDATE>10-MAY-97 12.00.00.000000 AM</SHIPDATE>
<LINEITEMS_NTAB>
  <LINEITEM_TYP LineItemNo="1">
    <ITEM StockNo="1534">
      <PRICE>2234</PRICE>
      <TAXRATE>2</TAXRATE>
    </ITEM>
    <QUANTITY>12</QUANTITY>
    <DISCOUNT>0</DISCOUNT>
  </LINEITEM_TYP>
  <LINEITEM_TYP LineItemNo="2">
    <ITEM StockNo="1535">
      <PRICE>3456.23</PRICE>
      <TAXRATE>2</TAXRATE>
    </ITEM>
    <QUANTITY>10</QUANTITY>
    <DISCOUNT>10</DISCOUNT>
  </LINEITEM_TYP>
</LINEITEMS_NTAB>
<SHIPTOADDR/>
</PurchaseOrder>
<PurchaseOrder>
  <PONO>2001</PONO>
  <CUSTOMER>
    <CUSTNO>2</CUSTNO>
    <CUSTNAME>John Nike</CUSTNAME>
    <ADDRESS>
      <STREET>323 College Drive</STREET>
      <CITY>Edison</CITY>
      <STATE>NJ</STATE>
      <ZIP>08820</ZIP>
    </ADDRESS>
    <PHONELIST>
      <VARCHAR2>609-555-1212</VARCHAR2>
      <VARCHAR2>201-555-1212</VARCHAR2>
    </PHONELIST>
  </CUSTOMER>
  <ORDERDATE>20-APR-97</ORDERDATE>
  <SHIPDATE>20-MAY-97 12.00.00.000000 AM</SHIPDATE>
<LINEITEMS_NTAB>
  <LINEITEM_TYP LineItemNo="1">
    <ITEM StockNo="1004">
      <PRICE>6750</PRICE>
      <TAXRATE>2</TAXRATE>
    </ITEM>
    <QUANTITY>15</QUANTITY>
    <DISCOUNT>0</DISCOUNT>
  </LINEITEM_TYP>
</LINEITEMS_NTAB>
</PurchaseOrder>
Other Aggregation Methods

**ROLLUP and CUBE**

Oracle provides powerful functionality for OLAP operations such as CUBE and ROLLUP. SYS_XMLAGG function also works in these cases. You can, for example, create different XML documents based on the ROLLUP or CUBE operation.

**WINDOWING Function**

Oracle provides windowing functions that can be used to compute cumulative, moving, and centered aggregates. SYS_XMLAGG can also be used here to create documents based on rank and partition.

**TABLE Functions**

The previous sections talked about the new functions and operators that Oracle has introduced that can help query the XML instances using a XPath-like syntax. However, you also need to be able to explode the XML into simple relational or object-relational data so that you can insert that into tables or query using standard relational SQL statements. You can do this using a powerful mechanism called TABLE functions.
Table functions are new in Oracle9i. They can be used to model any arbitrary data (internal to the database or from an external source) as a collection of SQL rows. Table functions are executed pipelined and in parallel for improved performance. You can use Table functions to break XML into SQL rows. These can then be consumed by regular SQL queries and inserted into regular relational or object-relational tables.

With Oracle8i you could have a function returning a collection and use it in the FROM clause in the SELECT statement. However, the function would have to materialize the entire collection before it can be consumed by the outer query block.

With TABLE functions, these are both pipelined and parallel. Thus the function need not instantiate the whole collection in memory and instead pipe the results to the outer query block. The result is a faster response time and lower memory usage.

**See Also:** Oracle9i Application Developer's Guide - Fundamentals, Table Functions.

### Using Table Functions with XML

With XML, you can use these TABLE functions, to break the XML into SQL rows that can be queried by regular SQL queries and put into regular relational or object relational tables. Since they are parallel and piped, the performance of such an operation is vastly improved.

You can define a function for instance that takes an XMLType or a CLOB and returns a well known collection type. The function, for example, can use the XML parser available with Oracle9i to perform SAX parsing and return the results, or use the `extract()` function to extract pieces of the XML document and return it.

### Table Functions Example 1: Exploding the PO to Store in a Relational Table

Assuming that you have the purchase order document explained in earlier sections and you need to explode it to store it in relational table containing the purchase order details, you first create a type to describe the structure of the result,

```sql
create type poRow_type as object
(
    poName varchar2(20),
    poStreet varchar2(20),
    poCity varchar2(20),
    poState char(2),
    poZip char(10)
);
```


create type poRow_list as TABLE of poRow_type;
/

Now, you can either create an ODCI implementation type to implement the TABLE interface, or use native PL/SQL.

**See Also:** *Oracle9i Application Developer’s Guide - Fundamentals*, Table Functions, for details on what the interface definitions are and an example of the body,...

Assuming that you have created the body of the implementation type in PL/SQL, by creating the function itself, you can define the TABLE function as follows:

```sql
create function poExplode_func (arg IN sys.XMLType) return poRow_list pipelined is
    out_rec poRow_type;
    poxml sys.XMLType;
    i binary_integer := 1;
    argnew sys.XMLType := arg;
begin

    loop

        -- extract the i’th purchase order!
        poxml := argnew.extract('//PO['||i||']');
        exit when poxml is null;

        -- extract the required attributes..!!
        out_rec.poname := poxml.extract('/PONAME/text()').getStringVal();
        out_rec.postreet := poxml.extract('/POADDR/STREET/text()').getStringVal();
        out_rec.pocity := poxml.extract('/POADDR/CITY/text()').getStringVal();
        out_rec.postate := poxml.extract('/POADDR/STATE/text()').getStringVal();
        out_rec.pozip := poxml.extract('/POADDR/ZIP/text()').getStringVal();
        PIPE ROW(out_rec);

        i := i+1;
    end loop;

    return;
end;
/
```

You can use this function in the FROM list, and the interfaces defined in the Imp_t would be automatically called to get the values in a pipelined fashion as follows:
select *  
    from TABLE( CAST(
            poExplode_func(
                sys.XMLType.createXML(
                    '<?xml version="1.0"?>
                    <POLIST>
                    <PO>
                        <PONAME>Po_1</PONAME>
                        <POADDR>
                            <STREET>100 Main Street</STREET>
                            <CITY>Sunnyvale</CITY>
                            <STATE>CA</STATE>
                            <ZIP>94086</ZIP>
                        </POADDR>
                    </PO>
                    <PO>
                        <PONAME>Po_2</PONAME>
                        <POADDR>
                            <STREET>200 First Street</STREET>
                            <CITY>Oaksdale</CITY>
                            <STATE>CA</STATE>
                            <ZIP>95043</ZIP>
                        </POADDR>
                    </PO>
                    </POLIST>
            ) AS poRow_list));

Note: In the foregoing example, XMLType static constructor was used to construct the XML document. You can also use bind variables or select list subqueries to get the value.

The SQL statement returns the following values:

<table>
<thead>
<tr>
<th>PONAME</th>
<th>POSTREET</th>
<th>PCITY</th>
<th>POSTATE</th>
<th>POZIP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Po_1</td>
<td>100 Main Street</td>
<td>Sunnyvale</td>
<td>CA</td>
<td>94086</td>
</tr>
<tr>
<td>Po_2</td>
<td>200 First Street</td>
<td>Oaksdale</td>
<td>CA</td>
<td>95043</td>
</tr>
</tbody>
</table>

which can then be inserted into relational tables or queried with SQL.
Frequently Asked Questions (FAQs): XMLType

Is Replication and Materialized Views (MV) Supported by XMLType?

Question Are there any issues regarding using XMLType with replication and materialized views?

Answer Replication treats XMLType as another user-defined type. It should work as other user-defined types. dbms_defer_query and user-defined conflict resolution routines do not support XMLType until a forthcoming release. So, no, replication and MV are not fully supported by XMLType in this release.

How Can I Update an XML Tag in a Database Record?

Question Can I update an XML tag within a record of a database table? Also, can I create indexes based on XML tags?

Answer In this release Oracle offers XMLType with a CLOB storage. Updatability is at the level of a single document, so you must replace the whole document. XMLType can be indexed using Oracle Text (interMedia). This kind of index is best for locating documents that match certain XML search criteria, but if your applications are going to want to select out data like the “quantity” and the “price” as numerical values to operate on (for example, by some graphing or data warehousing software), then storing data-oriented XML as real tables and columns will give the best fit. You can also use functional indexes to speed up certain well-known XPath expressions.

See Also:
- Chapter 8, “Searching XML Data with Oracle Text”
- Oracle Text Application Developer’s Guide
- Oracle Text Reference

The book “Building Oracle XML Applications”, by Steve Muench (O’Reilly), covers Oracle8i with examples.
Does XMLType Support the Enforcing of Business Rules Such as Attribute Constraints?

**Question** Does the XML datatype in Oracle9i, provide the ability to enforce business rules or constraints as an attribute of an element within the data? For example:

```xml
<Address>
  <Street type=string> </Street>
  <City type=string> </City>
  <State type=string size=2> </State>
  <Zip type=number size=5> </Zip>
</Address>
```

**Answer** With Oracle9i, you can use the trigger mechanism to enforce any constraints. Oracle9i currently only supports storage as a CLOB and no constraint checks are inherently available. With forthcoming releases we may offer XML schema compliance and you would be able to create columns conforming to schemas, and so on. This would enable you to have any constraint that the schema enforces.

How Do I Create XML Documents with the Appropriate Encoding for Japanese?

**Question** I need to use `SYS_XMLGEN()` to create documents in Japanese, on Oracle9i. The database character set is EUC_JP, but creating XML with `SYS_XMLGEN()`, `SYS_XMLAGG()` and `DBMS_XMLGEN()` produces documents with the XML declaration:

```xml
<?xml version="1.0"?>
```

Can I create XML documents with the appropriate declaration, such as:

```xml
<?xml version="1.0" encoding="EUC_JP"?>
```

**Answer** In Oracle9i, you cannot generate XML output with an encoding tag. If you have data in binary datatype, then the conversion will fail if it is not convertible to the database character set.

Note that the encoding declaration does not constitute a part of the document per-se. It is an identifier to help processors identify its encoding.
What is the Best Way to Store XML in a Database?

**Question** What is the best way to store XML in a database? Can I store XML data in an NCLOB? Or is it preferable to store it in a BLOB and manipulate it within the application only?

**Answer** The best way to store XML in Oracle9i is to use the **XMLType**. Oracle currently allows only CLOB storage natively, but in forthcoming releases Oracle may allow native storage in NCLOBs, BLOBs, etc.

For Oracle9i Release 1 (9.0.1), you can store the XML as:

- **XMLType** - the preferred way. This helps the server to know that the data is XML. You can do XML based querying and indexing.
- **BLOBs** - This helps you store the XML document intact in the same encoding as the original document. The burden is on the user to deal with all the encoding issues.
- **NCLOBs** - Storing XML as NCLOBs will allow you to SQL operations on the data, as well as do Context searches. Context search is supported only if the database charset is a proper super-set or convertible set of the NCHAR, that is, the NCLOB must be convertible to the database char set without loss of data.