**4  
Using XMLType**

This chapter describes how to use the XMLType datatype, create and manipulate XMLType tables and columns, and query on them. It contains the following sections:

* [What Is XMLType?](http://docs.oracle.com/cd/B10500_01/appdev.920/a96620/xdb04cre.htm" \l "1035001)

* [When to Use XMLType](http://docs.oracle.com/cd/B10500_01/appdev.920/a96620/xdb04cre.htm" \l "1035574)

* [Storing XMLType Data in Oracle XML DB](http://docs.oracle.com/cd/B10500_01/appdev.920/a96620/xdb04cre.htm" \l "1035571)

* [XMLType Member Functions](http://docs.oracle.com/cd/B10500_01/appdev.920/a96620/xdb04cre.htm" \l "1030046)

* [How to Use the XMLType API](http://docs.oracle.com/cd/B10500_01/appdev.920/a96620/xdb04cre.htm" \l "1030582)

* [Guidelines for Using XMLType Tables and Columns](http://docs.oracle.com/cd/B10500_01/appdev.920/a96620/xdb04cre.htm" \l "1028927)

* [Manipulating XML Data in XMLType Columns/Tables](http://docs.oracle.com/cd/B10500_01/appdev.920/a96620/xdb04cre.htm" \l "1029855)

* [Inserting XML Data into XMLType Columns/Tables](http://docs.oracle.com/cd/B10500_01/appdev.920/a96620/xdb04cre.htm" \l "1032735)

* [Selecting and Querying XML Data](http://docs.oracle.com/cd/B10500_01/appdev.920/a96620/xdb04cre.htm" \l "1024805)

* [Updating XML Instances and Data in Tables and Columns](http://docs.oracle.com/cd/B10500_01/appdev.920/a96620/xdb04cre.htm" \l "1032612)

* [Deleting XML Data](http://docs.oracle.com/cd/B10500_01/appdev.920/a96620/xdb04cre.htm" \l "1032654)

* [Using XMLType In Triggers](http://docs.oracle.com/cd/B10500_01/appdev.920/a96620/xdb04cre.htm" \l "1031488)

* [Indexing XMLType Columns](http://docs.oracle.com/cd/B10500_01/appdev.920/a96620/xdb04cre.htm" \l "1031630)

|  |
| --- |
| **Note:**   * + ***Non-schema-based:*** XMLType tables and columns described in this chapter are not based on XML schema. You can, however, use the techniques and examples provided in this chapter regardless of which storage option you choose for your XMLType tables and columns. See [Chapter 3, "Using Oracle XML DB"](http://docs.oracle.com/cd/B10500_01/appdev.920/a96620/xdb03usg.htm#1656) for further storage recommendations.   + ***XML schema-based:*** [Appendix B, "XML Schema Primer"](http://docs.oracle.com/cd/B10500_01/appdev.920/a96620/appbsch.htm#621642) and [Chapter 5, "Structured Mapping of XMLType"](http://docs.oracle.com/cd/B10500_01/appdev.920/a96620/xdb05obj.htm#1656) describe how to work with XML schema-based XMLType tables and columns. |

**What Is XMLType?**

Oracle9*i* Release 1 (9.0.1) introduced a new datatype, XMLType, to facilitate native handling of XML data in the database. The following summarizes XMLType:

* XMLType can be used in PL/SQL stored procedures as parameters, return values, and variables.
* XMLType can represent an XML document as an instance (of XMLType) in SQL.
* XMLType has built-in member functions that operate on XML content. For example, you can use XMLTypefunctions to create, extract, and index XML data stored in Oracle9*i* database.
* Functionality is also available through a set of Application Program Interfaces (APIs) provided in PL/SQL and Java.

With XMLType and these capabilities, SQL developers can leverage the power of the relational database while working in the context of XML. Likewise, XML developers can leverage the power of XML standards while working in the context of a relational database.

XMLType datatype can be used as the datatype of 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 also use XMLType in SQL, PL/SQL, and Java (through JDBC).

|  |
| --- |
| **Note:**  In Oracle9*i* Release 1 (9.0.1), XMLType was only supported in the server in SQL, PL/SQL, and Java. In Oracle9*i* Release 2 (9.2), XMLType is also supported on the client side through SQL, Java, and protocols such as FTP and HTTP/WebDav. |

A number of useful functions that operate on XML content are provided. Many of these are provided as both SQL and member functions of XMLType. For example, the extract() function extracts a specific node(s) from an XMLType instance.

You can use XMLType in SQL queries in the same way as any other user-defined datatypes in the system.

|  |
| --- |
| **See Also:**   * ["Oracle XML DB Offers Faster Storage and Retrieval of Complex XML Documents"](http://docs.oracle.com/cd/B10500_01/appdev.920/a96620/xdb01int.htm" \l "1047784)  * [Chapter 26, "Oracle XML DB Basic Demo"](http://docs.oracle.com/cd/B10500_01/appdev.920/a96620/xdbxcases2.htm" \l "1047171)  * *[Oracle9i SQL Reference](http://docs.oracle.com/cd/B10500_01/server.920/a96540/toc.htm)* Appendix D, "Using XML in SQL Statements" |

**Benefits of the XMLType Data Type and API**

The XMLType datatype and API provides significant advantages. It enables SQL operations on XML content, as well as XML operations on SQL content:

* ***Versatile API.*** XMLType has a versatile API for application development, as it includes built-in functions, indexing support, navigation, and so on.
* ***XMLType and SQL***. You can use XMLType in SQL statements combined with other columns and datatypes. For example, you can query XMLType columns and join the result of the extraction with a relational column, and then Oracle can determine an optimal way to execute these queries.
* ***Optimized evaluation using XMLType***. XMLType is optimized to not materialize the XML data into a tree structure unless needed. Therefore 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.
* ***Indexing.*** Oracle Text index has been enhanced to support XMLType columns. You can also create function-based indexes on existsNode() and extract() functions to speed up query evaluation.

|  |
| --- |
| **See Also:**  [Chapter 10, "Generating XML Data from the Database"](http://docs.oracle.com/cd/B10500_01/appdev.920/a96620/xdb12gen.htm#1656) |

**When to Use XMLType**

Use XMLType when you need to perform the following:

* SQL queries on part of or the whole XML document: The functions existsNode() and extract() provide the necessary SQL query functions over XML documents.
* 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.
* XPath functionality provided by extract() and existsNode() functions: Note that XMLType uses the built-in C XML parser and processor and hence provides better performance and scalability when used inside the server.
* Indexing on XPath searches on documents: XMLType has member functions that you can use to create function-based indexes to optimize searches.
* To shield applications from storage models. 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.
* To prepare for future optimizations. New XML functionality will support XMLType. Since Oracle9*i* database is natively aware that XMLType can store XML data, better optimizations and indexing techniques can be done. By writing applications to use XMLType, these optimizations and enhancements can be easily achieved and preserved in future releases without your needing to rewrite applications.

**Storing XMLType Data in Oracle XML DB**

XMLType data can be stored in two ways or a combination thereof:

* ***In Large Objects (LOBs)***. LOB storage maintains content accuracy to the original XML (whitespaces and all). Here the XML documents are stored composed as whole documents like files. In this release, for non-schema-based storage, XMLType offers a CLOB storage option. In future releases, Oracle may provide other storage options, such as BLOBs, NCLOBS, and so on. You can also create a CLOB-based storage for XML schema-based storage.

When you create an XMLType column without any XML schema specification, 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 CLOB column; however, you can set the storage characteristics for the column using the XMLType storage clause.

* ***In Structured storage (in tables and views)***. Structured storage maintains DOM (Document Object Model) fidelity. Here the XML documents are 'broken up (decomposed)' into object- relational tables or views.XMLType achieves DOM fidelity by maintaining information that SQL or Java objects normally do not provide for, such as:
  + Ordering of child elements and attributes.
  + Distinguishing between elements and attributes.
  + Unstructured content declared in the schema. For example, content="mixed" or <any> declarations.
  + Undeclared data in instance documents, such as processing instructions, comments, and namespace declarations.
  + Support for basic XML datatypes not available in SQL (Boolean, QName, and so on).
  + Support for XML constraints (facets) not supported directly by SQL, such as enumerated lists.

Native XMLType instances contain hidden columns that store this extra information that does not quite fit in the SQL object model. This information can be accessed through APIs in SQL or Java, using member functions, such as extractNode().

Changing XMLType storage from structured storage to LOB, or vice versa, is possible using database IMPORT and EXPORT. Your application code does not have to change. You can then change XML storage options when tuning your application, since each storage option has its own benefits.

**Pros and Cons of XML Storage Options in Oracle XML DB**

[Table 4-1](http://docs.oracle.com/cd/B10500_01/appdev.920/a96620/xdb04cre.htm" \l "1035076) summarizes some advantages and disadvantages to consider when selecting your Oracle XML DB storage option.

***Table 4-1 XML Storage Options in Oracle XML DB***

| **Feature** | **LOB Storage (with Oracle Text index)** | **Structured Storage (with B\*Tree index)** |
| --- | --- | --- |
| Database schema flexibility | Very flexible when schemas change. | Limited flexibility for schema changes. Similar to the ALTER TABLE restrictions. |
| Data integrity and accuracy | Maintains the original XML byte for byte - important in some applications. | Trailing new lines, whites pace within tags, and data format for non-string datatypes is lost. But maintains DOM fidelity. |
| Performance | Mediocre performance for DML. | Excellent DML performance. |
| Access to SQL | Some accessibility to SQL features. | Good accessibility to existing SQL features, such as constraints, indexes, and so on |
| Space needed | Can consume considerable space. | Needs less space in particular when used with an Oracle XML DB registered XML schema. |

**When to Use CLOB Storage for XMLType**

Use CLOB storage for XMLType in the following cases:

* You need to store XML as a whole document in the database and retrieve it as a whole document.
* You do not need to perform piece-wise updates on XML documents.

|  |
| --- |
| **Note:**  **XMLType and Varray:**   * + You cannot create VARRAYs of XMLType and store them in the database since VARRAYs do not support CLOBs when stored in tables.   + You cannot create columns of VARRAY types that contain XMLType. This is because Oracle does not support LOB locators inside VARRAYs. |



|  |
| --- |
| **See Also:**   * + [Chapter 2, "Getting Started with Oracle XML DB"](http://docs.oracle.com/cd/B10500_01/appdev.920/a96620/xdb02rep.htm" \l "1656)  * + [Chapter 3, "Using Oracle XML DB"](http://docs.oracle.com/cd/B10500_01/appdev.920/a96620/xdb03usg.htm" \l "1656), ["Storing XML: Structured or Unstructured Storage"](http://docs.oracle.com/cd/B10500_01/appdev.920/a96620/xdb03usg.htm#1037679)  * + [Chapter 10, "Generating XML Data from the Database"](http://docs.oracle.com/cd/B10500_01/appdev.920/a96620/xdb12gen.htm" \l "1656), for information on how to generate XMLType data. |

**XMLType Member Functions**

Oracle9*i* Release 1 (9.0.1) introduced several SQL functions and XMLType member functions that operate onXMLType values. Oracle9*i* Release 2 (9.2) has expanded functionality. It provides several new SQL functions and XMLType member functions.

|  |
| --- |
| **See Also:**   * [Appendix F, "Oracle XML DB XMLType API, PL/SQL and Resource PL/SQL APIs: Quick Reference"](http://docs.oracle.com/cd/B10500_01/appdev.920/a96620/appfspe2.htm" \l "621642)  * *[Oracle9i XML API Reference - XDK and Oracle XML DB](http://docs.oracle.com/cd/B10500_01/appdev.920/a96616/toc.htm)* for a list of all XMLTypeand member functions, their syntax, and descriptions. |

All XMLType functions use the built-in C parser and processor to parse XML data, validate it, and apply XPath expressions on it. They also use an optimized in-memory DOM tree for processing, such as extracting XML documents or fragments.

|  |
| --- |
| **See Also:**  [Appendix C, "XPath and Namespace Primer"](http://docs.oracle.com/cd/B10500_01/appdev.920/a96620/appcxpa.htm#621642) |

**How to Use the XMLType API**

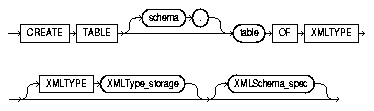
You can use the XMLType API to create tables and columns. The createXML() static function of the XMLTypeAPI can be used to create XMLType instances for insertion. By storing your XML documents as XMLType, XML content can be readily searched using standard SQL queries.

[Figure 4-1](http://docs.oracle.com/cd/B10500_01/appdev.920/a96620/xdb04cre.htm" \l "1037446) shows the syntax for creating an XMLType table:

CREATE TABLE [schema.] table OF XMLTYPE

[XMLTYPE XMLType\_storage] [XMLSchema\_spec];

***Figure 4-1 Creating an XMLType Table***

 [Text description of the illustration XMLType\_table.gif](http://docs.oracle.com/cd/B10500_01/appdev.920/a96620/img_text/XMLType_table.htm)

This section shows some simple examples of how to create an XMLType column and use it in a SQL statement, and how to create XMLType tables.

**Creating, Adding, and Dropping XMLType Columns**

The following are examples of creating, adding, and dropping XMLType columns:

***Example 4-1 Creating XMLType: Creating XMLType Columns***

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

CREATE TABLE warehouses(

warehouse\_id NUMBER(4),

warehouse\_spec XMLTYPE,

warehouse\_name VARCHAR2(35),

location\_id NUMBER(4));

***Example 4-2 Creating XMLType: Creating XMLType Columns***

As explained, you can create XMLType columns by simply using the XMLType as the datatype. The following statement creates a purchase order document column, poDoc, of XMLType:

CREATE TABLE po\_xml\_tab(

poid number,

poDoc XMLTYPE);

CREATE TABLE po\_xtab of XMLType; -- this creates a table of XMLType. The default

-- is CLOB based storage.

***Example 4-3 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:

ALTER TABLE po\_xml\_tab add (custDoc XMLType);

***Example 4-4 Dropping XMLType Columns***

You can alter tables to drop XMLType columns, similar to any other datatype. The following statement drops column custDoc:

ALTER TABLE po\_xml\_tab drop (custDoc);

**Inserting Values into an XMLType Column**

To insert values into the XMLType column, you need to bind an XMLType instance.

***Example 4-5 Inserting into XMLTYpe Using the XMLType() Constructor***

An XMLType instance can be easily created from a VARCHAR or a Character Large Object (CLOB) by using theXMLType() constructor:

INSERT INTO warehouses VALUES

( 100, XMLType(

'<Warehouse whNo="100">

<Building>Owned</Building>

</Warehouse>'), 'Tower Records', 1003);

This example creates an XMLType instance from a string literal. The input to createXML() can be any expression that returns a VARCHAR2 or CLOB. createXML() also checks that the input XML is well-formed.

**Using XMLType in an SQL Statement**

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

***Example 4-6 Using XMLType and in a SELECT Statement***

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

-----------------

Owned

|  |
| --- |
| **See Also:**  ["How to Use the XMLType API"](http://docs.oracle.com/cd/B10500_01/appdev.920/a96620/xdb04cre.htm#1030582) . |

**Updating an XMLType Column**

An XML document in an XMLType can be stored packed in a CLOB. Then updates have to replace the whole document in place.

***Example 4-7 Updating XMLType***

To update an XML document, you can execute a standard SQL UPDATE statement. You need to bind anXMLType instance, as follows:

UPDATE warehouses SET warehouse\_spec = XMLType

('<Warehouse whono="200">

<Building>Leased</Building>

</Warehouse>');

This example created an XMLType instance from a string literal and updates column warehouse\_spec with the new value.

|  |
| --- |
| **Note:**  Any triggers would get fired on the UPDATE statement You can see and modify the XML value inside the triggers. |

**Deleting a Row Containing an XMLType Column**

Deleting a row containing an XMLType column is no different from deleting a row containing any other datatype.

***Example 4-8 Deleting an XMLType Column Row***

You can use 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, you can write a statement such as:

DELETE FROM warehouses e

WHERE e.warehouse\_spec.extract('//Building/text()').getStringVal()

= 'Leased';

|  |
| --- |
| **Note:**  In this release, Oracle supports XMLType as a public synonym for sys.XMLType. XMLTypenow also supports a set of user-defined constructors (mirroring the createXML static functions). For example:   * In Oracle9*i* Release 1 (9.0.1), you could use the following syntax:sys.XMLType.createXML('<Warehouse whNo="100">...) * In Oracle9*i* Release 2 (9.2), you can use the following abbreviated version:XMLType('<Warehouse whNo="100">...). |

**Guidelines for Using XMLType Tables and Columns**

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

**Define table/column of XMLType**

First, define a table/column of XMLType. You can include optional storage characteristics with the table/column definition.

|  |
| --- |
| **Note:**  This release of Oracle supports creating tables of XMLType. You can create object references (REFs) to these tables and use them in the object cache. |

**Create an XMLType Instance**

Use the XMLType constructor to create the XMLType instance before inserting into the column/table. You can also use a variety of other functions that return XMLType.

|  |
| --- |
| **See Also:**  ["SYS\_XMLGEN(): Converting an XMLType Instance"](http://docs.oracle.com/cd/B10500_01/appdev.920/a96620/xdb12gen.htm#1031573) , for an example. |

**Select or Extract a Particular XMLType Instance**

You can select out the XMLType instance from the column. XMLType also offers a choice of member functions, such as extract() and existsNode(), to extract a particular node and to check to see if a node exists respectively. See the table of XMLType member functions in [*Oracle9i XML API Reference - XDK and Oracle XML DB*](http://docs.oracle.com/cd/B10500_01/appdev.920/a96616/toc.htm).

|  |
| --- |
| **See Also:**   * ["Selecting XMLType Columns using getClobVal()"](http://docs.oracle.com/cd/B10500_01/appdev.920/a96620/xdb04cre.htm" \l "1024814)  * ["Extracting Fragments from XMLType Using extract()"](http://docs.oracle.com/cd/B10500_01/appdev.920/a96620/xdb04cre.htm" \l "1025063) |

**You can Define an Oracle Text Index**

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

**You Can Define XPath Index, CTXXPATH**

In this release, a new Oracle Text index type, CTXXPATH is introduced. This helps existsNode() implement indexing and optimizes the evaluation of existsNode() in a predicate.

|  |
| --- |
| **See Also:**   * ["Indexing XMLType Columns"](http://docs.oracle.com/cd/B10500_01/appdev.920/a96620/xdb04cre.htm" \l "1031630)  * [Chapter 7, "Searching XML Data with Oracle Text"](http://docs.oracle.com/cd/B10500_01/appdev.920/a96620/xdb11sea.htm" \l "1006757)  * [Chapter 10, "Generating XML Data from the Database"](http://docs.oracle.com/cd/B10500_01/appdev.920/a96620/xdb12gen.htm" \l "1656)  * *[Oracle9i Application Developer's Guide - Large Objects (LOBs)](http://docs.oracle.com/cd/B10500_01/appdev.920/a96591/toc.htm)* |

**Specifying Storage Characteristics on XMLType Columns**

XML data in an XMLType column can be stored as a CLOB column. Hence you can also specify LOB storage characteristics for that column. In example, ["Creating XMLType: Creating XMLType Columns"](http://docs.oracle.com/cd/B10500_01/appdev.920/a96620/xdb04cre.htm#1035601), thewarehouse\_spec column is an XMLType column.

***Example 4-9 Specifying Storage When Creating an XMLType Table***

You can specify storage characteristics on this column when creating the table as follows:

CREATE TABLE po\_xml\_tab(

poid NUMBER(10),

poDoc XMLTYPE

)

XMLType COLUMN poDoc

STORE AS CLOB (

TABLESPACE lob\_seg\_ts

STORAGE (INITIAL 4096 NEXT 4096)

CHUNK 4096 NOCACHE LOGGING

);

The STORE AS clause is also supported when adding columns to a table.

***Example 4-10 Adding an XMLType Columns and Specifying Storage***

To add a new XMLType column to this table and specify the storage clause for that column, you can use the following SQL statement:

ALTER TABLE po\_xml\_tab add(

custDoc XMLTYPE

)

XMLType COLUMN custDoc

STORE AS CLOB (

TABLESPACE lob\_seg\_ts

STORAGE (INITIAL 4096 NEXT 4096)

CHUNK 4096 NOCACHE LOGGING

);

**Changing Storage Options on an XMLType Column Using XMLData**

In non- schema-based storage, you can use XMLDATA to change storage characteristics on an XMLType column.

***Example 4-11 Changing Storage Characteristics on an XMLType Column Using XMLDATA***

For example, consider table foo\_tab:

CREATE TABLE foo\_tab (a xmltype);

To change the storage characteristics of LOB column a in foo\_tab, you can use the following statement:

ALTER TABLE foo\_tab MODIFY LOB (a.xmldata) (storage (next 5K) cache);

XMLDATA identifies the internal storage column. In the case of CLOB-based storage this corresponds to the CLOB column. The same holds for XML schema-based storage. You can use XMLDATA to explore structured storage and modify the values.

|  |
| --- |
| **Note:**  In this release, the XMLDATA attribute helps access the XMLType's internal storage columns so that you can specify storage characteristics, constraints, and so on directly on that column. |

You can use the XMLDATA attribute in constraints and indexes, in addition to storage clauses.

|  |
| --- |
| **See also:**  [*Oracle9i Application Developer's Guide - Large Objects (LOBs)*](http://docs.oracle.com/cd/B10500_01/appdev.920/a96591/toc.htm) f and [*Oracle9i SQL Reference*](http://docs.oracle.com/cd/B10500_01/server.920/a96540/toc.htm) for more information about LOB storage options |

**Specifying Constraints on XMLType Columns**

You can specify NOT NULL constraint on an XMLType column.

***Example 4-12 Specifying Constraints on XMLType Columns***

CREATE TABLE po\_xml\_tab (

poid number(10),

poDoc XMLType NOT NULL

);

prevents inserts such as:

INSERT INTO po\_xml\_tab (poDoc) VALUES (null);

***Example 4-13 Using ALTER TABLE to Change NOT NULL of XMLType Columns***

You can also use the ALTER TABLE statement to change NOT NULL information of an 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);

You can also define check constraints on XMLType columns. Other default values are not supported on this datatype.

**Manipulating XML Data in XMLType Columns/Tables**

Since XMLType is a user-defined data type with functions defined on it, you can invoke functions on XMLTypeand obtain results. You can use XMLType wherever you use a user-defined type, including for table columns, views, trigger bodies, and type definitions.

You can perform the following manipulations or Data Manipulation Language (DML) on XML data in XMLTypecolumns and tables:

* [Inserting XML Data into XMLType Columns/Tables](http://docs.oracle.com/cd/B10500_01/appdev.920/a96620/xdb04cre.htm" \l "1032735)

* [Selecting and Querying XML Data](http://docs.oracle.com/cd/B10500_01/appdev.920/a96620/xdb04cre.htm" \l "1024805)

* [Updating XML Instances and Data in Tables and Columns](http://docs.oracle.com/cd/B10500_01/appdev.920/a96620/xdb04cre.htm" \l "1032612)

* [Deleting XML Data](http://docs.oracle.com/cd/B10500_01/appdev.920/a96620/xdb04cre.htm" \l "1032654)

**Inserting XML Data into XMLType Columns/Tables**

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. See [Chapter 22, "Loading XML Data into Oracle XML DB"](http://docs.oracle.com/cd/B10500_01/appdev.920/a96620/xdb26loa.htm#1656)

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

**Using INSERT Statements**

To 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:

* By using XMLType constructors. This can be done in SQL, PL/SQL, and Java.
* By using SQL functions like XMLElement(), XMLConcat(), and XMLAGG(). This can be done in SQL, PL/SQL, and Java.

***Example 4-14 Inserting XML Data Using createXML() with CLOB***

The following examples use INSERT...SELECT and the XMLType constructor to first create an XML document and then insert the document into the XMLType columns. Consider table po\_clob\_tab that contains a CLOB,poClob, for storing an XML document:

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>');

***Example 4-15 Inserting XML Data Using an XMLType Instance***

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:

INSERT INTO po\_xml\_tab

SELECT poid, XMLType(poClob)

FROM po\_clob\_tab;

|  |
| --- |
| **Note:**  You can also get the CLOB value from any expression, including functions that can create temporary CLOBs or select out CLOBs from other table or views. |

***Example 4-16 Inserting XML Data Using XMLType() with String***

This example inserts a purchase order into table po\_tab using the XMLType constructor:

INSERT INTO po\_xml\_tab

VALUES(100, XMLType**(**'<?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>'**)**);

***Example 4-17 Inserting XML Data Using XMLElement()***

This example inserts a purchase order into table po\_xml\_tab by generating it using the XMLElement() SQL function. Assume that the purchase order is an object view that contains a purchase order object. The whole definition of the purchase order view is given in "DBMS\_XMLGEN: Generating a Purchase Order from the Database in XML Format".

INSERT INTO po\_xml\_tab

SELECT XMLelement("po", value(p))

FROM po p

WHERE p.pono=2001;

XMLElement() creates an XMLType from the purchase order object, which is then inserted into tablepo\_xml\_tab. You can also use SYS\_XMLGEN() in the INSERT statement.

**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 existsNode()
* By using Oracle Text operators to query the XML content. See ["Indexing XMLType Columns"](http://docs.oracle.com/cd/B10500_01/appdev.920/a96620/xdb04cre.htm#1031630) and[Chapter 7, "Searching XML Data with Oracle Text"](http://docs.oracle.com/cd/B10500_01/appdev.920/a96620/xdb11sea.htm#1006757).

**SQL Functions for Manipulating XML data**

SQL functions such as existsNode(), extract(), XMLTransform(), and updateXML() operate on XML data inside SQL. XMLType datatype supports most of these as member functions. You can use either the selfish style of invocation or the SQL functions.

**Selecting XML Data**

You can select XMLType data using PL/SQL or Java. You can also use the getClobVal(), getStringVal(), orgetNumberVal() functions to retrieve XML as a CLOB, VARCHAR, or NUMBER, respectively.

***Example 4-18 Selecting XMLType Columns using getClobVal()***

This example shows how to select an XMLType column using 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**

You can query XMLType data and extract portions of it using the existsNode() and extract() functions. Both these functions use a subset of the W3C XPath recommendation to navigate the document.

**Using XPath Expressions for Searching XML Documents**

XPath is a W3C recommendation for navigating 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 "PONO" child elements under the "PO" root element of the document.

[Table 4-2](http://docs.oracle.com/cd/B10500_01/appdev.920/a96620/xdb04cre.htm" \l "1025285) lists some common constructs used in XPath.

***Table 4-2 Some Common XPath Constructs***

| **XPath Construct** | **Description** |
| --- | --- |
| "/" | Denotes the root of the tree in an XPath expression. For example, /PO refers to the child of the root node whose name is "PO". |
| "/" | Also used as a path separator to identify the children node of any given node. For example, /PO/PNAME identifies the purchase order name element, a child of the root element. |
| "//" | Used to identify all descendants of the current node. For example, PO//ZIP matches any zip code element under the "PO" element. |
| "\*" | Used as a wildcard to match any child node. For example, /PO/\*/STREET matches any street element that is a grandchild of the "PO" element. |
| [ ] | Used to denote predicate expressions. XPath supports a rich list of binary operators such as OR, AND, and NOT. For example, /PO[PONO=20 and PNAME="PO\_2"]/SHIPADDR select out the shipping address element of all purchase orders whose purchase order number is 20 and whose purchase order name is "PO\_2". [ ] is also used for denoting an index into a list. For example, /PO/PONO[2] identifies the second purchase order number element under the "PO" root element. |

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.

|  |
| --- |
| **See Also:**  [Appendix C, "XPath and Namespace Primer"](http://docs.oracle.com/cd/B10500_01/appdev.920/a96620/appcxpa.htm#621642) |

**Querying XML Data Using XMLType Member Functions**

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

***Example 4-19 Retrieving an XML Document as a CLOB Using getClobVal() and existsNode()***

This example shows how to select an XMLType column using getClobVal() and existsNode():

set long 2000

SELECT e.poDoc.getClobval() AS poXML

FROM po\_xml\_tab e

WHERE e.poDoc.existsNode('/PO[PNAME = "po\_2"]') = 1;

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>

**existsNode Function**

The syntax for the existsNode() function is described in [Figure 4-2](http://docs.oracle.com/cd/B10500_01/appdev.920/a96620/xdb04cre.htm#1032193) and also as follows:

existsNode(XMLType\_instance IN XMLType,

XPath\_string IN VARCHAR2, namespace\_string IN varchar2 := null)

RETURN NUMBER

***Figure 4-2 existsNode() Syntax***

Text description of existsnode.gif follows [Text description of the illustration existsnode.gif](http://docs.oracle.com/cd/B10500_01/appdev.920/a96620/img_text/existsnode.htm)

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. Namespace can be used to identify the mapping of prefix(es) specified in the XPath\_string to the corresponding namespace(s).

***Example 4-20 Using existsNode() on XMLType***

For example, consider an XML document such as:

<PO>

<PONO>100</PONO>

<PNAME>Po\_1</PNAME>

<CUSTOMER CUSTNAME="John"/>

<SHIPADDR>

<STREET>1033, Main Street</STREET>

<CITY>Sunnyvalue</CITY>

<STATE>CA</STATE>

</SHIPADDR>

</PO>

An XPath expression such as /PO/PNAME results in a single node. Therefore, existsNode() will return 1 for that XPath. This is the same with /PO/PNAME/text(), which results in a single text node.

An XPath expression such as /PO/POTYPE does not return any nodes. Therefore, an existsNode() on this would return the value 0.

To summarize, existsNode() member function can be used in queries and to create function-based indexes to speed up evaluation of queries.

***Example 4-21 Using existsNode() to Find a node***

The following example tests for the existence of the /Warehouse/Dock node in the warehouse\_spec column XML path of the sample table oe.warehouses:

SELECT warehouse\_id, EXISTSNODE(warehouse\_spec, '/Warehouse/Docks')

"Loading Docks"

FROM warehouses

WHERE warehouse\_spec IS NOT NULL;

WAREHOUSE\_ID Loading Docks

------------ -------------

1 1

2 1

3 0

4 1

**Using Indexes to Evaluate existsNode()**

You can create function-based indexes using existsNode() to speed up the execution. You can also create aCTXXPATH index to help speed up arbitrary XPath searching.

|  |
| --- |
| **See Also:**  ["Creating XPath Indexes on XMLType Columns: CTXXPATH Index"](http://docs.oracle.com/cd/B10500_01/appdev.920/a96620/xdb04cre.htm#1034245) |

**extract****() Function**

The extract() function is similar to the existsNode() function. It applies a VARCHAR2 XPath string with an optional namespace parameter and returns an XMLType instance containing an XML fragment. The syntax is described in [Figure 4-3](http://docs.oracle.com/cd/B10500_01/appdev.920/a96620/xdb04cre.htm#1035784) and as follows:

extract(XMLType\_instance IN XMLType, XPath\_string IN VARCHAR2,

namespace\_string In varchar2 := null) RETURN XMLType;

***Figure 4-3 extract() Syntax***

Text description of extract_xml.gif follows [Text description of the illustration extract\_xml.gif](http://docs.oracle.com/cd/B10500_01/appdev.920/a96620/img_text/extract_xml.htm)

extract() on XMLType extracts the node or a set of nodes from the document identified by the XPath expression. The extracted nodes can be elements, attributes, or text nodes. When extracted out, all text nodes are collapsed into a single text node value. Namespace can be used to supply namespace information for prefixes in the XPath string.

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() orgetNumberVal() methods on XMLType to extract this scalar data.

For example, the XPath expression /PO/PNAME identifies the PNAME element inside the XML document shown previously. The expression /PO/PNAME/text(), on the other hand, refers to the text node of the PNAMEelement.

|  |
| --- |
| **Note:**  The latter is still considered an XMLType. In other words,extract(poDoc,'/PO/PNAME/text()') still returns an XMLtype instance although the instance may actually contain only text. You can use getStringVal() to get the text value out as a VARCHAR2 result. |

Use text() node test function to identify text nodes in elements before using the getStringVal() orgetNumberVal() to convert them to SQL data. Not having the text() node would produce an XML fragment.

For example, XPath expressions:

* /PO/PNAME identifies the fragment <PNAME>PO\_1</PNAME>
* /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 you have an XML document such as:

<PO>

<PONO>100</PONO>

<PONO>200</PONO>

</PO>

you can use:

* //PONO[1] to identify the first "PONO" element (with value 100).
* //PONO[2] to identify the second "PONO" element (with value 200).

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

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

* Extracting numerical values on which function-based 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

***Example 4-22 Using extract() to Extract the Value of a Node***

This example extracts the value of node, /Warehouse/Docks, of column, warehouse\_spec in tableoe.warehouses:

SELECT warehouse\_name,

extract(warehouse\_spec, '/Warehouse/Docks').getStringVal()

"Number of Docks"

FROM warehouses

WHERE warehouse\_spec IS NOT NULL;

WAREHOUSE\_NAME Number of Docks

-------------------- --------------------

Southlake, Texas <Docks>2</Docks>

San Francisco <Docks>1</Docks>

New Jersey <Docks/>

Seattle, Washington <Docks>3</Docks>

**extractValue() Function**

The extractValue() function takes as arguments an XMLType instance and an XPath expression. It returns a scalar value corresponding to the result of the XPath evaluation on the XMLType instance. extractValue()syntax is also described in [Figure 4-4](http://docs.oracle.com/cd/B10500_01/appdev.920/a96620/xdb04cre.htm#1035874).

* **XML schema-based documents**. For documents based on XML schema, if Oracle9*i* can infer the type of the return value, then a scalar value of the appropriate type is returned. Otherwise, the result is of typeVARCHAR2.
* **Non- schema-based documents.** For documents not based on XML schemas, the return type is alwaysVARCHAR2.

extractValue() tries to infer the proper return type from the XML schema of the document. If the XMLType is non- schema-based or the proper return type cannot be determined, Oracle XML DB returns a VARCHAR2.

***Figure 4-4 extractValue() Syntax***

Text description of extractvalue.gif follows [Text description of the illustration extractvalue.gif](http://docs.oracle.com/cd/B10500_01/appdev.920/a96620/img_text/extractvalue.htm)

**A Shortcut Function**

extractValue() permits you to extract the desired value more easily than when using the equivalent extract function. It is an ease-of-use and shortcut function. So instead of using:

extract(x,'path/text()').get(string|num)val()

you can replace extract().getStringVal() or extract().getnumberval() with extractValue() as follows:

extractValue(x, 'path/text()')

With extractValue() you can leave off the text(), but ONLY if the node pointed to by the 'path' part has only one child and that child is a text node. Otherwise, an error is thrown.

extractValue() syntax is the same as extract().

**extractValue() Characteristics**

extractValue() has the following characteristics:

* It always returns only scalar content, such as NUMBER...VARCHAR2, and so on.
* It cannot return XML nodes or mixed content. It raises an error at compile or run time if it gets XML nodes as the result.
* It always returns VARCHAR2 by default. If the node's value is bigger than 4K, a runtime error would occur.
* In the presence of XML schema information, at compile time, extractValue() can automatically return the appropriate datatype based on the XML schema information, if it can detect so at compile time of the query. For instance, if the XML schema information for the path /PO/POID indicates that this is a numerical value, then extractValue() returns a NUMBER.
* If the XPath identifies a node, it automatically gets the scalar content from its text child. The node must have exactly one text child. For example:
* extractValue(xmlinstance, '/PO/PNAME')

extracts out the text child of PNAME. This is equivalent to:

extract(xmlinstance, '/PO/PNAME/text()').getstringval()

***Example 4-23 Extracting the Scalar Value of an XML Fragment Using extractValue()***

The following example takes as input the same arguments as the example for [extract () Function](http://docs.oracle.com/cd/B10500_01/appdev.920/a96620/xdb04cre.htm#1030924). Instead of returning an XML fragment, as extract() does, it returns the scalar value of the XML fragment:

SELECT warehouse\_name,

extractValue(e.warehouse\_spec, '/Warehouse/Docks')

"Docks"

FROM warehouses e

WHERE warehouse\_spec IS NOT NULL;

WAREHOUSE\_NAME Docks

-------------------- ------------

Southlake, Texas 2

San Francisco 1

New Jersey

Seattle, Washington 3

ExtractValue() automatically extracted out the text child of Docks element and returned that value. You can also write this using extract() as follows:

extract(e.warehouse\_spec, '/Warehouse/Docks/text()').getstringval()

**More SQL Examples That Query XML**

The following SQL examples illustrate ways you can query XML.

***Example 4-24 Querying XMLType Using extract() and existsNode()***

Assume the po\_xml\_tab table, which contains the purchase order identification and the purchase order XML columns, and assume that the following values are inserted into the table:

INSERT INTO po\_xml\_tab values (100,

xmltype('<?xml version="1.0"?>

<PO>

<PONO>221</PONO>

<PNAME>PO\_2</PNAME>

</PO>'));

INSERT INTO po\_xml\_tab values (200,

xmltype('<?xml version="1.0"?>

<PO>

<PONAME>PO\_1</PONAME>

</PO>'));

Now you can extract the numerical values for the purchase order numbers using extract():

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 nodes where "PONO" exists as a child of "PO".

|  |
| --- |
| **Note:**  Here text() function is only used to return the text nodes. getNumberVal() function can convert only text values into numerical quantity |

|  |
| --- |
| **See Also:**  ["XMLType Member Functions"](http://docs.oracle.com/cd/B10500_01/appdev.920/a96620/xdb04cre.htm#1030046) |

***Example 4-25 Querying Transient XMLType Data***

The following example shows how you can select out the XML data and query it inside PL/SQL: create a transient instance from the purchase order table and then perform some extraction on it. Assume po\_xml\_tabcontains the data shown in [Example 4-16, "Inserting XML Data Using XMLType() with String"](http://docs.oracle.com/cd/B10500_01/appdev.920/a96620/xdb04cre.htm#1024702), modified:

set serverout on

declare

poxml XMLType;

cust XMLType;

val VARCHAR2(200);

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;

/

***Example 4-26 Extracting Data from an XML Document and Inserting It Into a Table Using extract()***

The following example shows how you can extract out data from an XML purchase order and insert it into an SQL relational table. Consider the following relational tables:

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 XML of the form:

<?xml version = '1.0'?>

<PO>

<PONO>2001</PONO>

<PNAME>Po\_1</PNAME>

<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 extract().

Here is an SQL example assuming that the XML described in the previous example is present in thepo\_xml\_tab:

INSERT INTO po\_rel\_tab

SELECT p.poDoc.extract('/PO/PONO/text()').getnumberval() as pono,

p.poDoc.extract('/PO/PNAME/text()').getstringval() as pname,

-- get the customer id corresponding to the customer name

( SELECT c.custid

FROM cust\_tab c

WHERE c.custname = p.poDoc.extract('/PO/CUSTOMER/@CUSTNAME').getstringval()

) as custid,

p.poDoc.extract('/PO/SHIPADDR/STREET/text()').getstringval() as shipstreetr,

p.poDoc.extract('//CITY/text()').getstringval() as shipcity,

p.poDoc.extract('//ZIP/text()').getstringval() as shipzip

FROM po\_xml\_tab p;

Table po\_tab should now have the following values:

PONO PNAME CUSTID SHIPSTREET SHIPCITY SHIPZIP

----------------------------------------------------------------

2001 Po\_1 1001 323 College Drive Edison 08820

|  |
| --- |
| **Note:**  PNAME is null, since the input XML document did not have the element called PNAMEunder PO. Also, the preceding example used //CITY to search for the city element at any depth. |

***Example 4-27 Extracting Data from an XML Document and Inserting It Into a Table Using extract() In a PL/SQL Block***

You can do the same in an equivalent fashion inside a PL/SQL block, as follows:

DECLARE

poxml 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 := poxml.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;

/

***Example 4-28 Searching XML Data with extract() and existsNode()***

Using extract() and existsNode() functions, you can perform a variety of search 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('//PONO/text()').getNumberVal() = 300 AND

e.poDoc.extract('//@CUSTNAME').getStringVal() like '%John%';

This SQL statement extracts the purchase order name "PNAME" from purchase order element PO, from all XML documents containing a shipping address with a purchase order number of 300, and a customer name "CUSTNAME" containing the string "John".

***Example 4-29 Searching XML Data with extractValue()***

Using extractValue(), you can rewrite the preceding query as:

SELECT extractvalue(e.poDoc, '/PO/PNAME') PNAME

FROM po\_xml\_tab e

WHERE e.poDoc.existsNode('/PO/SHIPADDR') = 1 AND

extractvalue(e.poDoc,'//PONO') = 300 AND

extractvalue(e.poDoc,'//@CUSTNAME') like '%John%';

***Example 4-30 Extracting Fragments from XMLType Using extract()***

extract() member function *extracts* nodes identified by the XPath expression and returns an XMLTypecontaining 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.poDoc.extract('/PO/SHIPADDR/STATE').isFragment()

FROM po\_xml\_tab e;

|  |
| --- |
| **Note:**  You cannot insert fragments into XMLType columns. You can use SYS\_XMLGEN() to convert a fragment into a well-formed document by adding an enclosing tag. See["SYS\_XMLGEN() Function"](http://docs.oracle.com/cd/B10500_01/appdev.920/a96620/xdb12gen.htm#1026350). You can, however, query further on the fragment using the various XMLType functions. |

The previous SQL statement returns 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() is considered a fragment, since it is not a well-formed XML document.

**Updating XML Instances and Data in Tables and Columns**

This section talks about updating transient XML instances and XML data stored in tables.

With CLOB-based storage, in this release, an update effectively replaces the whole document. Use the SQLUPDATE statement to update the whole XML document. The right hand side of the UPDATE's SET clause must be an XMLType instance. This can be created using the SQL functions and XML constructors that return an XML instance, or using the PL/SQL DOM APIs for XMLType or Java DOM API, that change and bind existing XML instances.

**updateXML() SQL Function**

updateXML() function takes in a source XMLType instance, and a set of XPath value pairs. It returns a new XML instance consisting of the original XMLType instance with appropriate XML nodes updated with the given values. The optional namespace parameter specifies the namespace mapping of prefix(es) in the XPath parameters.

updateXML() can be used to update, replace elements, attributes and other nodes with new values. They cannot be directly used to insert new nodes or delete existing ones. The containing parent element should be updated with the new values instead.

Text description of updatexmla.gif follows [Text description of the illustration updatexmla.gif](http://docs.oracle.com/cd/B10500_01/appdev.920/a96620/img_text/updatexmla.htm)

updateXML() updates only the transient XML instance in memory. Use an SQL UPDATE statement to update data stored in tables. The updateXML() syntax is:

UPDATEXML(xmlinstance, xpath1, value\_expr1

[, xpath2, value\_expr2]...[,namespace\_string]);

***Example 4-31 Updating XMLType Using the UPDATE Statement***

This example updates the XMLType using the UPDATE statement. It updates only those documents whose purchase order number is 2001.

UPDATE po\_xml\_tab e

SET e.poDoc = XMLType(

'<?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.poDoc.EXTRACT('/PO/PONO/text()').getNumberVal() = 2001;

|  |
| --- |
| **Note:**  Updates for non- schema based XML documents always update the whole XML document. |

***Example 4-32 Updating XMLType Using UPDATE and updateXML()***

To update the XML document in the table instead of creating a new one, you can use the updateXML() in the right hand side of an UPDATE statement to update the document.

|  |
| --- |
| **Note:**  This will also update the whole document, not just the part updated. |

UPDATE po\_xml\_tab

SET poDoc = UPDATEXML(poDoc,

'/PO/CUSTNAME/text()', 'John');

1 row updated

SELECT e.poDoc.getstringval() AS newpo

FROM po\_xml\_tab e;

NEWPO

--------------------------------------------------------------------

<?xml version="1.0"?>

<PO pono="2">

<PNAME>Po\_2</PNAME>

<CUSTNAME>**John**</CUSTNAME>

<SHIPADDR>

<STREET>2 Avocet Drive</STREET>

<CITY>Redwood Shores</CITY>

<STATE>CA</STATE>

</SHIPADDR>

</PO>

***Example 4-33 Updating Multiple Elements in the Column Using updateXML()***

You can update multiple elements within a single updateXML() expression. For instance, you can use the same UPDATE statement as shown in the preceding example and update purchase order, po:

UPDATE emp\_tab e

SET e.emp\_col = UPDATEXML(e.emp\_col,

'/EMPLOYEES/EMP[EMPNAME="Joe"]/SALARY/text()',100000,

'//EMP[EMPNAME="Jack"]/EMPNAME/text()','Jackson',

'//EMP[EMPNO=217]',XMLTYPE.CREATEXML(

'<EMP><EMPNO>217</EMPNO><EMPNAME>Jane</EMPNAME></EMP>'))

WHERE EXISTSNODE(e.emp\_col, '//EMP') = 1;

This updates all rows that have an employee element with the new values.

***Example 4-34 Updating Customer Name in Purchase Order XML Document Using updateXML()***

The following example updates the customer name in the purchase order XML document, po:

|  |
| --- |
| **Note:**  This example only selects the document and the update occurs on a transient XMLTypeinstance. The original document is not affected. |

SELECT

UPDATEXML(poDoc,

'/PO/CUSTNAME/text()', 'John').getstringval() AS updatedPO

FROM po\_xml\_tab;

UPDATEDPO

--------------------------------------------------------------------

<?xml version="1.0"?>

<PO pono="2">

<PNAME>Po\_2</PNAME>

<CUSTNAME>John</CUSTNAME>

<SHIPADDR>

<STREET>2 Avocet Drive</STREET>

<CITY>Redwood Shores</CITY>

<STATE>CA</STATE>

</SHIPADDR>

</PO>

***Example 4-35 Updating Multiple Transient XML Instances Using updateXML()***

You can also use updateXML() to update multiple pieces of a transient instance. For example, consider the following XML document stored in column emp\_col of table, emp\_tab:

<EMPLOYEES>

<EMP>

<EMPNO>112</EMPNO>

<EMPNAME>Joe</EMPNAME>

<SALARY>50000</SALARY>

</EMP>

<EMP>

<EMPNO>217</EMPNO>

<EMPNAME>Jane</EMPNAME>

<SALARY>60000</SALARY>

</EMP>

<EMP>

<EMPNO>412</EMPNO>

<EMPNAME>Jack</EMPNAME>

<SALARY>40000</SALARY>

</EMP>

</EMPLOYEES>

To generate a new document with Joe's salary updated to 100,000, update the Name of Jack to Jackson, and modify the Employee element for 217, to remove the salary element. You can write a query such as:

SELECT UPDATEXML(emp\_col, '/EMPLOYEES/EMP[EMPNAME="Joe"]/SALARY/text()', 100000,

'//EMP[EMPNAME="Jack"]/EMPNAME/text()','Jackson',

'//EMP[EMPNO=217]',

XMLTYPE.CREATEXML('<EMP><EMPNO>217</EMPNO><EMPNAME>Jane</EMPNAME>'))

FROM emp\_tab e;

This generates the following updated XML:

<EMPLOYEES>

<EMP>

<EMPNO>112</EMPNO>

<EMPNAME>Joe</EMPNAME>

<SALARY>100000</SALARY>

</EMP>

<EMP>

<EMPNO>217</EMPNO>

<EMPNAME>Jane</EMPNAME>

</EMP>

<EMP>

<EMPNO>412</EMPNO>

<EMPNAME>Jackson</EMPNAME>

<SALARY>40000</SALARY>

</EMP>

</EMPLOYEES>

**Creating Views of XML Data with updateXML()**

You can use updateXML() to create new views of XML data. This can be useful when you do not want a particular set of users to see sensitive data such as SALARY.

***Example 4-36 Creating Views Using updateXML()***

A view such as:

CREATE VIEW new\_emp\_view

AS SELECT

UPDATEXML(emp\_col, '/EMPLOYEES/EMP/SALARY/text()', 0) emp\_view\_col

FROM emp\_tab e;

ensures that users selecting from view, new\_emp\_view, do not see the SALARY field for any employee.

**Optimization of updateXML()**

In most cases, updateXML() materializes the whole input XML document in memory and updates the values. However, it is optimized for UPDATE statements on XML schema-based object-relationally stored XMLType tables and columns so that the function updates the value directly in the column.

The conditions for rewrite are explained in [Chapter 5, "Structured Mapping of XMLType"](http://docs.oracle.com/cd/B10500_01/appdev.920/a96620/xdb05obj.htm#1656), ["Query Rewrite with XML Schema-Based Structured Storage"](http://docs.oracle.com/cd/B10500_01/appdev.920/a96620/xdb05obj.htm#1038780), in detail. If all of the rewrite conditions are met, then the updateXML()is rewritten to update the object-relational columns directly with the values. For example, the following UPDATE statement:

UPDATE po\_xml\_tab

SET poDoc = UPDATEXML(poDoc,

'/PO/CUSTNAME/text()', 'John');

could get rewritten (if the rewrite rules are satisfied) to an UPDATE of the custname column directly:

UPDATE po\_xml\_tab p

SET p.xmldata.CUSTNAME = 'John';

**updateXML() and NULL Values**

If you update an XML element to null, Oracle removes the attributes and children of the element, and the element becomes empty. The type and namespace properties of the element are retained. A NULL value for an element update is equivalent to setting the element to empty.

If you update the text node of an element to null, Oracle removes the text value of the element, and the element itself remains but is empty. For example, if you update node, '/empno/text()' with a NULL value, the text values for the empno element are removed and the empno element becomes empty.

Setting an attribute to NULL, similarly sets the value of the attribute to the empty string.

You cannot use updateXML() to remove, add, or delete a particular element or an attribute. You have to update the *containing* element with a new value.

|  |
| --- |
| **Note:**  Setting 'empno' to NULL has the same effect as setting 'empno/text()' to NULL, if empnois a simple scalar element with no attributes. |

***Example 4-37 NULL Updates with updateXML()***

Consider the XML document:

<PO>

<pono>21</pono>

<shipAddr gate="xxx">

<street>333</street>

<city>333</city>

</shipAddr>

</PO>

The clause:

updateXML(xmlcol,'/PO/shipAddr',null)

is equivalent to making it:

<PO>

<pono>21</pono>

<shipAddr/>

</PO>

If you update the text node to NULL, then this is equivalent to removing the text value alone. For example:

UPDATEXML(xmlcol,'/PO/shipAddr/street/text()', null)

results in:

<PO>

<pono>21</pono>

<shipAddr>

<street/>

<city>333</city>

</shipAddr>

</PO>

**Updating the Same XML Node More Than Once**

You can update the same XML node more than once in the updateXML() statement. For example, you can update both /EMP[EMPNO=217] and /EMP[EMPNAME="Jane"]/EMPNO, where the first XPath identifies the EMPNOnode containing it as well. The order of updates is determined by the order of the XPath expressions in left-to-right order. Each successive XPath works on the result of the previous XPath update.

**XMLTransform() Function**

The XMLTransform() function takes in an XMLType instance and an XSLT stylesheet. It applies the stylesheet to the XML document and returns a transformed XML instance. See [Figure 4-5](http://docs.oracle.com/cd/B10500_01/appdev.920/a96620/xdb04cre.htm#1035934).

***Figure 4-5 XMLTransform() Syntax***

Text description of XMLTransform.gif follows [Text description of the illustration XMLTransform.gif](http://docs.oracle.com/cd/B10500_01/appdev.920/a96620/img_text/XMLTransform.htm)

XMLTransform() is explained in detail in [Chapter 6, "Transforming and Validating XMLType Data"](http://docs.oracle.com/cd/B10500_01/appdev.920/a96620/xdb07tra.htm#1656).

**Deleting XML Data**

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

***Example 4-38 Deleting Rows Using extract()***

For example, to delete all purchase order rows with a purchase order name of "Po\_2", execute a statement such as:

DELETE FROM po\_xml\_tab e

WHERE e.poDoc.extract('/PO/PNAME/text()').getStringVal()='Po\_2';

**Using XMLType In Triggers**

You can use the new and old binds inside triggers to read and modify the XMLType column values. For INSERT and UPDATE statements, you can modify the new value to change the value being inserted.

***Example 4-39 Creating XMLType Triggers***

For example, you can write a trigger to change the purchase order 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

declare

pono Number;

begin

if inserting then:

if :NEW.poDoc.existsnode('//SHIPADDR') = 0 then

:NEW.poDoc := xmltype('<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 := xmltype('<PO>INVALID\_PO</PO>');

end if;

end if;

end;

/

This example is only an illustration. 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.

**Indexing XMLType Columns**

You can create the following indexes when using XMLType. Indexing speeds up query evaluation.

**Creating Function-Based Indexes on XMLType Columns**

You can speed up by queries by building function-based indexes on existsNode() or those portions of the XML document that use extract().

***Example 4-40 Creating a Function-Based Index on an extract() Function***

For example, to speed up the search on the query,

SELECT \* FROM po\_xml\_tab e

WHERE e.poDoc.extract('//PONO/text()').getNumberVal()= 100;

you can create a function-based index on the extract() function as follows:

CREATE INDEX city\_index ON po\_xml\_tab

(poDoc.extract('//PONO/text()').getNumberVal());

The SQL query uses this function-based index, to evaluate the predicate instead of parsing the XML document row by row, and evaluating the XPath expression.

***Example 4-41 Creating a Function-Based index on an existsNode() Function***

You can also create bitmapped function-based indexes to speed up the evaluation of the operators.existsNode() is suitable, since it returns a value of 1 or 0 depending on whether the XPath is satisfied in the XML document or not.

For example, to speed up a query that searches whether the XML document contains an element calledShipping address (SHIPADDR) at any level:

SELECT \* FROM po\_xml\_tab e

WHERE e.poDoc.existsNode('//SHIPADDR') = 1;

you can create a bitmapped function-based index on the existsNode() function as follows:

CREATE BITMAP INDEX po\_index ON po\_xml\_tab

(poDoc.existsNode('//SHIPADDR'));

This speeds up the query processing.

**Creating Oracle Text Indexes on XMLType Columns**

Oracle Text index works on CLOB and VARCHAR columns. It has been extended in Oracle9*i* to also work onXMLType columns. The default behavior of Oracle Text index is to automatically create XML sections, when defined over XMLType columns. Oracle Text also provides the CONTAINS operator which has been extended to support XPath.

In general, Oracle Text indexes can be created using the CREATE INDEX SQL statement with the INDEXTYPEspecified as for other CLOB or VARCHAR columns. Oracle Text indexes on XMLType columns, however, are created as function-based indexes.

***Example 4-42 Creating an Oracle Text Index***

CREATE INDEX po\_text\_index ON

po\_xml\_tab(poDoc) indextype is ctxsys.context;

You can also perform Oracle Text operations such as CONTAINS and SCORE. on XMLType columns. In Oracle9*i*Release (9.0.1), the CONTAINS operator was enhanced to support XPath using two new operators, INPATH andHASPATH:

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

***Example 4-43 Searching XML Data Using HASPATH***

For example:

SELECT \* FROM po\_xml\_tab w

WHERE CONTAINS(w.poDoc,

'haspath(/PO[./@CUSTNAME="John Nike"])') > 0;

**QUERY\_REWRITE PRIVILEGE Is No Longer Needed**

In Oracle9*i* Release (9.0.1), to create and use Oracle Text index in queries, in addition to having the privileges for creating indexes and for creating Oracle Text indexes, you also needed privileges and settings for creating function-based indexes:

* QUERY\_REWRITE privilege. You must have this privilege granted to create text indexes on XMLType columns in your own schema.
* GLOBAL\_QUERY\_REWRITE privilege. If you need to create Oracle Text indexes on XMLType columns in other schemas or on tables residing in other schemas, you must have this privilege granted.

Oracle Text index uses the PATH\_SECTION\_GROUP as the default section group when indexing XMLTypecolumns. This default can be overridden during Oracle Text index creation.

With this release, you no longer need the additional QUERY\_REWRITE privileges when creating Oracle Text indexes.

|  |
| --- |
| **See Also:**   * [Chapter 7, "Searching XML Data with Oracle Text"](http://docs.oracle.com/cd/B10500_01/appdev.920/a96620/xdb11sea.htm" \l "1006757)  * [Chapter 10, "Generating XML Data from the Database"](http://docs.oracle.com/cd/B10500_01/appdev.920/a96620/xdb12gen.htm" \l "1656)  * *[Oracle Text Reference](http://docs.oracle.com/cd/B10500_01/text.920/a96518/toc.htm)*  * *[Oracle Text Application Developer's Guide](http://docs.oracle.com/cd/B10500_01/text.920/a96517/toc.htm)* |

|  |
| --- |
| **Note:**  The QUERY\_REWRITE\_INTEGRITY and QUERY\_REWRITE\_ENABLED session settings are no longer needed to create Oracle Text or other function-based indexes on XMLTypecolumns. |

**Creating XPath Indexes on XMLType Columns: CTXXPATH Index**

existsNode() SQL function, unlike the CONTAINS operator, cannot use Oracle Text indexes to speed up its evaluation. To improve the performance of XPath searches in existsNode(), this release introduces a new index type, CTXXPATH.

CTXXPATH index is a new indextype provided by Oracle Text. It is designed to serve as a primary filter forexistsNode() processing, that is, it produces a superset of the results that would be produced by theexistNode() function. The existsNode() functional implementation is then applied on the results to return the correct set of rows.

CTXXPATH index can handle XPath path searching, wildcards, and string equality predicates.

***Example 4-44 Using CTXXPATH Index or existsNode() for XPath Searching***

CREATE INDEX po\_text\_index ON

po\_xml\_tab(poDoc) indextype is ctxsys.ctxxpath;

For example, a query such as:

SELECT \*

FROM po\_xml\_doc w

WHERE existsNode(w.poDoc,'/PO[@CUSTNAME="John Nike"]') = 1;

could potentially use CTXXPATH indexing to satisfy the existsNode() predicate.

|  |
| --- |
| **See Also:**   * [Chapter 7, "Searching XML Data with Oracle Text"](http://docs.oracle.com/cd/B10500_01/appdev.920/a96620/xdb11sea.htm" \l "1006757)  * [Chapter 10, "Generating XML Data from the Database"](http://docs.oracle.com/cd/B10500_01/appdev.920/a96620/xdb12gen.htm" \l "1656) |

**Differences Between CONTAINS and existsNode()/extract()**

The differences in XPath support when using CONTAINS compared to XPath support with existsNode() andextract() functions are:

* 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.
* Oracle Text index may give wrong results if the XML document only has tag names and attribute names without any text. For example, consider the following XML document:
* <A>
* <B>
* <C>
* </C>
* </B>
* <D>
* <E>
* </E>
* </D>
* </A>

the XPath expression - A/B/E falsely matches the preceding XML document.

* Both the function-based indexes and Oracle Text indexes support navigation. Thus you can use the Oracle Text index as a primary filter, to filter out all documents that potentially match the criterion, efficiently, and then apply secondary filters such as existsNode() or extract() operations on the remainder of the XML documents.