Oracle® XML DB

Developer's Guide

11*g* Release 2 (11.2)

E16659-03

August 2010

This manual describes Oracle XML DB. It includes guidelines and examples for storing, generating, accessing, searching, validating, transforming, evolving, and indexing XML data in Oracle Database.

ORACLE®

Oracle XML DB Developer's Guide, 11*g* Release 2 (11.2)

E16659-03

Copyright © 2002, 2010, Oracle and/or its affiliates. All rights reserved.

Primary Author: Drew Adams

Contributing Author: Nipun Agarwal, Abhay Agrawal, Omar Alonso, David Anniss, Sandeepan Banerjee, Mark Bauer, Ravinder Booreddy, Stephen Buxton, Yuen Chan, Sivasankaran Chandrasekar, Vincent Chao, Ravindranath Chennoju, Dan Chiba, Mark Drake, Fei Ge, Janis Greenberg, Wenyun He, Shelley Higgins, Thuvan Hoang, Sam Idicula, Namit Jain, Neema Jalali, Deepti Kamal, Bhushan Khaladkar, Viswanathan Krishnamurthy, Muralidhar Krishnaprasad, Geoff Lee, Wesley Lin, Annie Liu, Anand Manikutty, Jack Melnick, Nicolas Montoya, Steve Muench, Chuck Murray, Ravi Murthy, Eric Paapanen, Syam Pannala, John Russell, Eric Sedlar, Vipul Shah, Cathy Shea, Asha Tarachandani, Tarvinder Singh, Simon Slack, Muralidhar Subramanian, Asha Tarachandani, Priya Vennapusa, James Warner

Contributor: Reema Al-Shaikh, Harish Akali, Vikas Arora, Deanna Bradshaw, Paul Brandenstein, Lisa Eldridge, Craig Foch, Wei Hu, Reema Koo, Susan Kotsovolos, Sonia Kumar, Roza Leyderman, Zhen Hua Liu, Diana Lorentz, Yasuhiro Matsuda, Valarie Moore, Bhagat Nainani, Visar Nimani, Sunitha Patel, Denis Raphaely, Rebecca Reitmeyer, Ronen Wolf

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this software or related documentation is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, duplication, disclosure, modification, and adaptation shall be subject to the restrictions and license terms set forth in the applicable Government contract, and, to the extent applicable by the terms of the Government contract, the additional rights set forth in FAR 52.227-19, Commercial Computer Software License (December 2007). Oracle USA, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

This software is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications which may create a risk of personal injury. If you use this software in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure the safe use of this software. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software in dangerous applications.

Oracle is a registered trademark of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

This software and documentation may provide access to or information on content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services.

Contents

Preface xli

Audience xli
Documentation Accessibility xli
Related Documents xliii
Conventions xliv
Code Examples xliv
Syntax Descriptions xlv

What's New in Oracle XML DB? xlvii

Oracle Database 11g Release 2 (11.2.0.2) New Features in Oracle XML DB xlvii
Oracle Database 11g Release 2 (11.2.0.2) Deprecated Oracle XML DB Constructs xlviii
Oracle Database 11g Release 2 (11.2.0.1) New Features in Oracle XML DB xlviii
Oracle Database 11g Release 2 (11.2.0.1) Deprecated Oracle XML DB Constructs xlix
Oracle Database 11g Release 1 (11.1) New Features in Oracle XML DB l

Part I Oracle XML DB Basics

1 Introduction to Oracle XML DB

Overview of Oracle XML DB 1-1
Oracle XML DB Architecture 1-2
APIs for XML 1-4
Catalog Views Related to XML 1-6
Overview of Oracle XML DB Repository 1-7
XML Services 1-7
Views RESOURCE_VIEW and PATH_VIEW 1-8
Oracle XML DB Repository Architecture 1-8
Files and Folders 1-9
Oracle XML DB Protocol Architecture 1-10
Programmatic Access to Oracle XML DB (Java, PL/SQL, and C) 1-11
Oracle XML DB Features 1-12
XMLType Data Type 1-12
XMLType Tables and Columns Can Conform to an XML Schema 1-13
XMLType API 1-13
XML Schema Support 1-13
XMLType Storage Models 1-14

XML/SQL Duality 1-19
SQL/XML Standard Functions 1-20
Automatic Rewriting of XQuery and XPath Expressions 1-20
How XPath Expressions Are Evaluated by Oracle XML DB 1-20
Rewriting SQL Code That Contains XQuery and XPath Expressions 1-21
When Can XPath Rewrite Occur? 1-21
What is the XPath-Rewrite Process? 1-21
Oracle XML DB Benefits 1-22
Unifying Data and Content 1-23
Exploiting Database Capabilities 1-24
Exploiting XML Capabilities 1-25
Efficient Storage and Retrieval of Complex XML Documents 1-26
Use XMLType Views If Your Data Is Not XML 1-27
Search XML Data using Oracle Text 1-27
Build Messaging Applications using Oracle Streams Advanced Queuing 1-27
Standards Supported by Oracle XML DB 1-28
Oracle XML DB Technical Support 1-29
Oracle XML DB Examples Used in This Manual 1-29
Further Oracle XML DB Case Studies and Demonstrations 1-29

2 Getting Started with Oracle XML DB

Oracle XML DB Installation 2-1
Oracle XML DB Use Cases 2-1
Application Design Considerations for Oracle XML DB 2-2
Structure of Your Data 2-2
Oracle XML DB Repository Access 2-3
Application Language 2-3
Processing 2-4
Messaging 2-5
Storage 2-5
Oracle XML DB Performance 2-6
XML Storage Requirements 2-7
XML Memory Management 2-7
Use of XOBs Reduces Memory Overhead for XML Schema-Based Documents 2-8
XOB Uses a Lazily-Loaded Virtual DOM 2-8
XML Parsing Optimizations 2-8
Node-Searching Optimizations 2-8
XML Schema Optimizations 2-8
Load Balancing Through Cached XML Schema 2-9
Reduced Bottlenecks From Code That Is Not Native 2-9
Reduced Java Type Conversion Bottlenecks 2-9

3 Using Oracle XML DB

Storing XML Data as XMLType 3-1
What is XMLType? 3-2
Benefits of XMLType Data Type and API 3-2
Creating XMLType Tables and Columns 3-3

Partitioning or Constraining Binary XML Data using Virtual Columns 3-3
Loading XML Content into Oracle XML DB 3-5
Loading XML Content using SQL or PL/SQL 3-5
Loading XML Content using Java 3-6
Loading XML Content using C 3-6
Loading Large XML Files That Contain Small XML Documents 3-9
Loading Large XML Files using SQL*Loader 3-9
Loading XML Documents into the Repository using DBMS_XDB 3-9
Loading Documents into the Repository using Protocols 3-10
Character Sets of XML Documents 3-10
XML Encoding Declaration 3-11
Character-Set Determination When Loading XML Documents into the Database 3-11
Character-Set Determination When Retrieving XML Documents from the Database 3-12
Overview of the W3C XML Schema Recommendation 3-13
XML Instance Documents 3-13
XML Schema for Schemas 3-13
Editing XML Schemas 3-14
XML Schema Features 3-14
Text Representation of the Purchase Order XML Schema 3-14
Graphical Representation of the Purchase-Order XML Schema 3-17
Using XML Schema with Oracle XML DB 3-18
Why Use XML Schema with Oracle XML DB? 3-18
Validating Instance Documents with XML Schema 3-18
Constraining Instance Documents for Business Rules or Format Compliance 3-18
Defining How XMLType Contents Must be Stored in the Database 3-18
Structured Storage of XML Documents 3-18
Annotating an XML Schema to Control Naming, Mapping, and Storage 3-19
Controlling How Collections Are Stored for Object-Relational XMLType Storage 3-19
Declaring the Oracle XML DB Namespace 3-20
Registering an XML Schema with Oracle XML DB 3-24
SQL Types and Tables Created During XML Schema Registration 3-25
Working with Large XML Schemas 3-26
Working with Global Elements 3-27
Creating XML Schema-Based XMLType Columns and Tables 3-28
Default Tables 3-29
Identifying XML Schema Instance Documents 3-30
Attributes noNamespaceSchemaLocation and schemaLocation 3-30
Dealing with Multiple Namespaces 3-31
Enforcing XML Data Integrity using the Database 3-31
Comparing Partial to Full XML Schema Validation 3-32
Partial Validation 3-32
Full Validation 3-33
Full XML Schema Validation Costs Processing Time and Memory Usage 3-33
Enforcing Referential Integrity using SQL Constraints 3-34
DML Operations on XML Content using Oracle XML DB 3-37
XPath and Oracle XML 3-38
Querying XML Content Stored in Oracle XML DB 3-38

PurchaseOrder XML Document 3-38
Retrieving the Content of an XML Document using Pseudocolumn OBJECT_VALUE 3-39
Accessing Fragments or Nodes of an XML Document using XMLQUERY 3-40
Accessing Text Nodes and Attribute Values using XMLCAST and XMLQUERY 3-41
Searching an XML Document using XMLEXISTS, XMLCast, and XMLQuery 3-42
Performing SQL Operations on XMLType Fragments using XMLTABLE 3-46
Accessing XML Data in Oracle XML DB using Relational Views 3-48
Breaking Up a Single Level of XML Data 3-49
Breaking Up Multiple Levels of XML Data 3-49
Querying XML Content As Relational Data 3-51
Updating XML Content Stored in Oracle XML DB 3-52
Updating XML Schema-Based and Non-Schema-Based XML Documents 3-57
Namespace Support in Oracle XML DB 3-57
How Oracle XML DB Processes XMLType Methods and SQL Functions 3-58
Generating XML Data from Relational Data 3-59
Generating XML Data from Relational Data using SQL/XML Functions 3-59
Generating XML Data from Relational Data using DBURITYPE 3-63
XSL Transformation and Oracle XML DB 3-64
Using Oracle XML DB Repository 3-68
Installing and Uninstalling Oracle XML DB Repository 3-68
Oracle XML DB Provides Name-Level Locking 3-69
Use Protocols or SQL to Access and Process Repository Content 3-69
Storing and Retrieving Database Content using Standard Protocols 3-70
Uploading Content to Oracle XML DB using FTP 3-70
Accessing Oracle XML DB Repository Programmatically 3-72
Accessing and Updating XML Content in the Repository 3-73
Accessing XML Documents using SQL 3-73
Repository Content is Exposed Through RESOURCE_VIEW and PATH_VIEW 3-73
Use EXISTS_PATH and UNDER_PATH for Path-Based Predicates in a WHERE Clause 3-73
You Can Also Store Non-XML Documents in the Repository 3-74
PL/SQL Packages to Create, Delete, Rename, Move,... Folders and Documents 3-74
Accessing the Content of Documents using SQL 3-74
Accessing the Content of XML Schema-Based Documents 3-76
Accessing Resource Content using Element XMLRef in Joins 3-76
Updating the Content of Documents Stored in the Repository 3-77
Updating Repository Content using Protocols 3-77
Updating Repository Content using SQL 3-78
Updating XML Schema-Based Documents in the Repository 3-80
Controlling Access to Repository Data 3-80
Oracle XML DB Transactional Semantics 3-81
Querying Metadata and the Folder Hierarchy 3-81
RESOURCE_VIEW and PATH_VIEW 3-81
Querying Resources in RESOURCE_VIEW and PATH_VIEW 3-82
Oracle XML DB Hierarchical Repository Index 3-86
How Documents are Stored in the Repository 3-87
Viewing Relational Data as XML From a Browser 3-87
Accessing a Table or View from a Browser using DBUri SERVLET 3-87

XSL Transformation using DBUri Servlet 3-88

Part II Storing and Retrieving XML Data in Oracle XML DB

4 XMLType Operations

Selecting and Querying XML Data 4-1
Searching XML Documents using XPath Expressions 4-1
Querying XMLType Data using SQL/XML Functions XMLExists and XMLCast 4-2
XMLEXISTS SQL/XML Function 4-3
XMLCAST SQL/XML Function 4-4
Examples of Querying XML Data using SQL/XML Functions 4-6
Updating XML Data 4-10
Updating an Entire XML Document 4-11
SQL Functions that Update XML Data 4-12
Inserting XML Elements using SQL Functions 4-13
UPDATEXML SQL Function 4-14
UPDATEXML and NULL Values 4-18
Updating the Same XML Node More Than Once 4-20
Preserving DOM Fidelity When using UPDATEXML 4-20
When DOM Fidelity is Preserved 4-20
When DOM Fidelity is Not Preserved 4-20
Determining Whether DOM Fidelity is Preserved 4-20
Optimization of Oracle SQL Functions that Modify XML Data 4-20
Creating XML Views using Oracle SQL Functions that Modify XML Data 4-22
INSERTCHILDXML SQL Function 4-23
INSERTCHILDXMLBEFORE SQL Function 4-25
INSERTCHILDXMLAFTER SQL Function 4-26
INSERTXMLBEFORE SQL Function 4-27
INSERTXMLAFTER SQL Function 4-29
APPENDCHILDXML SQL Function 4-30
DELETEXML SQL Function 4-32

5 Using XQuery with Oracle XML DB

Overview of XQuery in Oracle XML DB 5-1
Overview of the XQuery Language 5-2
Functional Language Based on Sequences 5-2
XQuery Expressions 5-3
FLWOR Expressions 5-4
SQL/XML Functions XMLQUERY and XMLTABLE 5-5
XMLQUERY SQL/XML Function in Oracle XML DB 5-6
XMLTABLE SQL/XML Function in Oracle XML DB 5-7
When To Use XQuery 5-9
Predefined Namespaces and Prefixes 5-10
URI Scheme oradb: Querying Table or View Data with XQuery 5-10
Oracle XQuery Extension Functions 5-11
ora:contains XQuery Function 5-12

ora:matches XQuery Function 5-12
ora:replace XQuery Function 5-12
ora:sqrt XQuery Function 5-13
ora:tokenize XQuery Function 5-13
Oracle XQuery Extension-Expression Pragmas 5-14
XMLQUERY and XMLTABLE Examples 5-15
XQuery Is About Sequences 5-16
Querying XML Data in Oracle XML DB Repository using XQuery 5-17
Querying Table or View Data using XQuery 5-19
Using XQuery with XMLType Data 5-23
Using Namespaces with XQuery 5-28
Performance Tuning for XQuery 5-30
Rule-Based and Cost-Based XQuery Optimization 5-31
XQuery Optimization over Relational Data 5-31
XQuery Optimization over XML Schema-Based XMLType Data 5-32
Diagnosing XQuery Optimization: XMLOptimizationCheck 5-35
Improving Performance for fn:doc and fn:collection on Repository Data 5-36
Using equals_path and under_path Instead of fn:doc and fn:collection 5-36
Using Oracle XQuery Pragma ora:defaultTable 5-37
XQuery Static Type-Checking in Oracle XML DB 5-37
SQL*Plus XQUERY Command 5-38
Using XQuery with PL/SQL, JDBC, and ODP.NET 5-39
Oracle XML DB Support for XQuery 5-42
Support for XQuery and SQL 5-42
Implementation Choices Specified in the XQuery Standard 5-43
XQuery Features Not Supported by Oracle XML DB 5-43
XQuery Optional Features 5-43
Support for XQuery Functions and Operators 5-43
XQuery Functions fn:doc, fn:collection, and fn:doc-available 5-44

6 Indexing XMLType Data

Oracle XML DB Tasks Involving Indexes 6-1
Overview of Indexing XMLType Data 6-4
XMLIndex Addresses the Fine-Grained Structure of XML Data 6-4
Oracle Text Indexes 6-4
Optimization Chooses the Right Indexes to Use 6-5
Deprecated Indexes for XML Data 6-5
Function-Based Indexes 6-5
CTXXPath Indexes 6-5
Indexing XMLType Data Stored Object-Relationally 6-6
Indexing Non-Repeating text() Nodes or Attribute Values 6-6
Indexing Repeating (Collection) Elements 6-7
XMLIndex 6-7
Advantages of XMLIndex 6-8
Structured and Unstructured XMLIndex Components 6-9
XMLIndex Structured Component 6-10
Ignore the Index Content Tables; They Are Transparent 6-11

Data Type Considerations for XMLIndex Structured Component 6-11
XMLIndex Unstructured Component 6-13
Ignore the Path Table – It Is Transparent 6-15
Column VALUE of an XMLIndex Path Table 6-16
Secondary Indexes on Column VALUE 6-17
XPath Expressions that Are Not Indexed by an XMLIndex Unstructured Component 6-17
Creating, Dropping, Altering, and Examining an XMLIndex Index 6-18
Using XMLIndex with an Unstructured Component 6-19
Creating Additional Secondary Indexes on an XMLIndex Path Table 6-21
Using XMLIndex with a Structured Component 6-23
How to Tell Whether XMLIndex is Used 6-25
Turning Off Use of XMLIndex 6-30
XMLIndex Path Subsetting: Specifying the Paths You Want to Index 6-31
Examples of XMLIndex Path Subsetting 6-31
XMLIndex Path-Subsetting Rules 6-32
Guidelines for Using XMLIndex with an Unstructured Component 6-32
Guidelines for Using XMLIndex with a Structured Component 6-34
Using XMLIndex on Oracle XML DB Repository 6-34
Creating an XMLIndex Index on Repository Resources 6-34
Removing Repository Resources from Indexing with XMLIndex 6-35
Querying Repository Data and Metadata Indexed with XMLIndex 6-35
Dropping an XMLIndex Index Created on Repository Resources 6-36
XMLIndex Partitioning and Parallelism 6-36
Asynchronous (Deferred) Maintenance of XMLIndex Indexes 6-38
Collecting Statistics on XMLIndex Objects for the Cost-Based Optimizer 6-39
Data Dictionary Static Public Views Related to XMLIndex 6-40
PARAMETERS Clause for CREATE INDEX and ALTER INDEX 6-41
Using a Registered PARAMETERS Clause for XMLIndex 6-41
PARAMETERS Clause Syntax for CREATE INDEX and ALTER INDEX 6-41
Usage of XMLIndex_parameters_clause 6-46
Usage of XMLIndex_parameters 6-46
Usage of PATHS Clause 6-46
Usage of create_index_paths_clause and alter_index_paths_clause 6-46
Usage of pikey_clause, path_id_clause, and order_key_clause 6-47
Usage of value_clause 6-47
Usage of async_clause 6-47
Usage of groups_clause and alter_index_group_clause 6-48
Usage of XMLIndex_xmltable_clause 6-48
Usage of column_clause 6-48
Oracle Text Indexes on XML Data 6-48
Creating and Using Oracle Text Indexes 6-48
Oracle Text Indexes Are Used Independently of Other Indexes 6-49

7 XML Schema Storage and Query: Basic

Overview of XML Schema and Oracle XML DB 7-2
Using Oracle XML DB with XML Schema 7-5
Why XML Schema? 7-5

DTD Support in Oracle XML DB 7-6
Inline DTD Definitions 7-6
External DTD Definitions 7-6
Managing XML Schemas with DBMS_XMLSCHEMA 7-6
Registering an XML Schema with Oracle XML DB 7-7
Delete and Reload Documents Before Registering Their XML Schema 7-8
Storage and Access Infrastructure 7-8
Atomic Nature of XML Schema Registration 7-8
Managing and Storing XML Schemas 7-9
Debugging XML Schema Registration for XML Data Stored Object-Relationally 7-9
SQL Object Types Created During XML Schema Registration, for Structured Storage 7-9
Default Tables Created During XML Schema Registration 7-10
Generated Names are Case Sensitive 7-10
Database Objects That Depend on Registered XML Schemas 7-11
Listing All Registered XML Schemas 7-11
Deleting an XML Schema 7-12
DBMS_XMLSCHEMA.DELETESCHEMA Options 7-12
XMLType Methods Related to XML Schema 7-13
Local and Global XML Schemas 7-14
Local XML Schema 7-14
Global XML Schema 7-15
DOM Fidelity 7-16
What is DOM Fidelity? 7-16
SYS_XDBPD$ and DOM Fidelity for Structured Storage 7-16
XML Translations 7-17
Changing an XML Schema and XML Instance Documents for Translation 7-17
Indicating Translatable Elements in an XML Schema 7-17
Indicating Translation Language Attributes in an XML Instance Document 7-17
Making XML Documents Translatable 7-18
Operations on Translated Documents 7-24
Creating XMLType Tables and Columns Based on XML Schemas 7-27
Specifying XMLType Storage Options for XML Schema-Based Data 7-29
Binary XML Storage of XML Schema-Based Data 7-30
Unstructured Storage of XML Schema-Based Data 7-32
Structured Storage of XML Schema-Based Data 7-32
Specifying Relational Constraints on XMLType Tables and Columns 7-34
Oracle XML Schema Annotations 7-34
Common Uses of XML Schema Annotations 7-34
XML Schema Annotation Example 7-35
Available Oracle XML DB XML Schema Annotations 7-39
XML Schema Annotation Guidelines for Structured Storage 7-41
Avoid Creation of Unnecessary Tables for Unused Top-Level Elements 7-42
Provide Your Own Names for Default Tables 7-42
Turn Off DOM Fidelity If Not Needed 7-42
Use Unordered Collection Elements When Order Doesn't Matter 7-42
Annotate Time-Related Elements with a Timestamp Data Type 7-42
Add Table and Column Properties 7-43

Store Large Collections Out of Line 7-43
Querying a Registered XML Schema to Obtain Annotations 7-43
Mapping XML Schema Data Types to Oracle XML DB Storage 7-44
Mapping XML Schema Data Types to SQL Data Types 7-45
Example of Mapping XML Schema Data Types to SQL 7-45
Mapping XML Schema Attribute Data Types to SQL 7-47
Overriding the SQLType Value in an XML Schema When Declaring Attributes 7-47
Mapping XML Schema Element Data Types to SQL 7-47
Overriding the SQLType Value in an XML Schema when Declaring Elements 7-48
Mapping simpleType to SQL 7-48
NCHAR, NVARCHAR, and NCLOB SQLType Values are Not Supported 7-51
simpleType: Mapping XML Strings to SQL VARCHAR2 Versus CLOB 7-51
Working with Time Zones 7-51
Using Trailing Z to Indicate UTC Time Zone 7-52
Mapping complexType to SQL 7-52
Specifying Attributes in a complexType XML Schema Declaration 7-52
Mapping XML Schema Data Types To Binary XML Encoding Types 7-53

8 XPath Rewrite for Structured Storage

Overview of XPath Rewrite for Structured Storage 8-1
Sample of XPath Expressions that Are Rewritten 8-2
Analyzing and Optimizing XPath Queries using Execution Plans 8-3
Guideline: Look for underlying tables versus XML functions in execution plans 8-3
Guideline: Name the default tables, so you recognize them in execution plans 8-4
Guideline: Create an index on a column targeted by a predicate 8-4
Guideline: Create indexes on ordered collection tables 8-6
Guideline: Use XMLOptimizationCheck to determine why a query is not rewritten 8-7

9 XML Schema Storage and Query: Advanced

Generating XML Schemas with DBMS_XMLSCHEMA.GENERATESCHEMA 9-1
Adding Unique Constraints to the Parent Element of an Attribute 9-3
Setting Annotation Attribute SQLInline to false for Out-Of-Line Storage 9-4
XPath Rewrite for Out-Of-Line Tables 9-7
Storing Collections in Out-Of-Line Tables 9-8
Partitioning XMLType Tables and Columns Stored Object-Relationally 9-10
Examples of Partitioning XMLType Data 9-11
Partition Maintenance 9-12
Fully Qualified XML Schema URLs 9-13
Mapping XML Fragments to Large Objects (LOBs) 9-14
complexType Extensions and Restrictions in Oracle XML DB 9-15
complexType Declarations in XML Schema: Handling Inheritance 9-15
Mapping complexType: simpleContent to Object Types 9-17
Mapping complexType: any and anyAttribute 9-18
XML Schema: Working with Circular and Cyclical Dependencies 9-18
For Circular XML Schema Dependencies Set Parameter GENTABLES to TRUE 9-19
complexType Declarations XML Schema: Handling Cycles 9-19

How a complexType Can Reference Itself 9-21
Cyclical References Among XML Schemas 9-22
Support for Recursive Schemas 9-24
Sharing defaultTable Among Common Out-Of-Line Elements 9-25
Query Rewrite when DOCID is Present 9-27
Disabling DOCID Column Creation 9-28
Loading and Retrieving Large Documents with Collections 9-28
Guidelines for Setting xdbcore Parameters 9-29

10 XML Schema Evolution

Overview of XML Schema Evolution 10-1
Using Copy-Based Schema Evolution 10-2
Scenario for Copy-Based Evolution 10-2
copyEvolve Parameters and Errors 10-5
Limitations of Procedure COPYEVOLVE 10-7
Guidelines for Using Procedure COPYEVOLVE 10-8
Top-Level Element Name Changes 10-8
User-Created Virtual Columns of Tables Other Than Default Tables 10-8
Ensure that the XML Schema and Dependents Are Not Used by Concurrent Sessions 10-8
Rollback When Procedure DBMS_XMLSCHEMA.COPYEVOLVE Raises an Error 10-9
Failed Rollback From Insufficient Privileges 10-9
Privileges Needed for XML Schema Evolution 10-9
Updating Existing XML Instance Documents using a Style Sheet 10-10
Examples of Using Procedure COPYEVOLVE 10-12
Using In-Place XML Schema Evolution 10-15
Restrictions for In-Place XML Schema Evolution 10-15
Backward-Compatibility Restrictions 10-15
Changes in Data Layout on Disk 10-16
Reordering of XML Schema Constructs 10-16
Changes from a Collection to a Non-Collection 10-16
Other Restrictions on In-Place Evolution 10-17
Changes to Attributes in Namespace xdb 10-17
Changes from a Non-Collection to a Collection 10-17
Supported Operations for In-Place XML Schema Evolution 10-17
Guidelines for Using In-Place XML Schema Evolution 10-18
inPlaceEvolve Parameters 10-19
Creating the Document for the diffXML Parameter 10-20
diffXML Operations and Examples 10-21

11 Transforming and Validating XMLType Data

Transforming XMLType Instances 11-1
SQL Function XMLTRANSFORM and XMLType Method TRANSFORM() 11-2
XMLTRANSFORM and XMLType.transform(): Examples 11-2
Validating XMLType Instances 11-7
Validating XML Data Stored as XMLType: Examples 11-8

12 Full-Text Search Over XML Data

Overview of Full-Text Search for XML 12-1
Comparison of Full-Text Search and Other Search Types 12-1
Searching XML Data 12-2
Searching Documents using Full-Text Search and XML Structure 12-2
About the Full-Text Search Examples 12-2
Roles and Privileges 12-2
Schema and Data for Full-Text Search Examples 12-2
Overview of CONTAINS and ora:contains 12-3
Overview of SQL Function CONTAINS 12-3
Overview of XPath Function ora:contains 12-4
Comparison of CONTAINS and ora:contains 12-4
CONTAINS SQL Function 12-5
Full-Text Search using SQL Function CONTAINS 12-5
Full-Text Boolean Operators AND, OR, and NOT 12-6
Full-Text Stemming: $ 12-6
Combining Boolean and Stemming Operators 12-6
SCORE SQL Function 12-6
Restricting the Scope of a CONTAINS Search 12-7
WITHIN Structure Operator 12-7
Nested WITHIN 12-7
WITHIN Attributes 12-8
WITHIN and AND 12-8
Definition of Section 12-9
INPATH Structure Operator 12-9
Text Path 12-9
Text Path Compared to XPath 12-10
Nested INPATH 12-10
HASPATH Structure Operator 12-11
Projecting the CONTAINS Result 12-12
Indexing with a CONTEXT Index 12-12
Introduction to CONTEXT Indexes 12-12
CONTEXT Index on XMLType Table 12-13
Maintaining a CONTEXT Index 12-13
Roles and Privileges 12-14
Effect of a CONTEXT Index on CONTAINS 12-14
CONTEXT Index Preferences 12-14
Making Search Case-Sensitive 12-14
Introduction to Section Groups 12-15
Choosing a Section Group Type 12-15
Choosing a Section Group 12-16
ora:contains XQuery Function 12-17
Full-Text Search using XQuery Function ora:contains 12-17
Restricting the Scope of an ora:contains Query 12-17
Projecting the ora:contains Result 12-18
Policies for ora:contains Queries 12-18
Introduction to Policies for ora:contains Queries 12-18

Policy Example: Supplied Stoplist 12-19
Effect of Policies on ora:contains 12-19
Policy Example: User-Defined Lexer 12-20
Policy Defaults 12-21
Performance of ora:contains 12-22
Use a Primary Filter in the Query 12-22
XPath Rewrite and CONTEXT Indexes 12-23
Benefits of XPath Rewrite 12-23
From Documents to Nodes 12-24
From ora:contains to contains 12-24
Summary of Using XPath Rewrite with ora:contains 12-25
Text Path BNF Specification 12-25
Support for Full-Text XML Examples 12-26
Purchase-Order XML Document, po001.xml 12-26
CREATE TABLE Statements 12-27
Purchase-Order XML Schema for Full-Text Search Examples 12-29

Part III Using XMLType APIs

13 PL/SQL APIs for XMLType

Overview of PL/SQL APIs for XMLType 13-1
API Features 13-1
Lazy Loading of XML Data (Lazy Manifestation) 13-2
XMLType Data Type Supports XML Schema 13-2
XMLType Supports Data in Different Character Sets 13-2
PL/SQL DOM API for XMLType (DBMS_XMLDOM) 13-3
Overview of the W3C Document Object Model (DOM) Recommendation 13-3
Oracle XDK Extensions to the W3C DOM Standard 13-3
Supported W3C DOM Recommendations 13-3
Difference Between DOM and SAX 13-4
PL/SQL DOM API for XMLType (DBMS_XMLDOM): Features 13-4
XML Schema Support 13-4
Enhanced Performance 13-5
Designing End-to-End Applications using Oracle XDK and Oracle XML DB 13-5
Preparing XML Data to Use the PL/SQL DOM API for XMLType 13-6
Defining an XML Schema Mapping to SQL Object Types 13-6
DOM Fidelity for XML Schema Mapping 13-7
Wrapping Existing Data into XML with XMLType Views 13-7
DBMS_XMLDOM Methods Supported 13-7
PL/SQL DOM API for XMLType: Node Types 13-8
Working with XML Schema-Based Data 13-9
DOM NodeList and NamedNodeMap Objects 13-9
Using the PL/SQL DOM API for XMLType (DBMS_XMLDOM) 13-10
PL/SQL DOM API for XMLType – Examples 13-10
Large Node Handling using DBMS_XMLDOM 13-12
Get-Push Model 13-14
Get-Pull Model 13-15

Set-Pull Model 13-16
Set-Push Model 13-17
Determining Binary Stream or Character Stream 13-19
PL/SQL Parser API for XMLType (DBMS_XMLPARSER) 13-19
Features of the PL/SQL Parser API for XMLType 13-19
Using the PL/SQL Parser API for XMLType (DBMS_XMLPARSER) 13-19
PL/SQL XSLT Processor for XMLType (DBMS_XSLPROCESSOR) 13-20
Enabling Transformations and Conversions with XSLT 13-21
PL/SQL XSLT Processor for XMLType: Features 13-21
Using the PL/SQL XSLT Processor API for XMLType (DBMS_XSLPROCESSOR) 13-21
PL/SQL Translation API for XMLType (DBMS_XMLTRANSLATIONS) 13-23
DBMS_XMLTRANSLATIONS Methods 13-24

14 PL/SQL Package DBMS_XMLSTORE

Overview of PL/SQL Package DBMS_XMLSTORE 14-1
Using Package DBMS_XMLSTORE 14-1
Inserting with DBMS_XMLSTORE 14-2
Updating with DBMS_XMLSTORE 14-4
Deleting with DBMS_XMLSTORE 14-5

15 Java DOM API for XMLType

Overview of Java DOM API for XMLType 15-1
Java DOM API for XMLType 15-1
Accessing XMLType Data using JDBC 15-2
Using XMLType Data with JDBC 15-2
How Java Applications Use JDBC to Access XML Documents in Oracle XML DB 15-2
Manipulating XML Database Documents using JDBC 15-4
Loading a Large XML Document into the Database using JDBC 15-11
Java DOM API for XMLType Features 15-13
Creating XML Schema-Based Documents 15-13
JDBC or SQLJ 15-14
Java DOM API for XMLType Classes 15-14
Java Methods That Are Deprecated or Not Supported 15-15
Using the Java DOM API for XMLType 15-16
Handling Large Nodes using Java 15-16
Stream Extensions to Java DOM 15-17
Get-Pull Model 15-17
Get-Push Model 15-18
Set-Pull Model 15-18
Set-Push Model 15-19
Using the Java DOM API and JDBC with Binary XML 15-20

16 Using the C API for XML

Overview of the C API for XML (Oracle XDK and Oracle XML DB) 16-1
Using OCI and the C API for XML with Oracle XML DB 16-2
Accessing XMLType Data Stored in the Database 16-2

Creating XMLType Instances on the Client 16-2
XML Context Parameter for C DOM API Functions 16-2
OCIXmlDbInitXmlCtx() Syntax 16-2
OCIXmlDbFreeXmlCtx() Syntax 16-3
Initializing and Terminating an XML Context 16-3
Using the C API for XML with Binary XML 16-6
Using the Oracle XDK Pull Parser with Oracle XML DB 16-9
Common XMLType Operations in C 16-14

17 Using Oracle Data Provider for .NET with Oracle XML DB

ODP.NET XML Support and Oracle XML DB 17-1
ODP.NET Sample Code 17-1

Part IV Viewing Existing Data as XML

18 Generating XML Data from the Database

Overview of Generating XML Data From Oracle Database 18-1
Overview of Generating XML using SQL/XML Functions 18-1
Overview of Generating XML using Oracle SQL Functions 18-1
Overview of Generating XML using DBMS_XMLGEN 18-2
Overview of Generating XML using XSQL Pages Publishing Framework 18-2
Overview of Generating XML using XML SQL Utility (XSU) 18-2
Overview of Generating XML using DBURIType 18-2
Generating XML using SQL Functions 18-2
XMLELEMENT and XMLATTRIBUTES SQL/XML Functions 18-3
Escaping Characters in Generated XML Data 18-5
Formatting of XML Dates and Timestamps 18-6
XMLElement Examples 18-6
XMLFOREST SQL/XML Function 18-10
XMLCONCAT SQL/XML Function 18-11
XMLAGG SQL/XML Function 18-12
XMLPI SQL/XML Function 18-15
XMLCOMMENT SQL/XML Function 18-16
XMLSERIALIZE SQL/XML Function 18-17
XMLPARSE SQL/XML Function 18-18
XMLROOT SQL Function 18-19
XMLCOLATTVAL SQL Function 18-20
XMLCDATA SQL Function 18-21
Generating XML using DBMS_XMLGEN 18-22
Using PL/SQL Package DBMS_XMLGEN 18-22
Functions and Procedures of Package DBMS_XMLGEN 18-24
DBMS_XMLGEN Examples 18-28
Generating XML using SQL Function SYS_XMLGEN 18-46
Advantages of using Oracle SQL Function SYS_XMLGEN 18-47
Using XMLFormat Object Type 18-48
Generating XML using SQL Function SYS_XMLAGG 18-54

Generating XML using XSQL Pages Publishing Framework 18-54
Using XSLT and XSQL 18-57
Generating XML using XML SQL Utility (XSU) 18-57
Guidelines for Generating XML with Oracle XML DB 18-57
Ordering Query Results Before Aggregating, using XMLAGG ORDER BY Clause 18-57
Returning a Rowset using XMLTABLE 18-58

19 XMLType Views

What Are XMLType Views? 19-1
Creating XMLType Views: Syntax 19-2
Creating Non-Schema-Based XMLType Views 19-2
Creating Non-Schema-Based XMLType Views using SQL/XML Publishing Functions 19-3
Creating Non-Schema-Based XMLType Views using Object Types and SYS_XMLGEN 19-3
Creating XML Schema-Based XMLType Views 19-4
Creating XML Schema-Based XMLType Views using SQL/XML Publishing Functions 19-4
Using Namespaces with SQL/XML Publishing Functions 19-7
Creating XML Schema-Based XMLType Views using Object Types or Object Views 19-10
Creating XMLType Employee View, with Nested Department Information 19-11
Step 1. Create Object Types 19-11
Step 2. Create and Register XML Schema emp_complex.xsd 19-12
Step 3a. Create XMLType View emp_xml using Object Type emp_t 19-13
Step 3b. Create XMLType View emp_xml using Object View emp_v 19-14
Creating XMLType Department View, with Nested Employee Information 19-14
Step 1. Create Object Types 19-14
Step 2. Register XML Schema dept_complex.xsd 19-14
Step 3a. Create XMLType View dept_xml using Object Type dept_t 19-16
Step 3b. Create XMLType View dept_xml using Relational Data Directly 19-16
Creating XMLType Views from XMLType Tables 19-17
Referencing XMLType View Objects using SQL Function REF 19-17
DML (Data Manipulation Language) on XMLType Views 19-18

20 Accessing Data Through URIs

Overview of Oracle XML DB URL Features 20-1
URIs and URLs 20-1
URIType and its Subtypes 20-2
DBUris and XDBUris – What For? 20-3
URIType Methods 20-4
HTTPURIType PL/SQL Method GETCONTENTTYPE() 20-5
DBURIType PL/SQL Method GETCONTENTTYPE() 20-5
DBURIType PL/SQL Method GETCLOB() 20-6
DBURIType PL/SQL Method GETBLOB() 20-6
Accessing Data using URIType Instances 20-6
XDBUris: Pointers to Repository Resources 20-10
XDBUri URI Syntax 20-10
XDBUri Examples 20-10
DBUris: Pointers to Database Data 20-12

Viewing the Database as XML Data 20-12
DBUri URI Syntax 20-14
DBUris are Scoped to a Database and Session 20-15
DBUri Examples 20-16
Targeting a Table 20-16
Targeting a Row in a Table 20-17
Targeting a Column 20-17
Retrieving the Text Value of a Column 20-18
Targeting a Collection 20-19
Creating New Subtypes of URIType using Package URIFACTORY 20-20
Registering New URIType Subtypes with Package URIFACTORY 20-20
SYS_DBURIGEN SQL Function 20-22
Rules for Passing Columns or Object Attributes to SYS_DBURIGEN 20-23
SYS_DBURIGEN SQL Function: Examples 20-24
Returning Partial Results 20-24
RETURNING URLs to Inserted Objects 20-25
DBUriServlet 20-26
Overriding the MIME Type using a URL 20-27
Customizing DBUriServlet 20-27
DBUriServlet Security 20-28
Configuring Package URIFACTORY to Handle DBUris 20-29

Part V Oracle XML DB Repository

21 Accessing Oracle XML DB Repository Data

Overview of Oracle XML DB Repository 21-1
Two Ways to Access Oracle XML DB Repository Resources 21-3
Repository Terminology and Supplied Resources 21-3
Repository Terminology 21-3
Supplied Files and Folders 21-4
Oracle XML DB Repository Resources 21-5
Where Is Repository Data Stored? 21-5
Names of Generated Tables 21-5
Defining Structured Storage for Resources 21-6
Oracle ASM Virtual Folder 21-6
Path-Name Resolution 21-6
Link Types 21-7
Repository and Document Links 21-7
Hard Links and Weak Links 21-7
Creating a Weak Link with No Knowledge of Folder Hierarchy 21-8
Restricting Multiple Hard Links 21-9
Navigational or Path Access to Repository Resources 21-9
Accessing Oracle XML DB Resources using Internet Protocols 21-10
Where You Can Use Oracle XML DB Protocol Access 21-11
Using Protocol Access 21-11
Retrieving Oracle XML DB Resources 21-11
Storing Oracle XML DB Resources 21-11

Using Internet Protocols and XMLType: XMLType Direct Stream Write 21-12
Accessing Oracle ASM Files using Protocols and Resource APIs – For DBAs 21-12
Query-Based Access to Repository Resources 21-13
Servlet Access to Repository Resources 21-14
Operations on Repository Resources 21-14

22 Configuring Oracle XML DB Repository

Resource Configuration Files Configure a Resource 22-1
Configuring a Resource 22-2
Common Configuration Parameters 22-3
Configuration Element ResConfig 22-3
Configuration Element defaultChildConfig 22-3
Configuration Element applicationData 22-4

23 Using XLink and XInclude with Oracle XML DB

Overview of XLink and XInclude 23-1
XLink and XInclude Link Types 23-2
XLink and XInclude Links Model Document Relationships 23-2
XLink and XInclude Link Types 23-2
XInclude: Compound Documents 23-3
Using XLink with Oracle XML DB 23-4
Using XInclude with Oracle XML DB 23-4
Expanding Compound-Document Inclusions 23-5
Validating Compound Documents 23-6
Updating Compound Documents 23-6
Versioning, Locking, and Controlling Access to Compound Documents 23-6
Examining XLink and XInclude Links using DOCUMENT_LINKS View 23-7
Querying DOCUMENT_LINKS for XLink Information 23-7
Querying DOCUMENT_LINKS for XInclude Information 23-8
Configuring Resources for XLink and XInclude 23-9
Configuring Treatment of Unresolved Links: UnresolvedLink Attribute 23-9
Configuring the Document Links to Create: LinkType Element 23-10
Configuring the Path Format for Retrieval: PathFormat Element 23-10
Configuring Conflict-Resolution for XInclude: ConflictRule Element 23-11
Configuring Decomposition of Documents using XInclude: SectionConfig Element 23-11
XLink and XInclude Configuration Examples 23-12
Managing XLink and XInclude Links using DBMS_XDB.processLinks 23-13

24 Managing Resource Versions

Overview of Oracle XML DB Versioning 24-1
Versioning and Resource IDs 24-2
Versioning and ACLs 24-4
Resource Versioning Examples 24-4

25 Accessing the Repository using RESOURCE_VIEW and PATH_VIEW

Overview of Oracle XML DB RESOURCE_VIEW and PATH_VIEW 25-1
RESOURCE_VIEW Definition and Structure 25-2
PATH_VIEW Definition and Structure 25-3
Understanding the Difference Between RESOURCE_VIEW and PATH_VIEW 25-4
Operations You Can Perform using UNDER_PATH and EQUALS_PATH 25-5
RESOURCE_VIEW and PATH_VIEW SQL Functions 25-5
UNDER_PATH SQL Function 25-5
EQUALS_PATH SQL Function 25-7
PATH SQL Function 25-7
DEPTH SQL Function 25-8
Using RESOURCE_VIEW and PATH_VIEW SQL Functions 25-8
Accessing Repository Data Paths, Resources and Links: Examples 25-8
Deleting Repository Resources: Examples 25-14
Deleting Nonempty Folder Resources 25-15
Updating Repository Resources: Examples 25-15
Working with Multiple Oracle XML DB Resources 25-18
Performance Tuning of Oracle XML DB Repository Operations 25-19
Searching for Resources using Oracle Text 25-20

26 Accessing the Repository using PL/SQL

Overview of PL/SQL Package DBMS_XDB 26-1
DBMS_XDB: Resource Management 26-1
DBMS_XDB: ACL-Based Security Management 26-3
DBMS_XDB: Configuration Management 26-7

27 Repository Access Control

Access Control Concepts 27-1
Principal: A User or Role 27-2
Database Roles Map Database Privileges to Users 27-2
Principal DAV::owner 27-2
Privilege: A Permission 27-3
Access Control Entry (ACE) 27-3
Access Control List (ACL) 27-4
Database Privileges for Repository Operations 27-4
Privileges 27-5
Atomic Privileges 27-5
Aggregate Privileges 27-6
ACLs and ACEs 27-6
System ACLs 27-7
ACL and ACE Evaluation 27-8
ACL Validation 27-8
ACL Inheritance 27-9
Complementing the Principals in an ACE: Element invert 27-10
ACE Validity Time Period 27-10
Working with Access Control Lists (ACLs) 27-11

Creating an ACL using DBMS_XDB.CREATERESOURCE 27-11
Retrieving an ACL Document, Given its Repository Path 27-12
Setting the ACL of a Resource 27-12
Deleting an ACL 27-12
Updating an ACL 27-13
Retrieving the ACL Document that Protects a Given Resource 27-14
Retrieving Privileges Granted to the Current User for a Particular Resource 27-15
Checking Whether the Current User Has Privileges on a Resource 27-15
Checking Whether a User Has Privileges using the ACL and Resource Owner 27-16
Retrieving the Path of the ACL that Protects a Given Resource 27-17
Retrieving the Paths of All Resources Protected by a Given ACL 27-17
ACL Caching 27-18
Repository Resources and Database Table Security 27-18
Optimization: Do not enforce acl-based security if you do not need it 27-19
Integrating Oracle XML DB with LDAP 27-20

28 Accessing the Repository using Protocols

Overview of Oracle XML DB Protocol Server 28-1
Session Pooling 28-2
Oracle XML DB Protocol Server Configuration Management 28-3
Configuring Protocol Server Parameters 28-3
Configuring Secure HTTP (HTTPS) 28-6
Enable the HTTP Listener to Use SSL 28-7
Enable TCPS Dispatcher 28-7
Interaction with Oracle XML DB File-System Resources 28-7
Protocol Server Handles XML Schema-Based or Non-Schema-Based XML Documents 28-8
Event-Based Logging 28-8
Using FTP and Oracle XML DB Protocol Server 28-8
Oracle XML DB Protocol Server: FTP Features 28-8
FTP Features That Are Not Supported 28-9
Supported FTP Client Methods 28-9
FTP Quote Methods 28-10
Using FTP with Oracle ASM Files 28-11
Using FTP on the Standard Port Instead of the Oracle XML DB Default Port 28-12
Using IPv6 IP Addresses with FTP 28-13
FTP Server Session Management 28-14
Handling Error 421. Modifying the Default Timeout Value of an FTP Session 28-14
FTP Client Failure in Passive Mode 28-14
Using HTTP(S) and Oracle XML DB Protocol Server 28-14
Oracle XML DB Protocol Server: HTTP(S) Features 28-15
HTTP(S) Features That Are Not Supported 28-15
Supported HTTP(S) Client Methods 28-15
Using HTTP(S) on a Standard Port Instead of an Oracle XML DB Default Port 28-15
Using IPv6 IP Addresses with HTTP(S) 28-16
HTTPS: Support for Secure HTTP 28-16
Controlling URL Expiration Time 28-17
Anonymous Access to Oracle XML DB Repository using HTTP 28-17

Using Java Servlets with HTTP(S) 28-18
Embedded PL/SQL Gateway 28-18
Sending Multibyte Data From a Client 28-19
Characters That Are Not ASCII in URLs 28-19
Controlling Character Sets for HTTP(S) 28-20
Request Character Set 28-20
Response Character Set 28-20
Using WebDAV and Oracle XML DB 28-20
Oracle XML DB WebDAV Features 28-21
WebDAV Features That Are Not Supported 28-21
Supported WebDAV Client Methods 28-21
Using WebDAV with Microsoft Windows XP SP2 28-21
Creating a WebFolder in Microsoft Windows using Oracle XML DB and WebDAV 28-22

29 User-Defined Repository Metadata

Overview of Metadata and XML 29-1
Kinds of Metadata – Uses of the Term 29-2
User-Defined Resource Metadata 29-2
Scenario: Metadata for a Photo Collection 29-3
XML Schemas to Define Resource Metadata 29-3
Adding, Updating, and Deleting Resource Metadata 29-4
Adding Metadata using APPENDRESOURCEMETADATA 29-5
Deleting Metadata using DELETERESOURCEMETADATA 29-6
Adding Metadata using SQL DML 29-7
Adding Metadata using WebDAV PROPPATCH 29-8
Querying XML Schema-Based Resource Metadata 29-9
XML Image Metadata from Binary Image Metadata 29-10
Adding Non-Schema-Based Resource Metadata 29-11
PL/SQL Procedures Affecting Resource Metadata 29-12

30 Oracle XML DB Repository Events

Overview of Repository Events 30-1
Repository Events: Use Cases 30-1
Repository Events and Database Triggers 30-2
Repository Event Listeners and Event Handlers 30-2
Repository Event Configuration 30-2
Possible Repository Events 30-3
Repository Operations and Events 30-5
Repository Event Handler Considerations 30-6
Configuring Repository Events 30-8
Configuration Element event-listeners 30-8
Configuration Element listener 30-9
Repository Events Configuration Examples 30-9

31 Using Oracle XML DB Content Connector

Overview of JCR and Oracle XML DB Content Connector 31-1

About the Content Repository API for Java (JCR) ... 31-1
About Oracle XML DB Content Connector ... 31-2
How Oracle XML DB Repository Is Exposed in JCR ... 31-2
Example of How Files and Folders are Exposed in JCR ... 31-3
Oracle Extensions to JCR Node Types ... 31-5
Binary and XML Content ... 31-5
System-Defined Metadata ... 31-5
User-Defined Metadata ... 31-6
Hard Links and Weak Links ... 31-6
How to Use Oracle XML DB Content Connector ... 31-7
Setting CLASSPATH ... 31-7
Obtaining the JCR Repository Object ... 31-7
Sample Code to Upload File ... 31-8
Additional Code Samples ... 31-9
Logging API for Oracle XML DB Content Connector ... 31-9
Supported JCR Compliance Levels ... 31-10
Oracle XML DB Content Connector Restrictions ... 31-10
Default Workspace Name ... 31-10
Operations Restricted to Specific Node Types ... 31-10
Determining the State of Files or Folders ... 31-10
Interaction Between Binary and XML Content ... 31-10
Order in Which Changes Are Saved ... 31-10
Undefined Properties ... 31-10
Node Type nt:base Is Abstract ... 31-11
Node jcr:content Is Created Automatically ... 31-11
Saving Normalizes Node jcr:xmltext ... 31-11
Node Type mix:referenceable ... 31-11
Full-Text Indexing ... 31-11
Using XML Schemas with JCR ... 31-11
Why Register XML Schemas for Use with JCR? ... 31-11
How to Register an XML Schema with JCR ... 31-13
How JCR Node Types are Generated from XML Schemas ... 31-14
Built-In Simple Types ... 31-14
XML Schema-Defined Simple Types ... 31-16
Complex Types ... 31-17
Global Element Declarations ... 31-17

32 Writing Oracle XML DB Applications in Java

Overview of Oracle XML DB Java Applications ... 32-1
Which Oracle XML DB APIs Are Available Inside and Outside the Database? ... 32-1
Design Guidelines: Java Inside or Outside the Database? ... 32-2
HTTP(S): Accessing Java Servlets or Directly Accessing XMLType Resources ... 32-2
Accessing Many XMLType Object Elements: Use JDBC XMLType Support ... 32-2
Use the Servlets to Manipulate and Write Out Data Quickly as XML ... 32-2
Writing Oracle XML DB HTTP Servlets in Java ... 32-2
Configuring Oracle XML DB Servlets ... 32-3
HTTP Request Processing for Oracle XML DB Servlets ... 32-6

Session Pool and Oracle XML DB Servlets 32-7
Native XML Stream Support 32-7
Oracle XML DB Servlet APIs 32-7
Oracle XML DB Servlet Example 32-7

33 Using Native Oracle XML DB Web Services

Overview of Native Oracle XML DB Web Services 33-1
Configuring and Enabling Web Services for Oracle XML DB 33-2
Configuring Web Services for Oracle XML DB 33-2
Enabling Web Services for Specific Users 33-3
Querying Oracle XML DB using a Web Service 33-3
Accessing PL/SQL Stored Procedures using a Web Service 33-6
Example of Using a PL/SQL Function with a Web Service 33-7

Part VI Oracle Tools that Support Oracle XML DB

34 Administering Oracle XML DB

Installing Oracle XML DB 34-1
Installing Oracle XML DB with Database Configuration Assistant 34-1
Dynamic Protocol Registration of FTP and HTTP(S) Services with Local Listener 34-2
Changing FTP or HTTP(S) Port Numbers 34-2
Post-installation 34-2
Installing Oracle XML DB Manually without DBCA 34-2
Post-Installation 34-3
Upgrading an Existing Oracle XML DB Installation 34-3
Validation of ACL Documents and Configuration File 34-4
Administering Oracle XML DB using Oracle Enterprise Manager 34-4
Configuring Oracle XML DB using xdbconfig.xml 34-5
Oracle XML DB Configuration File, xdbconfig.xml 34-5
<xdbconfig> (Top-Level Element) 34-5
<sysconfig> (Child of <xdbconfig>) 34-5
<userconfig> (Child of <xdbconfig>) 34-6
<protocolconfig> (Child of <sysconfig>) 34-6
<httpconfig> (Child of <protocolconfig>) 34-6
<servlet> (Descendant of <httpconfig>) 34-7
Oracle XML DB Configuration File Example 34-7
Oracle XML DB Configuration API 34-10
Configuring Default Namespace to Schema Location Mappings 34-10
Configuring XML File Extensions 34-12

35 Loading XML Data using SQL*Loader

Overview of Loading XMLType Data Into Oracle Database 35-1
Loading XMLType Data using SQL*Loader 35-1
Loading XMLType Data in LOBs using SQL*Loader 35-2
Loading LOB Data in Predetermined Size Fields 35-2
Loading LOB Data in Delimited Fields 35-2

Loading XML Columns Containing LOB Data from LOBFILEs 35-3
Specifying LOBFILEs 35-3
Loading XMLType Data Directly from a Control File using SQL*Loader 35-3
Loading Large XML Documents into Oracle Database 35-3

36 Exporting and Importing XMLType Tables

Overview of Oracle Data Pump 36-1
EXPORT/IMPORT Support in Oracle XML DB 36-2
Exporting XML Schema-Based XMLType Tables 36-2
Exporting Hierarchy-Enabled (Repository) Tables 36-3
Exporting and Importing Transportable Tablespaces 36-3
Repository Resources and Foldering Support 36-3
Full Database Export 36-3
Exporting and Importing with Different Character Sets 36-4
Export/Import Syntax and Examples 36-4
Performing a Table-Mode Export /Import 36-4
Performing a Schema-Mode Export/Import 36-5

37 Exchanging XML Data using Oracle Streams AQ

How Do AQ and XML Complement Each Other? 37-1
AQ and XML Message Payloads 37-1
Advantages of Using AQ 37-3
Oracle Streams and AQ 37-3
Streams Message Queuing 37-4
XMLType Attributes in Object Types 37-4
Internet Data Access Presentation (iDAP) 37-5
iDAP Architecture 37-5
XMLType Queue Payloads 37-5
Guidelines for Using XML and Oracle Streams Advanced Queuing 37-8
Storing Oracle Streams AQ XML Messages with Many PDFs as One Record? 37-8
Adding New Recipients After Messages Are Enqueued 37-8
Enqueuing and Dequeuing XML Messages? 37-8
Parsing Messages with XML Content from Oracle Streams AQ Queues 37-9
Preventing the Listener from Stopping Until the XML Document Is Processed 37-9
Using HTTPS with AQ 37-9
Storing XML in Oracle Streams AQ Message Payloads 37-9
Comparing iDAP and SOAP 37-9

Part VII Appendixes

A Oracle-Supplied XML Schemas and Examples

XDBResource.xsd: XML Schema for Oracle XML DB Resources A-1
XDBResource.xsd A-1
XDBResConfig.xsd: XML Schema for Resource Configuration A-9
XDBResConfig.xsd A-9
acl.xsd: XML Schema for ACLs A-13

acl.xsd A-13
xdbconfig.xsd: XML Schema for Configuring Oracle XML DB A-16
xdbconfig.xsd A-16
xdiff.xsd: XML Schema for Comparing Schemas for In-Place Evolution A-28
xdiff.xsd A-28
Purchase-Order XML Schemas A-30
XSL Style Sheet Example, PurchaseOrder.xsl A-38
Loading XML Data using C (OCI) A-43
Initializing and Terminating an XML Context (OCI) A-47

B Oracle XML DB Restrictions

Index

List of Examples

1–1 Listener Status with FTP and HTTP(S) Protocol Support Enabled 1-11
3–1 Creating a Table with an XMLType Column..... 3-3
3–2 Creating a Table of XMLType 3-3
3–3 Partitioning a Binary XML Table using Virtual Columns 3-4
3–4 Creating a Database Directory 3-6
3–5 Inserting XML Content into an XMLType Table..... 3-6
3–6 Inserting Content into an XMLType Table using Java 3-6
3–7 Inserting Content into an XMLType Table using C..... 3-7
3–8 Inserting XML Content into the Repository using CREATERESOURCE 3-9
3–9 Purchase-Order XML Schema, purchaseOrder.xsd 3-14
3–10 Annotated Purchase-Order XML Schema, purchaseOrder.xsd 3-20
3–11 Registering an XML Schema using DBMS_XMLSCHEMA.REGISTERSCHEMA 3-25
3–12 Objects Created During XML Schema Registration..... 3-25
3–13 Creating an XMLType Table that Conforms to an XML Schema 3-28
3–14 Creating an XMLType Table for Nested Collections..... 3-29
3–15 Using DESCRIBE with an XML Schema-Based XMLType Table 3-29
3–16 Error From Attempting to Insert an Incorrect XML Document..... 3-32
3–17 Error When Inserting Incorrect XML Document (Partial Validation) 3-32
3–18 Forcing Full XML Schema Validation using a CHECK Constraint..... 3-33
3–19 Enforcing Full XML Schema Validation using a BEFORE INSERT Trigger 3-34
3–20 Constraining a Binary XML Table using a Virtual Column 3-35
3–21 Integrity Constraints and Triggers for an XMLType Table Stored Object-Relationally 3-35
3–22 Enforcing Database Integrity When Loading XML using FTP 3-36
3–23 PurchaseOrder XML Instance Document..... 3-38
3–24 Retrieving an Entire XML Document using OBJECT_VALUE..... 3-39
3–25 Accessing XML Fragments using XMLQUERY 3-40
3–26 Accessing a Text Node Value using XMLCAST and XMLQuery..... 3-41
3–27 Searching XML Content using XMLExists, XMLCast, and XMLQuery 3-42
3–28 Finding the Reference for a Purchase Order using XMLQuery and XMLExists..... 3-45
3–29 Accessing Description Nodes using XMLTABLE..... 3-46
3–30 Counting the Number of Elements in a Collection using XMLTABLE 3-47
3–31 Counting the Number of Child Elements in an Element using XMLTABLE 3-48
3–32 Creating a Relational View of XML Content 3-49
3–33 Accessing Individual Members of a Collection using a View..... 3-50
3–34 Querying XML Data using Views 3-51
3–35 Business-Intelligence Query of XML Data using a View 3-52
3–36 Updating XML Content using UPDATEXML 3-53
3–37 Replacing an Entire Element using UPDATEXML..... 3-53
3–38 Incorrectly Updating a Node That Occurs Multiple Times in a Collection 3-54
3–39 Correctly Updating a Node That Occurs Multiple Times in a Collection 3-55
3–40 Changing Text Node Values using UPDATEXML 3-56
3–41 Generating XML Data using SQL/XML Functions 3-59
3–42 Creating XMLType Views Over Conventional Relational Tables 3-60
3–43 Querying XMLType Views..... 3-61
3–44 Generating XML Data from a Relational Table using DBURIType and getXML()..... 3-63
3–45 Restricting Rows using an XPath Predicate 3-64
3–46 Restricting Rows and Columns using an XPath Predicate 3-64
3–47 XSLT Style Sheet Example: PurchaseOrder.xsl 3-65
3–48 Applying a Style Sheet using TRANSFORM..... 3-67
3–49 Uploading Content to the Repository using FTP 3-70
3–50 Creating a Text Document Resource using CREATERESOURCE 3-72
3–51 Creating Folders using PL/SQL Package DBMS_XDB..... 3-74
3–52 Accessing a Text Document in the Repository using XDBURITYPE..... 3-75
3–53 Accessing Resource Content using RESOURCE_VIEW..... 3-75

3–54 Accessing XML Documents using Resource and Namespace Prefixes 3-75
3–55 Querying Repository Resource Data using SQL Function REF and Element XMLRef. 3-76
3–56 Selecting XML Document Fragments Based on Metadata, Path, and Content..... 3-77
3–57 Updating a Document using UPDATE and UPDATEXML on the Resource 3-78
3–58 Updating a Node using UPDATE and UPDATEXML 3-79
3–59 Updating XML Schema-Based Documents in the Repository 3-80
3–60 Viewing RESOURCE_VIEW and PATH_VIEW Structures..... 3-82
3–61 Accessing Resources using EQUALS_PATH and RESOURCE_VIEW..... 3-83
3–62 Determining the Path to XSL Style Sheets Stored in the Repository..... 3-84
3–63 Counting Resources Under a Path 3-84
3–64 Listing the Folder Contents in a Path..... 3-84
3–65 Listing the Links Contained in a Folder 3-85
3–66 Finding Paths to Resources that Contain Purchase-Order XML Documents 3-85
3–67 Execution Plan Output for a Folder-Restricted Query 3-86
4–1 Finding a Node using SQL/XML Function XMLExists 4-4
4–2 Extracting the Scalar Value of an XML Fragment using XMLCAST..... 4-5
4–3 Querying XMLTYPE Data 4-6
4–4 Querying Transient XMLTYPE Data using a PL/SQL Cursor 4-7
4–5 Extracting XML Data using XMLTABLE, and Inserting It into a Database Table 4-7
4–6 Extracting XML Data and Inserting It into a Table using a PL/SQL Procedure 4-9
4–7 Searching XML Data using SQL/XML Functions..... 4-9
4–8 Extracting Fragments from an XMLTYPE Instance using XMLQUERY 4-10
4–9 Updating XMLType Data using a SQL UPDATE Statement 4-11
4–10 Updating XMLTYPE using UPDATE and UPDATEXML..... 4-15
4–11 Updating Multiple Text Nodes and Attribute Values using UPDATEXML 4-15
4–12 Updating Selected Nodes within a Collection using UPDATEXML 4-16
4–13 NULL Updates with UPDATEXML – Element and Attribute 4-18
4–14 NULL Updates with UPDATEXML – Text Node 4-19
4–15 XPath Expressions in UPDATEXML Expression 4-21
4–16 Object Relational Equivalent of UPDATEXML Expression..... 4-21
4–17 Creating a View using UPDATEXML 4-22
4–18 Inserting a LineItem Element into a LineItems Element..... 4-24
4–19 Inserting an Element that Uses a Namespace..... 4-25
4–20 Inserting a LineItem Element Before the First LineItem ELement 4-28
4–21 Inserting a Date Element as the Last Child of an Action Element..... 4-31
4–22 Deleting LineItem Element Number 222..... 4-32
5–1 Creating Resources for Examples 5-15
5–2 XMLQuery Applied to a Sequence of Items of Different Types 5-16
5–3 FLOWR Expression using for, let, order by, where, and return 5-17
5–4 FLOWR Expression using Built-In Functions..... 5-18
5–5 Querying Relational Tables as XML..... 5-19
5–6 Using Relational Data in a Nested FLWOR Query 5-20
5–7 Querying a Relational Table as XML using XMLTable..... 5-22
5–8 Querying an XMLType Column using XMLQuery PASSING Clause 5-23
5–9 Using XMLTABLE with XML Schema-Based Data 5-24
5–10 Using XMLQUERY with Schema-Based Data 5-25
5–11 Using XMLTABLE with PASSING and COLUMNS Clauses 5-26
5–12 Decomposing XML Collection Elements into Relational Data using XMLTABLE..... 5-27
5–13 Using XMLQUERY with a Namespace Declaration..... 5-28
5–14 Using XMLTABLE with the XMLNAMESPACES Clause 5-29
5–15 Optimization of XMLQuery over Relational Data 5-31
5–16 Optimization of XMLTable over Relational Data 5-32
5–17 Optimization of XMLQuery with Schema-Based XMLType Data 5-33
5–18 Optimization of XMLTable with Schema-Based XMLType Data..... 5-33
5–19 Unoptimized Repository Query using fn:doc 5-36

5–20 Optimized Repository Query using EQUALS_PATH 5-36
5–21 Repository Query using Oracle XQuery Pragma ora:defaultTable 5-37
5–22 Static Type-Checking of XQuery Expressions: oradb URI scheme 5-37
5–23 Static Type-Checking of XQuery Expressions: Schema-Based XML 5-38
5–24 Using the SQL*Plus XQUERY Command 5-38
5–25 Using XQuery with PL/SQL 5-39
5–26 Using XQuery with JDBC 5-40
5–27 Using XQuery with ODP.NET and C# 5-41
6–1 CREATE INDEX using XMLCAST and XMLQUERY on a Singleton Element 6-6
6–2 CREATE INDEX using EXTRACTVALUE on a Singleton Element 6-7
6–3 Making Query Data Compatible with Index Data – SQL Cast 6-12
6–4 Making Query Data Compatible with Index Data – XQuery Cast 6-13
6–5 Path Table Contents for Two Purchase Orders 6-14
6–6 Creating an XMLIndex Index on XMLType Unstructured Storage 6-18
6–7 Obtaining the Name of an XMLIndex Index on a Particular Table 6-18
6–8 Renaming and Dropping an XMLIndex Index 6-18
6–9 Creating an XMLIndex Index on XMLType Hybrid Storage 6-19
6–10 XML Schema Fragment that Maps LineItems to CLOB Storage 6-19
6–11 Naming the Path Table of an XMLIndex Index 6-19
6–12 Determining the System-Generated Name of an XMLIndex Path Table 6-19
6–13 Specifying Storage Options When Creating an XMLIndex Index 6-20
6–14 Dropping an XMLIndex Unstructured Component 6-20
6–15 Determining the Names of the Secondary Indexes of an XMLIndex Index 6-20
6–16 Creating a Function-Based Index on Path-Table Column VALUE 6-21
6–17 Trying to Create a Numeric Index on Path-Table Column VALUE Directly 6-21
6–18 Creating a Numeric Index on Column VALUE with Procedure createNumberIndex . 6-21
6–19 Creating a Date Index on Column VALUE with Procedure createDateIndex 6-22
6–20 Creating an Oracle Text CONTEXT Index on Path-Table Column VALUE 6-22
6–21 Showing All Secondary Indexes on an XMLIndex Path Table 6-22
6–22 XMLIndex Index: Adding a Structured Component 6-23
6–23 Dropping an XMLIndex Structured Component 6-24
6–24 Creating a B-Tree Index on an XMLIndex Index Content Table 6-24
6–25 Oracle Text CONTEXT Index on an XMLIndex Index Content Table 6-24
6–26 XMLIndex with Only a Structured Component and using Namespaces 6-25
6–27 Checking Whether an XMLIndex Unstructured Component Is Used 6-26
6–28 Obtaining the Name of an XMLIndex Index from Its Path-Table Name 6-27
6–29 Extracting Data from an XML Fragment using XMLIndex 6-27
6–30 Using a Structured XMLIndex Component for a Query with Two Predicates 6-28
6–31 Using a Structured XMLIndex Component for a Query with Multilevel Chaining 6-29
6–32 Turning Off XMLIndex using Optimizer Hints 6-30
6–33 XMLIndex Path Subsetting with CREATE INDEX 6-32
6–34 XMLIndex Path Subsetting with ALTER INDEX 6-32
6–35 XMLIndex Path Subsetting using a Namespace Prefix 6-32
6–36 Using XMLIndex When Querying Resource Data 6-35
6–37 Using XMLIndex When Querying Resource Metadata 6-36
6–38 Creating an XMLIndex Index in Parallel 6-37
6–39 Using Different PARALLEL Degrees for XMLIndex Internal Objects 6-37
6–40 Specifying Deferred Synchronization for XMLIndex 6-39
6–41 Manually Synchronizing an XMLIndex Index using SYNCINDEX 6-39
6–42 Automatic Collection of Statistics on XMLIndex Objects 6-39
6–43 Creating an Oracle Text Index 6-49
6–44 Searching XML Data using SQL Function CONTAINS 6-49
6–45 Using an Oracle Text Index and an XMLIndex Index 6-50
7–1 XML Schema Instance purchaseOrder.xsd 7-2
7–2 purchaseOrder.xml: Document That Conforms to purchaseOrder.xsd 7-3

7–3 Registering an XML Schema using DBMS_XMLSCHEMA.REGISTERSCHEMA 7-7
7–4 Creating SQL Object Types to Store XMLType Tables..... 7-9
7–5 Default Table for Global Element PurchaseOrder 7-10
7–6 Data Dictionary Table for Registered Schemas 7-11
7–7 Deleting an XML Schema with DBMS_XMLSCHEMA.DELETESCHEMA 7-13
7–8 Registering a Local XML Schema 7-14
7–9 Registering a Global XML Schema 7-15
7–10 XML Schema Defining Documents with a Title To Be Translated 7-18
7–11 Untranslated Instance Document 7-19
7–12 XML Schema with Attribute xdb:translate for a Single-Valued Element..... 7-20
7–13 Translated Document 7-21
7–14 XML Schema with Attribute xdb:translate for a Multi-Valued Element..... 7-22
7–15 Translated Document for an XML Schema with Multiple-Valued Elements 7-23
7–16 Inserting a Document with No Language Information..... 7-24
7–17 Document After Insertion into the Repository 7-25
7–18 Inserting a Document with Language Information 7-25
7–19 Document After Insertion..... 7-25
7–20 Creating XML Schema-Based XMLType Tables and Columns 7-29
7–21 Specifying CLOB Storage for Schema-Based XMLType Tables and Columns..... 7-32
7–22 Specifying Structured Storage Options for XMLType Tables and Columns 7-32
7–23 Using STORE ALL VARRAYS AS..... 7-33
7–24 Using Common Schema Annotations..... 7-35
7–25 Registering an Annotated XML Schema 7-37
7–26 Querying Metadata from a Registered XML Schema..... 7-43
7–27 Mapping XML Schema Data Types to SQL Data Types using Attribute SQLType 7-46
8–1 XPath Rewrite..... 8-2
8–2 Execution Plan Generated When XPath Rewrite Does Not Occur 8-4
8–3 Analyzing an Execution Plan to Determine a Column to Index..... 8-4
8–4 Creating an Index on a Column Targeted by a Predicate..... 8-5
8–5 Creating a Function-Based Index for a Column Targeted by a Predicate 8-5
8–6 Execution Plan Showing that Index Is Picked Up..... 8-5
8–7 Creating a Function-Based Index for a Column Targeted by a Predicate 8-6
8–8 Execution Plan for a Selection of Collection Elements 8-6
8–9 Creating an Index for Direct Access to an Ordered Collection Table 8-7
9–1 Generating an XML Schema with Function GENERATESCHEMA 9-2
9–2 Adding a Unique Constraint to the Parent Element of an Attribute..... 9-3
9–3 Setting SQLInline to False for Out-Of-Line Storage 9-5
9–4 Generated XMLType Tables and Types 9-5
9–5 Querying an Out-Of-Line Table..... 9-6
9–6 XPath Rewrite for an Out-Of-Line Table..... 9-7
9–7 Using an Index with an Out-Of-Line Table 9-7
9–8 Storing a Collection Out of Line 9-8
9–9 Generated Out-Of-Line Collection Type 9-9
9–10 Renaming an Intermediate Table of REF Values..... 9-9
9–11 XPath Rewrite for an Out-Of-Line Collection..... 9-9
9–12 XPath Rewrite for an Out-Of-Line Collection, with Index on REFs..... 9-10
9–13 Specifying Partitioning Information During XML Schema Registration..... 9-11
9–14 Specifying Partitioning Information During Table Creation..... 9-12
9–15 Oracle XML DB XML Schema: Mapping complexType XML Fragments to LOBs..... 9-14
9–16 XML Schema Inheritance: complexContent as an Extension of complexTypes 9-15
9–17 Inheritance in XML Schema: Restrictions in complexTypes 9-16
9–18 XML Schema complexType: Mapping complexType to simpleContent 9-17
9–19 XML Schema: Mapping complexType to any/anyAttribute 9-18
9–20 An XML Schema with Circular Dependency 9-19
9–21 XML Schema: Cycling Between complexTypes 9-20

9–22 XML Schema: Cycling Between complexTypes, Self-Reference 9-21
9–23 An XML Schema that Includes a Non-Existent XML Schema 9-23
9–24 Using the FORCE Option to Register XML Schema xm40.xsd 9-23
9–25 Trying to Create a Table Using a Cyclic XML Schema 9-24
9–26 Using the FORCE Option to Register XML Schema xm40a.xsd 9-24
9–27 Recursive XML Schema 9-25
9–28 Out-of-line Table 9-26
9–29 Invalid Default Table Sharing 9-26
10–1 Revised Purchase-Order XML Schema 10-2
10–2 evolvePurchaseOrder.xsl: Style Sheet to Update Instance Documents 10-10
10–3 Loading Revised XML Schema and XSL Style Sheet 10-12
10–4 Updating an XML Schema using DBMS_XMLSCHEMA.COPYEVOLVE 10-12
10–5 Splitting a Complex Type into Two Complex Types 10-16
10–6 diffXML Parameter Document 10-21
11–1 Registering an XML Schema and Inserting XML Data 11-2
11–2 Retrieving a Style Sheet using XMLTRANSFORM and DBURITYPE 11-4
11–3 Retrieving a Style Sheet using XMLTRANSFORM and a Subquery 11-6
11–4 Using Method TRANSFORM() with a Transient Style Sheet 11-6
11–5 Validating XML using Method ISSCHEMAVALID() in SQL 11-8
11–6 Validating XML using Method ISSCHEMAVALID() in PL/SQL 11-8
11–7 Validating XML using Method SCHEMAVALIDATE() within Triggers 11-8
11–8 Checking XML Validity using XMLISVALID within CHECK Constraints 11-9
12–1 Simple Query using Oracle SQL Function CONTAINS 12-3
12–2 Restricting a Query using CONTAINS and WITHIN 12-3
12–3 Restricting a Query using CONTAINS and INPATH 12-4
12–4 ora:contains with an Arbitrarily Complex Text Query 12-4
12–5 CONTAINS Query with a Simple Boolean Operator 12-6
12–6 CONTAINS Query with Complex Boolean 12-6
12–7 CONTAINS Query with Stemming 12-6
12–8 CONTAINS Query with Complex Query Expression 12-6
12–9 Simple CONTAINS Query with SCORE 12-7
12–10 WITHIN 12-7
12–11 Nested WITHIN 12-7
12–12 WITHIN an Attribute 12-8
12–13 WITHIN and AND: Two Words in Some Comment Section 12-8
12–14 WITHIN and AND: Two Words in the Same Comment 12-8
12–15 WITHIN and AND: No Parentheses 12-8
12–16 WITHIN and AND: Parentheses Illustrating Operator Precedence 12-9
12–17 Structure Inside Full-Text Predicate: INPATH 12-9
12–18 Structure Inside Full-Text Predicate: INPATH 12-9
12–19 INPATH with Complex Path Expression (1) 12-10
12–20 INPATH with Complex Path Expression (2) 12-10
12–21 Nested INPATH 12-11
12–22 Nested INPATH Rewritten 12-11
12–23 Simple HASPATH 12-11
12–24 HASPATH Equality 12-11
12–25 HASPATH with Other Operators 12-11
12–26 Scoping the Results of a CONTAINS Query 12-12
12–27 Projecting the Result of a CONTAINS Query using ora:contains 12-12
12–28 Simple CONTEXT Index on Table PURCHASE_ORDERS 12-13
12–29 Simple CONTEXT Index on XMLType Table with Path Section Group 12-13
12–30 Simple CONTEXT Index on XMLType Column 12-13
12–31 Simple CONTEXT Index on XMLType Table 12-13
12–32 CONTAINS Query on XMLType Table 12-13
12–33 CONTAINS: Default Case Matching 12-14

12–34 Create a Preference for Mixed Case 12-15
12–35 CONTEXT Index on PURCHASE_ORDERS Table, Mixed Case 12-15
12–36 CONTAINS: Mixed (Exact) Case Matching 12-15
12–37 Simple CONTEXT Index on purchase_orders Table with Path Section Group 12-16
12–38 Using ora:contains with XMLQuery and XMLExists 12-18
12–39 Create a Policy to Use with ora:contains 12-19
12–40 Finding a Stopword using ora:contains 12-19
12–41 Finding a Stopword using ora:contains and Policy my_nostopwords_policy 12-19
12–42 ora:contains, Default Case-Sensitivity 12-20
12–43 Create a Preference for Mixed Case 12-20
12–44 Create a Policy with Mixed Case (Case-Insensitive) 12-21
12–45 ora:contains, Case-Sensitive (1) 12-21
12–46 ora:contains, Case-Sensitive (2) 12-21
12–47 ora:contains in Large Table 12-22
12–48 B-tree Index on ID 12-23
12–49 ora:contains in Large Table, with Additional Predicate 12-23
12–50 ora:contains Search for "electric" 12-24
12–51 Purchase Order XML Document, po001.xml 12-26
12–52 Create Table PURCHASE_ORDERS 12-27
12–53 Create Table PURCHASE_ORDERS_XMLTYPE 12-27
12–54 Create Table PURCHASE_ORDERS_XMLTYPE_TABLE 12-28
12–55 Purchase-Order XML Schema for Full-Text Search Examples 12-29
13–1 Creating and Manipulating a DOM Document 13-10
13–2 Creating an Element Node and Obtaining Information About It 13-12
13–3 Creating a User-Defined Subtype of SYS.util_BinaryOutputStream() 13-15
13–4 Retrieving Node Value with a User-Defined Stream 13-15
13–5 Get-Pull of Binary Data 13-16
13–6 Get-Pull of Character Data 13-16
13–7 Set-Pull of Binary Data 13-17
13–8 Set-Push of Binary Data 13-18
13–9 Parsing an XML Document 13-20
13–10 Transforming an XML Document using an XSL Style Sheet 13-22
14–1 Inserting Data with Specified Columns 14-2
14–2 Updating Data with Key Columns 14-4
14–3 DBMS_XMLSTORE.DELETEXML Example 14-5
15–1 Querying an XMLType Table using JDBC 15-2
15–2 Selecting XMLType Data using getStringVal() and getCLOB() 15-3
15–3 Returning XMLType Data using getObject() 15-3
15–4 Returning XMLType Data using an Output Parameter 15-3
15–5 Updating XMLType Data using SQL UPDATE with Constructor XMLType 15-4
15–6 Updating XMLType Data using SQL UPDATE with setObject() 15-5
15–7 Retrieving Metadata about XMLType Data using JDBC 15-5
15–8 Updating an Element in an XMLType Column using JDBC 15-5
15–9 Updated Purchase-Order Document 15-7
15–10 Manipulating an XMLType Column using JDBC 15-8
15–11 Java Method insertXML() 15-11
15–12 Java Method getCLOB() 15-12
15–13 Creating a DOM Object with the Java DOM API 15-14
15–14 Using the Java DOM API with Binary XML 15-21
16–1 Using OCIXMLDBINITXMLCTX() and OCIXMLDBFREEXMLCTX() 16-3
16–2 Using the C API for XML with Binary XML 16-7
16–3 Using the Oracle XML DB Pull Parser 16-10
16–4 Using the DOM to Count Ordered Parts 16-15
17–1 Retrieve XMLType Data to .NET 17-2
18–1 XMLELEMENT: Formatting a Date 18-6

18–2 XMLELEMENT: Generating an Element for Each Employee 18-6
18–3 XMLELEMENT: Generating Nested XML 18-7
18–4 XMLELEMENT: Generating Employee Elements with Attributes ID and Name 18-7
18–5 XMLELEMENT: Characters in Generated XML Are Not Escaped 18-8
18–6 Creating a Schema-Based XML Document using XMLELEMENT with Namespaces.. 18-8
18–7 XMLELEMENT: Generating an Element from a User-Defined Data-Type Instance..... 18-9
18–8 XMLFOREST: Generating Elements with Attribute and Child Elements..... 18-10
18–9 XMLFOREST: Generating an Element from a User-Defined Data-Type Instance..... 18-11
18–10 XMLCONCAT: Concatenating XMLType Instances from a Sequence..... 18-12
18–11 XMLCONCAT: Concatenating XML Elements 18-12
18–12 XMLAGG: Generating a Department Element with Child Employee Elements 18-13
18–13 XMLAGG: Using GROUP BY to Generate Multiple Department Elements..... 18-13
18–14 XMLAGG: Generating Nested Elements..... 18-14
18–15 Using SQL/XML Function XMLPI..... 18-16
18–16 Using SQL/XML Function XMLCOMMENT..... 18-16
18–17 Using SQL/XML Function XMLSERIALIZE..... 18-18
18–18 Using SQL/XML Function XMLPARSE..... 18-18
18–19 Using Oracle SQL Function XMLRoot..... 18-19
18–20 XMLCOLATTVAL: Generating Elements with Attribute and Child Elements 18-20
18–21 Using Oracle SQL Function XMLCDATA 18-21
18–22 DBMS_XMLGEN: Generating Simple XML 18-28
18–23 DBMS_XMLGEN: Generating Simple XML with Pagination (Fetch)..... 18-29
18–24 DBMS_XMLGEN: Generating XML using Object Types..... 18-31
18–25 DBMS_XMLGEN: Generating XML using User-Defined Data-Type Instances..... 18-33
18–26 DBMS_XMLGEN: Generating an XML Purchase Order..... 18-34
18–27 DBMS_XMLGEN: Generating a New Context Handle from a REF Cursor..... 18-39
18–28 DBMS_XMLGEN: Specifying NULL Handling 18-40
18–29 DBMS_XMLGEN: Generating Recursive XML with a Hierarchical Query 18-41
18–30 DBMS_XMLGEN: Binding Query Variables using SETBINDVALUE() 18-43
18–31 Creating XML Data using SYS_XMLGEN 18-46
18–32 SYS_XMLGEN: Generating an XML Element from a Database Column..... 18-47
18–33 SYS_XMLGEN: Converting a Scalar Value to XML Element Contents..... 18-48
18–34 SYS_XMLGEN: Default Element Name ROW 18-49
18–35 Overriding the Default Element Name using SYS_XMLGEN with XMLFormat 18-49
18–36 SYS_XMLGEN: Converting a User-Defined Data-Type Instance to XML 18-50
18–37 SYS_XMLGEN: Converting an XMLType Instance..... 18-51
18–38 Using SYS_XMLGEN with Object Views..... 18-52
18–39 Using XSQL Servlet <xsql:include-xml> with Nested XMLAgg Functions 18-54
18–40 Using XSQL Servlet <xsql:include-xml> with XMLElement and XMLAgg 18-55
18–41 Using XMLAGG ORDER BY Clause..... 18-57
18–42 Returning a Rowset using XMLTABLE..... 18-58
19–1 Creating an XMLType View using XMLELEMENT 19-3
19–2 Creating an XMLType View using Object Types and SYS_XMLGEN 19-3
19–3 Registering XML Schema emp_simple.xsd..... 19-5
19–4 Creating an XMLType View using SQL/XML Publishing Functions 19-5
19–5 Querying an XMLType View..... 19-6
19–6 Using Namespace Prefixes with SQL/XML Publishing Functions..... 19-7
19–7 XML Schema with No Target Namespace 19-8
19–8 Creating a View for an XML Schema with No Target Namespace 19-8
19–9 Using SQL/XML Functions in XML Schema-Based XMLType Views..... 19-9
19–10 Creating Object Types for Schema-Based XMLType Views..... 19-11
19–11 Generating an XML Schema with DBMS_XMLSCHEMA.GENERATESCHEMA 19-12
19–12 Registering XML Schema emp_complex.xsd..... 19-12
19–13 Creating XMLType View emp_xml 19-13
19–14 Creating an Object View and an XMLType View on the Object View 19-14

19–15 Creating Object Types 19-14
19–16 Registering XML Schema dept_complex.xsd 19-15
19–17 Creating XMLType View dept_xml using Object Type dept_t 19-16
19–18 Creating XMLType View dept_xml using Relational Data Directly 19-16
19–19 Creating an XMLType View by Restricting Rows from an XMLType Table 19-17
19–20 Creating an XMLType View by Transforming an XMLType Table 19-17
19–21 Determining Whether an XMLType View is Implicitly Updatable 19-18
20–1 Using HTTPURIType PL/SQL Method GETCONTENTTYPE() 20-5
20–2 Creating and Querying a URI Column 20-7
20–3 Using Different Kinds of URI, Created in Different Ways 20-8
20–4 Access a Repository Resource by URI using an XDBUri 20-10
20–5 Using PL/SQL Method GETXML() with XMLCAST and XMLQUERY 20-12
20–6 Targeting a Complete Table using a DBUri 20-16
20–7 Targeting a Particular Row in a Table using a DBUri 20-17
20–8 Targeting a Specific Column using a DBUri 20-18
20–9 Targeting an Object Column with Specific Attribute Values using a DBUri 20-18
20–10 Retrieve Only the Text Value of a Node using a DBUri 20-19
20–11 Targeting a Collection using a DBUri 20-19
20–12 URIFACTORY: Registering the ECOM Protocol 20-21
20–13 SYS_DBURIGEN: Generating a DBUri that Targets a Column 20-22
20–14 Passing Columns with Single Arguments to SYS_DBURIGEN 20-23
20–15 Inserting Database References using SYS_DBURIGEN 20-24
20–16 Creating the Travel Story Table 20-24
20–17 A Function that Returns the First 20 Characters 20-24
20–18 Creating a Travel View for Use with SYS_DBURIGEN 20-25
20–19 Retrieving a URL using SYS_DBURIGEN in RETURNING Clause 20-25
20–20 Changing the Installation Location of DBUriServlet 20-28
20–21 Restricting Servlet Access to a Database Role 20-28
20–22 Registering a Handler for a DBUri Prefix 20-29
21–1 Querying PATH_VIEW to Determine Link Type 21-8
21–2 Obtaining the OID Path of a Resource 21-8
21–3 Creating a Weak Link using an OID Path 21-9
22–1 Resource Configuration File 22-4
22–2 applicationData Element 22-5
23–1 XInclude Used in a Book Document to Include Parts and Chapters 23-3
23–2 Expanding Document Inclusions using XDBURIType 23-5
23–3 Querying Document Links Mapped From XLink Links 23-8
23–4 Querying Document Links Mapped From XInclude Links 23-8
23–5 Mapping XInclude Links to Hard Document Links, with OID Retrieval 23-12
23–6 Mapping XLInk Links to Weak Links, with Named-Path Retrieval 23-12
23–7 Configuring XInclude Document Decomposition 23-12
23–8 Repository Document, Showing Generated xi:include Elements 23-13
24–1 Creating a Repository Resource 24-4
24–2 Creating a Version-Controlled Resource 24-5
24–3 Retrieving Resource Content by Referencing the Resource ID 24-5
24–4 Checking Out a Version-Controlled Resource 24-5
24–5 Updating Resource Content 24-5
24–6 Checking In a Version-Controlled Resource 24-6
24–7 Retrieving Resource Version Content using XDBURITYPE and CREATEOIDPATH 24-7
24–8 Retrieving Resource Version Content using GETCONTENTSCLOBBYRESID 24-7
24–9 Retrieving Resource Version Metadata using GETRESOURCEBYRESID 24-8
24–10 Canceling a Check-Out using UNCHECKOUT 24-9
25–1 Determining Paths Under a Path: Relative 25-8
25–2 Determining Paths Under a Path: Absolute 25-8
25–3 Determining Paths Not Under a Path 25-9

25–4 Determining Paths using Multiple Correlations 25-9
25–5 Relative Path Names for Three Levels of Resources 25-10
25–6 Extracting Resource Metadata using UNDER_PATH 25-10
25–7 Using Functions PATH and DEPTH with PATH_VIEW 25-11
25–8 Extracting Link and Resource Information from PATH_VIEW 25-11
25–9 All Repository Paths to a Certain Depth Under a Path 25-12
25–10 Locating a Repository Path using EQUALS_PATH 25-12
25–11 Retrieve RESID of a Given Resource 25-12
25–12 Obtaining the Path Name of a Resource from its RESID 25-13
25–13 Folders Under a Given Path 25-13
25–14 Joining RESOURCE_VIEW with an XMLType Table 25-13
25–15 Deleting Resources 25-14
25–16 Deleting Links to Resources 25-14
25–17 Deleting a Nonempty Folder 25-15
25–18 Updating a Resource 25-16
25–19 Updating a Path in the PATH_VIEW 25-17
25–20 Updating Resources Based on Attributes 25-18
25–21 Finding Resources Inside a Folder 25-18
25–22 Copying Resources 25-19
25–23 Find All Resources Containing "Paper" 25-20
25–24 Find All Resources Containing "Paper" that are Under a Specified Path 25-20
26–1 Managing Resources using DBMS_XDB 26-2
26–2 Using DBMS_XDB.GETACLDOCUMENT 26-4
26–3 Using DBMS_XDB.SETACL 26-4
26–4 Using DBMS_XDB.CHANGEPRIVILEGES 26-5
26–5 Using DBMS_XDB.GETPRIVILEGES 26-6
26–6 Using DBMS_XDB.CFG_GET 26-7
26–7 Using DBMS_XDB.CFG_UPDATE 26-8
27–1 Simple Access Control Entry (ACE) that Grants a Privilege 27-4
27–2 Simple Access Control List (ACL) that Grants a Privilege 27-4
27–3 Element extends-from 27-9
27–4 Element constrained-with 27-9
27–5 Complementing a Set of Principals with Element invert 27-10
27–6 ACE with Start and End Dates 27-11
27–7 Creating an ACL using CREATERESOURCE 27-11
27–8 Retrieving an ACL Document, Given its Repository Path 27-12
27–9 Setting the ACL of a Resource 27-12
27–10 Deleting an ACL 27-12
27–11 Updating (Replacing) an Access Control List 27-13
27–12 Appending ACEs to an Access Control List 27-14
27–13 Deleting an ACE from an Access Control List 27-14
27–14 Retrieving the ACL Document for a Resource 27-14
27–15 Retrieving Privileges Granted to the Current User for a Particular Resource 27-15
27–16 Checking If a User Has a Certain Privileges on a Resource 27-15
27–17 Checking User Privileges using ACLCheckPrivileges 27-16
27–18 Retrieving the Path of the ACL that Protects a Given Resource 27-17
27–19 Retrieving the Paths of All Resources Protected by a Given ACL 27-17
27–20 ACL Referencing an LDAP User 27-21
27–21 ACL Referencing an LDAP Group 27-21
28–1 Navigating Oracle ASM Folders 28-11
28–2 Transferring Oracle ASM Files Between Databases with FTP proxy Method 28-12
28–3 FTP Connection Using IPv6 28-13
28–4 Modifying the Default Timeout Value of an FTP Session 28-14
29–1 Register an XML Schema for Technical Photo Information 29-3
29–2 Register an XML Schema for Photo Categorization 29-4

29–3 Add Metadata to a Resource – Technical Photo Information 29-5
29–4 Add Metadata to a Resource – Photo Content Categories 29-6
29–5 Delete Specific Metadata from a Resource 29-6
29–6 Adding Metadata to a Resource using DML with RESOURCE_VIEW 29-7
29–7 Adding Metadata using WebDAV PROPPATCH 29-8
29–8 Query XML Schema-Based Resource Metadata 29-10
29–9 Add Non-Schema-Based Metadata to a Resource 29-11
30–1 Resource Configuration File for Java Event Listeners with Preconditions 30-9
30–2 Resource Configuration File for PL/SQL Event Listeners with No Preconditions 30-11
30–3 PL/SQL Code Implementing Event Listeners 30-11
30–4 Java Code Implementing Event Listeners 30-13
30–5 Invoking Event Handlers 30-15
31–1 JCR Node Representation of MyFolder 31-3
31–2 Code Fragment Showing How to Get a Repository Object 31-7
31–3 Uploading a File using Oracle XML DB Content Connector 31-8
31–4 Uploading a File Using the Command Line 31-9
31–5 XML Document with XML Schema-Based Content 31-12
31–6 XML Schema 31-12
31–7 JCR Representation of XML Content Not Registered for JCR Use 31-12
31–8 JCR Representation of XML Content Registered for JCR Use 31-13
31–9 Registering an XML Schema for Use with Oracle XML DB 31-13
31–10 Registering an XML Schema for Use with JCR 31-14
32–1 An Oracle XML DB Servlet 32-7
32–2 Registering and Mapping an Oracle XML DB Servlet 32-9
33–1 Adding a Web Services Configuration Servlet 33-2
33–2 Verifying Addition of Web Services Configuration Servlet 33-3
33–3 XML Schema for Database Queries To Be Processed by Web Service 33-4
33–4 Input XML Document for SQL Query using Query Web Service 33-5
33–5 Output XML Document for SQL Query using Query Web Service 33-6
33–6 Definition of PL/SQL Function Used for Web-Service Access 33-7
33–7 WSDL Document Corresponding to a Stored PL/SQL Function 33-8
33–8 Input XML Document for PL/SQL Query using Web Service 33-9
33–9 Output XML Document for PL/SQL Query using Web Service 33-9
34–1 Oracle XML DB Configuration File 34-8
34–2 Updating the Configuration File using CFG_UPDATE and CFG_GET 34-10
35–1 Data File filelist.dat: List of XML Files to Load 35-4
35–2 Control File load_datra.ctl, for Loading Purchase-Order XML Documents 35-4
35–3 Loading XML Data Using Shell Command sqlldr 35-4
36–1 Exporting XMLType Data in TABLE Mode 36-4
36–2 Importing XMLType Data in TABLE Mode 36-4
36–3 Creating Table po2 36-5
36–4 Exporting XMLType Data in SCHEMA Mode 36-5
36–5 Importing XMLType Data in SCHEMA Mode 36-5
36–6 Importing XMLType Data in SCHEMA Mode, Remapping Schema 36-5
37–1 Creating a Queue Table and Queue 37-6
37–2 Creating a Transformation to Convert Message Data to XML 37-6
37–3 Applying a Transformation before Sending Messages Overseas 37-7
37–4 XMLType and AQ: Dequeuing Messages 37-7
A–1 Annotated Purchase-Order XML Schema, purchaseOrder.xsd A-30
A–2 Revised Purchase-Order XML Schema A-33
A–3 PurchaseOrder.xsl Style Sheet A-38
A–4 Inserting XML Data into an XMLType Table using C A-43
A–5 Using OCIXmlDbInitXmlCtx() and OCIXmlDbFreeXmlCtx() A-47

Preface

This manual describes Oracle XML DB, and how you can use it to store, generate, manipulate, manage, and query XML data in the database.

After introducing you to the heart of Oracle XML DB, namely the `XMLType` framework and Oracle XML DB Repository, the manual provides a brief introduction to design criteria to consider when planning your Oracle XML DB application. It provides examples of how and where you can use Oracle XML DB.

The manual then describes ways you can store and retrieve XML data using Oracle XML DB, APIs for manipulating `XMLType` data, and ways you can view, generate, transform, and search on existing XML data. The remainder of the manual discusses how to use Oracle XML DB Repository, including versioning and security, how to access and manipulate repository resources using protocols, SQL, PL/SQL, or Java, and how to manage your Oracle XML DB application using Oracle Enterprise Manager. It also introduces you to XML messaging and Oracle Streams Advanced Queuing `XMLType` support.

This Preface contains these topics:

- Audience
- Documentation Accessibility
- Related Documents
- Conventions

Audience

Oracle XML DB Developer's Guide is intended for developers building XML Oracle Database applications.

An understanding of XML, XML Schema, XQuery, XPath, and XSL is helpful when using this manual.

Many examples provided here are in SQL, PL/SQL, Java, or C. A working knowledge of one of these languages is presumed.

Documentation Accessibility

Our goal is to make Oracle products, services, and supporting documentation accessible to all users, including users that are disabled. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Accessibility standards will continue to

evolve over time, and Oracle is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For more information, visit the Oracle Accessibility Program Web site at http://www.oracle.com/accessibility/.

Accessibility of Code Examples in Documentation

Screen readers may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should appear on an otherwise empty line; however, some screen readers may not always read a line of text that consists solely of a bracket or brace.

Accessibility of Links to External Web Sites in Documentation

This documentation may contain links to Web sites of other companies or organizations that Oracle does not own or control. Oracle neither evaluates nor makes any representations regarding the accessibility of these Web sites.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/support/contact.html or visit http://www.oracle.com/accessibility/support.html if you are hearing impaired.

Related Documents

For more information, see these Oracle resources:

- *Oracle Database New Features Guide* for information about the differences between Oracle Database 11*g* and the Oracle Database 11*g* Enterprise Edition and the available features and options. This book also describes features new to Oracle Database 11*g* Release 2 (11.2).
- *Oracle Database XML Java API Reference*
- *Oracle XML Developer's Kit Programmer's Guide*
- *Oracle Database Error Messages*. Oracle Database error message documentation is available only as HTML. If you have access to only printed or PDF Oracle Database documentation, you can browse the error messages by range. Once you find the specific range, use the search (find) function of your Web browser to locate the specific message. When connected to the Internet, you can search for a specific error message using the error message search feature of the Oracle Database online documentation.
- *Oracle Text Application Developer's Guide*
- *Oracle Text Reference*
- *Oracle Database Concepts.*
- *Oracle Database Java Developer's Guide*
- *Oracle Database Advanced Application Developer's Guide*
- *Oracle Streams Advanced Queuing User's Guide*
- *Oracle Database PL/SQL Packages and Types Reference*

Many of the examples in this book use the sample schemas, which are installed by default when you select the Basic Installation option with an Oracle Database

installation. Refer to *Oracle Database Sample Schemas* for information about how these schemas were created and how you can use them yourself.

To download free release notes, installation documentation, white papers, or other collateral, please visit the Oracle Technology Network (OTN). You must register online before using OTN; registration is free and can be done at

http://www.oracle.com/technology/membership/

If you already have a username and password for OTN, then you can go directly to the documentation section of the OTN Web site at

http://www.oracle.com/technology/documentation/

For additional information, see:

- http://www.w3.org/TR/xml/ – XML (language)
- http://www.xml.com/pub/a/98/10/guide0.html – XML introduction
- http://www.w3.org/XML/Schema – XML Schema
- http://www.w3.org/2001/XMLSchema – XML Schema
- http://www.w3.org/TR/xmlschema-0/ – XML Schema: primer
- http://www.w3.org/TR/xmlschema-1/ – XML Schema: structures
- http://www.w3.org/TR/xmlschema-2/ – XML Schema: data types
- http://www.oasis-open.org/cover/schemas.html – XML Schema
- http://www.xml.com/pub/a/2000/11/29/schemas/part1.html – XML Schema
- http://xml.coverpages.org/xmlMediaMIME.html – media/MIME types
- http://www.w3.org/TR/xptr/ – XPointer
- http://www.w3.org/TR/xpath – XPath 1.0
- http://www.w3.org/TR/xpath20/ – XPath 2.0
- http://www.zvon.org/xxl/XPathTutorial/General/examples.html – XPath
- *XML In a Nutshell*, by Elliotte Rusty Harold and W. Scott Means, O'Reilly, January 2001, http://www.oreilly.com/catalog/xmlnut/chapter/ch09.html
- http://www.w3.org/TR/2002/NOTE-unicode-xml-20020218/ – Unicode in XML
- http://www.w3.org/TR/xml-names/ – XML namespaces
- http://www.w3.org/TR/xml-infoset/ – information sets
- http://www.w3.org/DOM/ – Document Object Model (DOM)
- http://www.w3.org/TR/xslt – XSLT
- http://www.w3.org/TR/xsl – XSL
- http://www.oasis-open.org/cover/xsl.html – XSL
- http://www.zvon.org/xxl/XSLTutorial/Output/index.html – XSL
- http://www.w3.org/2002/ws/Activity.html – Web services
- http://www.ietf.org/rfc/rfc959.txt – RFC 959: FTP Protocol Specification

- ISO/IEC 13249-2:2000, Information technology - Database languages - SQL Multimedia and Application Packages - Part 2: Full-Text, International Organization For Standardization, 2000
- http://java.sun.com/xml/tutorial_intro.html – XML and Java

Note: Throughout this manual, "**XML Schema**" refers to the XML Schema 1.0 recommendation, http://www.w3.org/XML/Schema.

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

Code Examples

The code examples in this book are for illustration only. In many cases, however, you can copy and paste parts of examples and run them in your environment.

Standard Database Schemas

Many of the examples in this book use the standard database schemas that are included in your database. In particular, database schema `OE` contains XML purchase-order documents in `XMLType` table `purchaseorder`, and XML documents with warehouse information in `XMLType` column `warehouse_spec` of table `warehouses`.

The purchase-order documents are also contained in Oracle XML DB Repository, under the repository path `/home/OE/PurchaseOrders/2002/`. The XML schema that governs these documents is file `purchaseorder.xsd`, at repository location `/home/OE/purchaseorder.xsd`. An XSL style sheet that is used in some examples to transform purchase-order documents is file `purchaseorder.xsl`, at repository location `/home/OE/purchaseorder.xsl`. This XML schema and style sheet can also be found in Appendix A, "Oracle-Supplied XML Schemas and Examples".

See Also:

- *Oracle Database Sample Schemas* for information about database schema `HR`
- *Oracle Database Sample Schemas* for information about database schema `OE`

Pretty Printing of XML Data

To promote readability, especially of lengthy or complex XML data, output is sometimes shown pretty-printed (formatted) in code examples.

Execution Plans

Some of the code examples in this book present execution plans. These are for illustration only. Running examples that are presented here in your environment is likely to result in different execution plans from those presented here.

Reminder About Case Sensitivity

When examining the examples in this book, keep in mind the following:

- SQL is case-insensitive, but names in SQL code are implicitly uppercase, unless you enclose them in double-quotes.
- XML is case-sensitive. You must refer to SQL names in XML code using the correct case: uppercase SQL names must be written as uppercase.

For example, if you create a table named `my_table` in SQL without using double-quotes, then you must refer to it in XML code as "`MY_TABLE`".

Syntax Descriptions

Syntax descriptions are provided in this book for various SQL, PL/SQL, or other command-line constructs in graphic form or Backus Naur Form (BNF). See *Oracle Database SQL Language Reference* for information about how to interpret these descriptions.

What's New in Oracle XML DB?

This chapter describes the new features and functionality, enhancements, APIs, and product integration support added to Oracle XML DB for Oracle Database 11*g*.

It also describes the deprecation of certain Oracle XML DB constructs.

Oracle Database 11g Release 2 (11.2.0.2) New Features in Oracle XML DB

The following Oracle XML DB features are new in Oracle Database 11*g* Release 2 (11.2.0.2).

Default Storage Model for XMLType

The default `XMLType` storage model is used if you do not specify a storage model when you create an `XMLType` table or column. Prior to Oracle Database 11*g* Release 2 (11.2.0.2), unstructured (`CLOB`) storage was used by default. The default storage model is now binary XML storage.

> **Note:** You can create a new table that uses binary XML storage and populate it with existing `XMLType` data that is stored using `CLOB` storage. Use `CREATE TABLE AS SELECT`..., selecting from the existing data.

This functionality is available starting with Oracle Database 11*g* Release 2 (11.2.0.2).

Default LOB Storage for Binary XML

`XMLType` data that uses the binary XML storage model is stored internally using large objects (LOBs). Prior to Oracle Database 11*g* Release 2 (11.2.0.2), binary XML data was stored by default using the BasicFile LOB storage option. By default, LOB storage for binary XML data now uses the SecureFile LOB storage option whenever possible.

If SecureFile LOB storage is not possible then the default behavior uses BasicFile LOB storage. This can occur if either of the following is true:

- The tablespace for the `XMLType` table does not use automatic segment space management.
- A setting in file `init.ora` prevents SecureFile LOB storage. For example, see parameter `DB_SECUREFILE`.

See Also:

- *Oracle Database Administrator's Guide* for information about automatic segment space management
- *Oracle Database Reference* for information about parameter `DB_SECUREFILE`
- "Oracle Database 11g Release 2 (11.2.0.2) Deprecated Oracle XML DB Constructs" on page -xlviii

This functionality is available starting with Oracle Database 11*g* Release 2 (11.2.0.2).

XQuery Pragma ora:defaultTable for Repository Query Performance

Previously, to obtain optimal performance for XQuery expressions that use `fn:doc` and `fn:collection` over Oracle XML DB Repository resources, you needed to carry out explicit joins with `RESOURCE_VIEW`. The new XQuery extension-expression pragma `ora:defaultTable` now performs the necessary joins automatically.

This functionality is available starting with Oracle Database 11*g* Release 2 (11.2.0.2).

XML Diagnosability Mode: SQL*Plus System Variable XMLOptimizationCheck

You can use the SQL*Plus `SET` command with the new system variable `XMLOptimizationCheck` to turn on an XML diagnosability mode for SQL. When this mode is on, execution plans are automatically checked for XPath rewrite, and if a plan is suboptimal then an error is raised and diagnostic information is written to the trace file indicating which operators are not rewritten.

This functionality is available starting with Oracle Database 11*g* Release 2 (11.2.0.2).

See Also: "Diagnosing XQuery Optimization: XMLOptimizationCheck" on page 5-35

Oracle Database 11g Release 2 (11.2.0.2) Deprecated Oracle XML DB Constructs

The following Oracle XML DB constructs are *deprecated* in Oracle Database 11*g* Release 2 (11.2.0.2). They are still supported in 11.2.0.2 for backward compatibility, but Oracle recommends that you do not use them in new applications.

- `XMLType` data stored as binary XML using BasicFile LOB storage. See also the new feature "Default LOB Storage for Binary XML" on page -xlvii.
- Oracle XQuery function `ora:view` – Use XQuery functions `fn:doc` and `fn:collection` instead. See Chapter 5, "Using XQuery with Oracle XML DB".

Oracle Database 11g Release 2 (11.2.0.1) New Features in Oracle XML DB

The following Oracle XML DB features are new in Oracle Database 11*g* Release 2 (11.2.0.1).

Partitioning XMLType Tables and Columns

For `XMLType` data stored object-relationally, when you partition a base `XMLType` table or a base table with an `XMLType` column, any collection tables that use heap-based table storage are now, by default, automatically equipartitioned also. Equipartitioning

means that there is a corresponding collection-table partition for each partition of the base table. A child element is stored in the collection-table partition that corresponds to the base-table partition of its parent element.

> **See Also:** "Partitioning XMLType Tables and Columns Stored Object-Relationally" on page 9-10

Access Control Enhancements

Access control lists (ACLs) have been enhanced in various ways, to provide fine-grained access control that you can customize. You can define your own privileges and associate them with users and roles in flexible ways. Inheritance is available for ACLs. Access control entries (ACEs) can stipulate start and end dates. You can control access for application users and roles that are not necessarily the same as database users and roles.

> **See Also:** Chapter 27, "Repository Access Control"

Repository Read and Write Performance Enhancements

Performance has been improved for Oracle XML DB Repository read and write operations.

Binary XML Performance Enhancements and Partitioning

The performance of queries and DML operations on binary XML tables has been improved, and you can now partition binary XML tables, using a virtual column as the partitioning key.

XMLIndex Enhancements

You can use `XMLIndex` to index islands of structured XML content embedded in content that is generally unstructured. An `XMLIndex` index can thus index both structured and unstructured XML content.

You can create a local `XMLIndex` index on data in partitioned `XMLType` tables.

> **See Also:** "XMLIndex" on page 6-7

Cost-Based XPath Rewrite

You can use a new optimizer hint to request cost-based optimization of XQuery expressions.

Oracle Database 11g Release 2 (11.2.0.1) Deprecated Oracle XML DB Constructs

The following Oracle XML DB constructs are *deprecated* in Oracle Database 11*g* Release 2 (11.2.0.1). They are still supported in 11.2.0.1 for backward compatibility, but Oracle recommends that you do not use them in new applications.

- Oracle SQL function `extract` – Use SQL/XML function `XMLQuery` instead. See "XMLQUERY SQL/XML Function in Oracle XML DB" on page 5-6.
- Oracle SQL function `extractValue` – Use SQL/XML function `XMLTable` or SQL/XML functions `XMLCast` and `XMLQuery` instead.
  - See "SQL/XML Functions XMLQUERY and XMLTABLE" on page 5-5 for information about using function `XMLTable`

  - See "XMLCAST SQL/XML Function" on page 4-4 for information about using functions `XMLCast` and `XMLQuery`

- Oracle SQL function `existsNode` – Use SQL/XML function `XMLExists` instead. See "XMLEXISTS SQL/XML Function" on page 4-3.
- Oracle SQL function `XMLSequence` – Use SQL/XML function `XMLTable` instead. See "XMLTABLE SQL/XML Function in Oracle XML DB" on page 5-7.
- Oracle XPath function `ora:instanceof` – Use XQuery operator `instance of` instead.
- Oracle XPath function `ora:instanceof-only` – Use XML Schema attribute `xsi:type` instead.
- PL/SQL `XMLType` methods `getStringVal()`, `getCLOBVal()`, and `getBLOBVal()`, – Use SQL/XML function `XMLSerialize` instead. See "XMLSERIALIZE SQL/XML Function" on page 18-17.
- PL/SQL `XMLType` method `getNamespace()` – Use XQuery function `fn:namespace-uri` instead.
- PL/SQL `XMLType` method `getRootElement()` – Use XQuery function `fn:local-name` instead.
- Function-based indexes on `XMLType` – Use `XMLIndex` with a structured component instead. See "Function-Based Indexes" on page 6-5.

Oracle Database 11g Release 1 (11.1) New Features in Oracle XML DB

Binary XML

Binary XML is a new storage model for abstract data type `XMLType`, joining the existing storage models of structured (object-relational) and unstructured (`CLOB`) storage. Binary XML is XML-Schema aware, but it can also be used with XML data that is not based on an XML schema. See "XMLType Storage Models" on page 1-14.

See Also:

- *Oracle Database Advanced Application Developer's Guide* for an overview of `XMLType` data stored as binary XML
- *Oracle Database SQL Language Reference* for information about creating `XMLType` tables and columns stored as binary XML
- *Oracle Database XML Java API Reference* for information about manipulating binary XML data using Java
- *Oracle Database XML C API Reference* for information about manipulating binary XML data using C

XMLIndex

A new index type is provided for `XMLType`: `XMLIndex`. This can greatly improve the performance of XPath-based predicates and fragment extraction for `XMLType` data, whether based on an XML schema or not. The new index type is a (logical) domain index that consists of underlying physical tables and secondary indexes. See Chapter 6, "Indexing XMLType Data".

Note: The `CTXSYS.CTXXPath` index is *deprecated* in Oracle Database 11g Release 1 (11.1). The functionality that was provided by `CTXXPath` is now provided by `XMLIndex`.

Oracle recommends that you replace `CTXXPath` indexes with `XMLIndex` indexes. The intention is that `CTXXPath` will no longer be supported in a future release of the database.

See Also:

- *Oracle Database Reference* for information about new view `XIDX_USER_PENDING`
- *Oracle Database PL/SQL Packages and Types Reference* for information about new PL/SQL package `DBMS_XMLINDEX`

XMLType OCTs Now Use Heap Storage Instead of IOTs

You can store collections of XML elements as ordered collection tables (OCTs). OCTs now use heap storage, by default. In prior releases, OCTs were index-organized tables (IOTs), by default. A new XML schema registration option, `REGISTER_NT_AS_IOT`, forces the use of IOTs.

See Also: "Controlling How Collections Are Stored for Object-Relational XMLType Storage" on page 3-19

Default Value of XML Schema Annotation storeVarrayAsTable Is Now true

In prior releases, the default value of XML schema annotation `storeVarrayAsTable` was `false`; the default value is now `true`. This means that by default an XML collection is stored as a set of rows in an ordered collection table (OCT). Each row corresponds to an element in the collection. With annotation `storeVarrayAsTable = "false"`, the entire collection is instead serialized as a varray and stored in a LOB column.

Using `storeVarrayAsTable = "true"` facilitates the efficient use of collections: queries, updates, and creation of B-tree indexes.

See Also: "Controlling How Collections Are Stored for Object-Relational XMLType Storage" on page 3-19 for more information about storing XML collections object-relationally

Repository Events

Applications can now register listeners with handlers for events associated with Oracle XML DB Repository operations such as creating, deleting, and updating a resource. See Chapter 30, "Oracle XML DB Repository Events".

See Also:

- *Oracle Database XML Java API Reference* for new Java methods
- *Oracle Database PL/SQL Packages and Types Reference* for information about new PL/SQL package `DBMS_XEVENT`
- *Oracle Database PL/SQL Packages and Types Reference* for information about new PL/SQL package `DBMS_RESCONFIG`
- *Oracle Database PL/SQL Packages and Types Reference* for information about new PL/SQL package `DBMS_XDBRESOURCE`

Support for Content Repository API for Java (JCR: JSR-170)

Oracle XML DB now supports Content Repository API for Java (JCR) and the JSR-170 standard. You can access Oracle XML DB Repository using the JCR APIs. See Chapter 31, "Using Oracle XML DB Content Connector".

> **See Also:** *Oracle Database XML Java API Reference* for new Java methods

New Repository Resource Link Types

You can now create weak folder links to represent Oracle XML DB Repository folder-child relationships. Hard links are still available, as well. See "Link Types" on page 21-7.

> **See Also:**
>
> - *Oracle Database PL/SQL Packages and Types Reference* for updates to PL/SQL package `DBMS_XDB`
> - *Oracle Database SQL Language Reference* for updates to function `under_path`

Support for WebDAV Privileges and New Oracle XML DB Privileges

All WebDAV privileges are now supported by Oracle XML DB Repository. In addition, there are some new Oracle XML DB-specific atomic privileges. See Chapter 27, "Repository Access Control".

> **See Also:**
>
> - *Oracle Database PL/SQL Packages and Types Reference* for information about new PL/SQL package `DBMS_NETWORK_ACL_ADMIN`
> - *Oracle Database PL/SQL Packages and Types Reference* for information about PL/SQL package `UTL_TCP`
> - *Oracle Database PL/SQL Packages and Types Reference* for information about PL/SQL package `UTL_INADDR`

Web Services

You can now access Oracle Database through Web services. You can write and deploy Web services that can query the database using SQL or XQuery, or access stored PL/SQL functions and procedures. See Chapter 33, "Using Native Oracle XML DB Web Services"

In-Place XML Schema Evolution

In many cases, you can now evolve XML schemas without copying the corresponding XML instance documents. See Chapter 10, "XML Schema Evolution".

> **See Also:** *Oracle Database PL/SQL Packages and Types Reference* for updates to PL/SQL package `DBMS_XMLSCHEMA`

Support for Recursive XML Schemas

Oracle XML DB now performs XPath rewrite on some queries that use `'//'` in XPath expressions to target nodes at multiple or arbitrary depths, even when the XML data conforms to a recursive XML schema. See "Support for Recursive Schemas" on page 9-24

See Also: *Oracle Database PL/SQL Packages and Types Reference* for updates to PL/SQL package `DBMS_XMLSCHEMA`

Support for XLink and XInclude

Oracle XML DB now supports the XLink and XInclude standards. See Chapter 23, "Using XLink and XInclude with Oracle XML DB".

Support for XML Translations

You can now associate natural-language translation information with XML schemas and corresponding instance documents. This includes support for standard attributes `xml:lang` and `xml:srclang`. See "XML Translations" on page 7-17.

See Also: *Oracle Database PL/SQL Packages and Types Reference* for information about new PL/SQL package `DBMS_XMLTRANSLATIONS`

Support for Large XML Nodes

The previous 64K limit on text nodes and attribute values has been lifted. Text nodes and attribute values are no longer limited in size to 64K bytes each. New streaming push and pull APIs are available in PL/SQL, Java, and C to provide virtually unlimited node sizes. See "Large Node Handling using DBMS_XMLDOM" on page 13-12 for information about handling large nodes in PL/SQL and "Handling Large Nodes using Java" on page 15-16.

See Also:

- *Oracle Database SQL Language Reference* for information about creating `XMLType` tables and columns stored as binary XML
- *Oracle Database XML Java API Reference* for information about new Java methods
- *Oracle Database PL/SQL Packages and Types Reference* for information about new PL/SQL package `DBMS_SDA` and updates to PL/SQL package `DBMS_XMLDOM`

Unified Java API

The Java XML APIs in Oracle XML DB and Oracle XDK have been unified.

See Also:

- *Oracle XML Developer's Kit Programmer's Guide*
- *Oracle Database XML Java API Reference*, package `oracle.xml.parser.v2`

Oracle Data Pump Support for XMLType

Oracle Data Pump is now the recommended way to import and export `XMLType` data. See Chapter 36, "Exporting and Importing XMLType Tables".

Support for XMLType by Oracle Streams and Logical Standby

Oracle Streams and logical standby now support `XMLType` stored as `CLOB`. Both XML schema-based and non-schema-based XML data are supported.

See Also:

- *Oracle Streams Concepts and Administration*
- *Oracle Data Guard Concepts and Administration*
- *Oracle Database Utilities*
- *Oracle Database Reference* for information on views `DBA_STREAMS_UNSUPPORTED` and `DBA_STREAMS_COLUMNS`

Oracle XDK Pull-Parser API (XML Events, JSR-173)

You can use the new Oracle XML Developer Kit (XDK) pull-parser API with Oracle XML DB. See "Using the Oracle XDK Pull Parser with Oracle XML DB" on page 16-9.

See Also:

- *Oracle Database XML C API Reference* for information about new C methods and types
- *Oracle XML Developer's Kit Programmer's Guide*

XQuery Standard Compliance

Oracle XML DB support for the XQuery language has been updated to reflect the latest version of the XQuery standard, W3C XQuery 1.0 Recommendation.

See Also:

- *Oracle XML Developer's Kit Programmer's Guide*
- `http://www.w3.org` for information about the XQuery language

Fine-Grained Access to Network Services Using PL/SQL

New atomic privileges are provided for access control entries (ACEs). These privileges are used for fine-grained PL/SQL access to network services.

SQL/XML Standard Compliance and Performance Enhancements

Oracle XML DB support for the SQL/XML standard has been updated to reflect the latest version of the standard. This includes support for standard SQL functions `XMLExists` and `XMLCast`. See "Querying XMLType Data using SQL/XML Functions XMLExists and XMLCast" on page 4-2 and "Generating XML using SQL Functions" on page 18-2.

See Also: *Oracle Database SQL Language Reference* for information about SQL/XML functions `XMLExists`, `XMLCast`, `XMLQuery`, `XMLTable`, and `XMLForest`.

XML-Update Performance Enhancements

The performance of SQL functions used to update XML data has been enhanced for XML schema-based data that is stored object-relationally. This includes XPath rewrite for SQL functions `updateXML`, `insertChildXML`, and `deleteXML`.

XQuery and SQL/XML Performance Enhancements

XQuery and SQL/XML performance enhancements include treatment of the following:

- User-defined XQuery functions

- XQuery prolog variables
- XQuery `count` function applied to the result of using a SQL/XML generation function
- Positional expressions in XPath predicates
- XQuery computed constructors
- SQL/XML function `XMLAgg`

XSLT Performance Enhancements

The performance of XSLT transformations using SQL function `XMLTransform` and `XMLType` method `transform()` has been enhanced.

Part I

Oracle XML DB Basics

Part I of this manual introduces Oracle XML DB. It contains the following chapters:

- Chapter 1, "Introduction to Oracle XML DB"
- Chapter 2, "Getting Started with Oracle XML DB"
- Chapter 3, "Using Oracle XML DB"

1

Introduction to Oracle XML DB

This chapter introduces the features and architecture of Oracle XML DB. It contains these topics:

- Overview of Oracle XML DB
- Oracle XML DB Architecture
- Oracle XML DB Features
- Oracle XML DB Benefits
- Search XML Data using Oracle Text
- Build Messaging Applications using Oracle Streams Advanced Queuing
- Standards Supported by Oracle XML DB
- Oracle XML DB Technical Support
- Oracle XML DB Examples Used in This Manual
- Further Oracle XML DB Case Studies and Demonstrations

Overview of Oracle XML DB

Oracle XML DB is a set of Oracle Database technologies related to high-performance handling of XML data: storing, generating, accessing, searching, validating, transforming, evolving, and indexing. It provides native XML support by encompassing both the SQL and XML data models in an interoperable way. Oracle XML DB is included as part of Oracle Database starting with Oracle9*i* Release 2 (9.2).

Oracle XML DB includes the following features:

- An abstract SQL data type, `XMLType`, for XML data.
- Enterprise-level Oracle Database features for XML content: reliability, availability, scalability, and security. XML-specific memory management and optimizations.
- Industry-standard ways to access and update XML data. The standards include the SQL/XML standard and the World Wide Web Consortium (W3C) XML and XML Schema data models and recommendations for XPath and XQuery. You can use FTP, HTTP(S), and WebDAV to move XML content into and out of Oracle Database. Industry-standard APIs provide programmatic access and manipulation of XML content using Java, C, and PL/SQL.
- Ways to store, query, update, and transform XML data while accessing it using SQL.
- Ways to perform XML operations on SQL data.

- Oracle XML DB Repository: a simple, lightweight repository where you can organize and manage database content, including XML content, using a file/folder/URL metaphor.
- Ways to access and combine data from disparate systems through gateways, using a single, common data model. This reduces the complexity of developing applications that must deal with data from different stores.
- Ways to use Oracle XML DB in conjunction with Oracle XML Developer's Kit (XDK) to build applications that run in the middle tier in either Oracle Application Server or Oracle Database.

See Also:

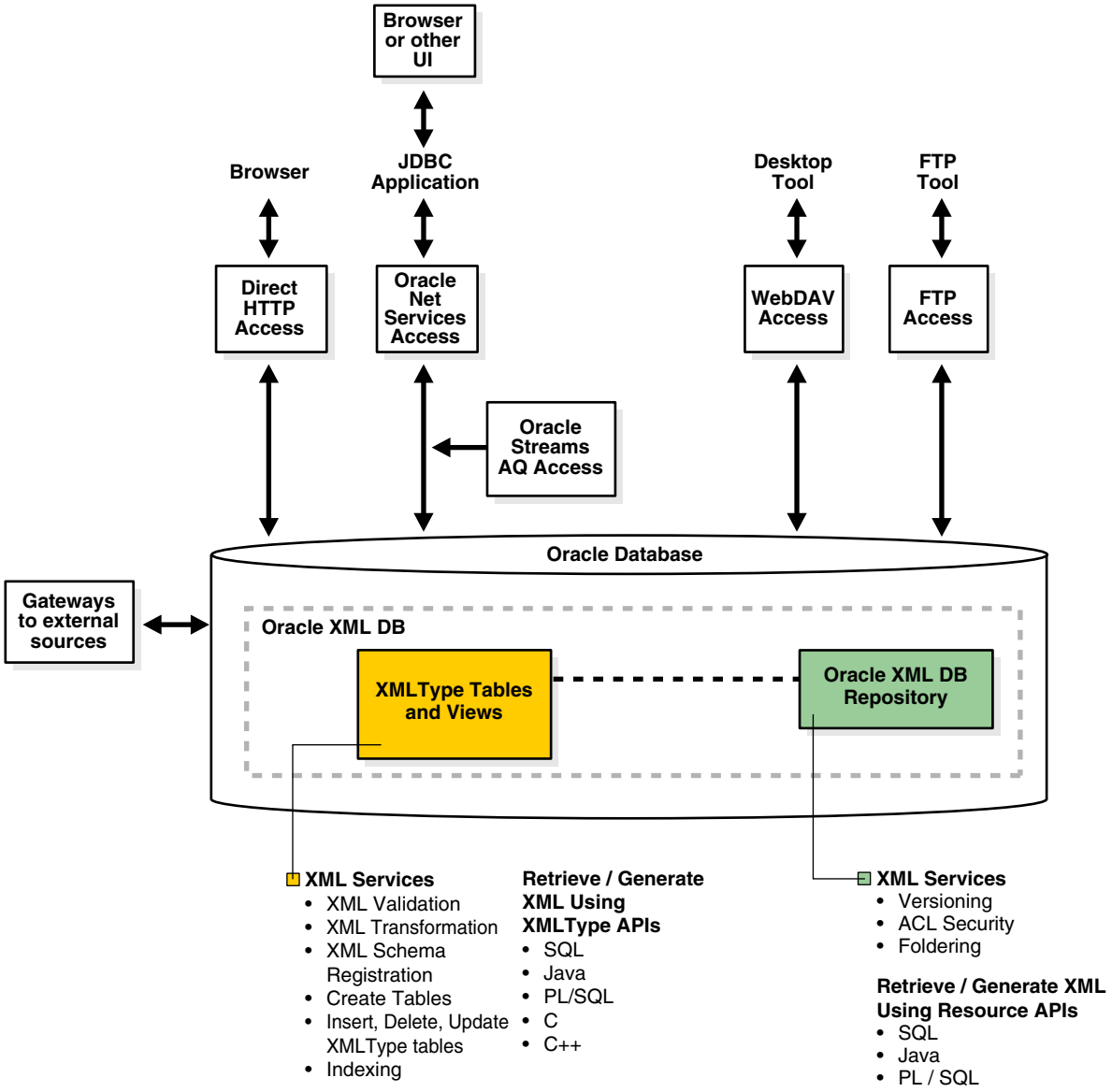
- "XMLType Data Type" on page 1-12
- `http://www.oracle.com/technology/tech/xml/` for the latest news and white papers about Oracle XML DB
- *Oracle XML Developer's Kit Programmer's Guide*

Oracle XML DB Architecture

Figure 1–1 and Figure 1–2 show the software architecture of Oracle XML DB. The main features are:

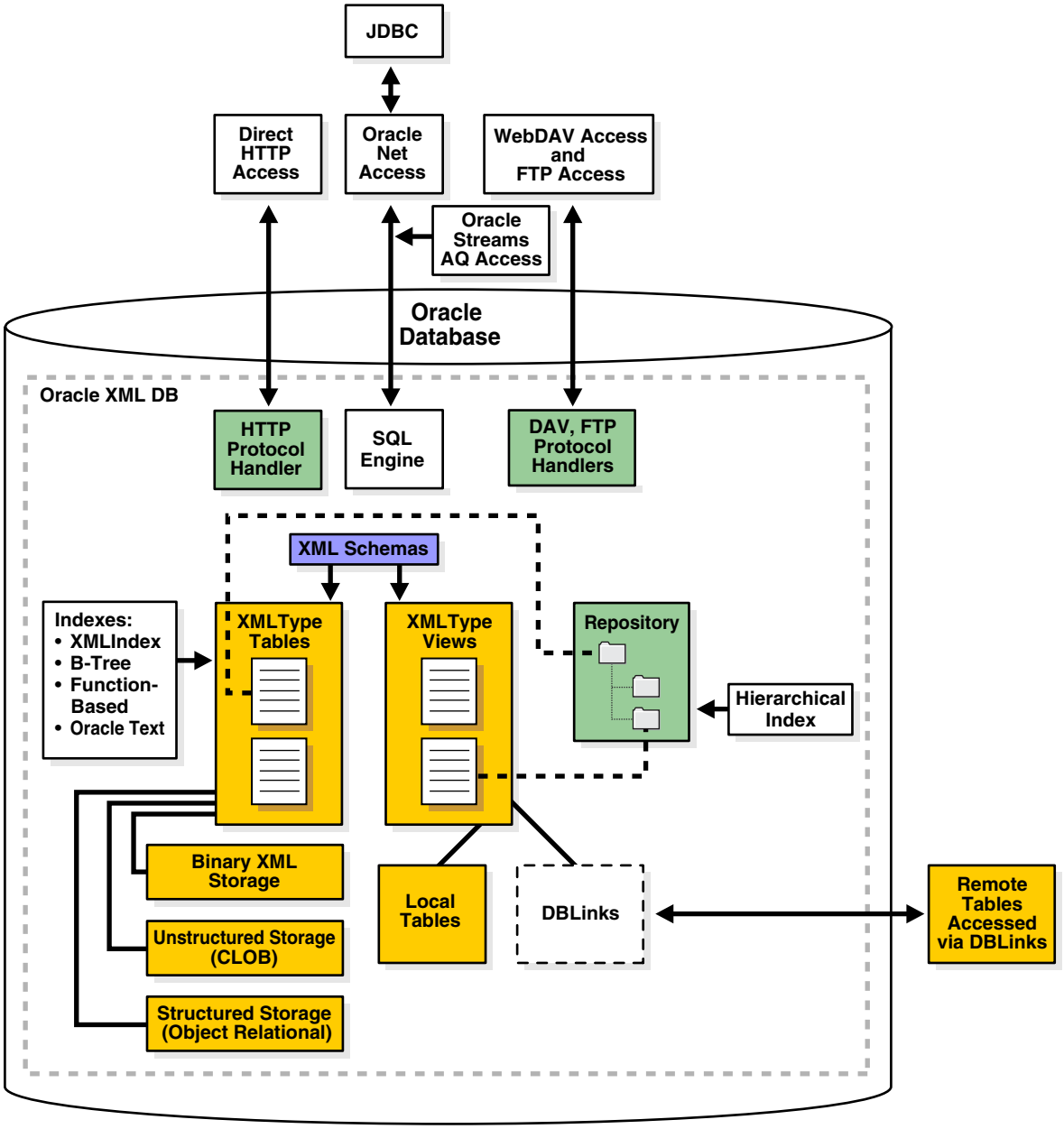
- Storage of `XMLType` tables and views.
  - You can index `XMLType` tables and views using `XMLIndex`, B-tree, and Oracle Text indexes.
  - You can store data that is in `XMLType` views in local or remote tables. You can access remote tables using database links.
- Oracle XML DB Repository. You can store any kind of documents in the repository, including XML documents that are associated with an XML schema that is registered with Oracle XML DB. You can access documents in the repository in any of the following ways:
  - HTTP(S), through the HTTP protocol handler
  - WebDAV and FTP, through the WebDAV and FTP protocol handlers
  - SQL, through Oracle Net Services, including Java Database Connectivity (JDBC)
- Support of XML data messaging using Oracle Streams Advanced Queuing (AQ) and Web Services.

Figure 1–1 XMLType Storage and Oracle XML DB Repository



****

Figure 1–2 XMLType Storage



See Also:

- Part II, "Storing and Retrieving XML Data in Oracle XML DB"
- Chapter 28, "Accessing the Repository using Protocols"
- Chapter 37, "Exchanging XML Data using Oracle Streams AQ"

APIs for XML

Table 1–1 lists the reference documentation for the PL/SQL, C, and C++ Application Programming Interfaces (APIs) that you can use to manipulate XML data. The main reference for PL/SQL, C, and C++ APIs is *Oracle Database PL/SQL Packages and Types Reference*.

See Also: *Oracle Database XML Java API Reference* for information about Java APIs for XML

Table 1–1 APIs Related to XML

API	Documentation	Description
`XMLType`	*Oracle Database PL/SQL Packages and Types Reference*, Chapter "XMLType", *Oracle Database XML C API Reference*, and *Oracle Database XML C++ API Reference*	PL/SQL, C, and C++ APIs with XML operations on `XMLType` data – validation, transformation.
Database URI types	*Oracle Database PL/SQL Packages and Types Reference*, Chapter "Database URI TYPEs"	Functions used for various URI types.
`DBMS_METADATA`	*Oracle Database PL/SQL Packages and Types Reference*, Chapter "DBMS_METADATA"	PL/SQL API for retrieving metadata from the database dictionary as XML, or retrieving creation DDL and submitting the XML to re-create the associated object.
`DBMS_RESCONFIG`	*Oracle Database PL/SQL Packages and Types Reference*, Chapter "DBMS_RESCONFIG"	PL/SQL API to operate on a resource configuration list, and to retrieve listener information for a resource.
`DBMS_XDB`	*Oracle Database PL/SQL Packages and Types Reference*, Chapter "DBMS_XDB"	PL/SQL API for managing Oracle XML DB Repository resources, ACL-based security, and configuration sessions.
`DBMS_XDB_ADMIN`	*Oracle Database PL/SQL Packages and Types Reference*, Chapter "DBMS_XDB_ADMIN"	PL/SQL API for managing miscellaneous features of Oracle XML DB, including the `XMLIndex` index on the Oracle XML DB Repository.
`DBMS_XDBRESOURCE`	*Oracle Database PL/SQL Packages and Types Reference*, Chapter "DBMS_XDBRESOURCE"	PL/SQL API to operate on repository resource metadata and contents.
`DBMS_XDBT`	*Oracle Database PL/SQL Packages and Types Reference*, Chapter "DBMS_XDBT"	PL/SQL API for creation of text indexes on repository resources.
`DBMS_XDB_VERSION`	*Oracle Database PL/SQL Packages and Types Reference*, Chapter "DBMS_XDB_VERSION"	PL/SQL API for version management of repository resources.
`DBMS_XDBZ`	*Oracle Database PL/SQL Packages and Types Reference*, Chapter "DBMS_XDBZ"	Oracle XML DB Repository ACL-based security.
`DBMS_XEVENT`	*Oracle Database PL/SQL Packages and Types Reference*, Chapter "DBMS_XEVENT"	PL/SQL API providing event-related types and supporting interface. .
`DBMS_XMLDOM`	*Oracle Database PL/SQL Packages and Types Reference*, Chapter "DBMS_XMLDOM"	PL/SQL implementation of the DOM API for `XMLType`.
`DBMS_XMLGEN`	*Oracle Database PL/SQL Packages and Types Reference*, Chapter "DBMS_XMLGEN"	PL/SQL API for transformation of SQL query results into canonical XML format.
`DBMS_XMLINDEX`	*Oracle Database PL/SQL Packages and Types Reference*, Chapter "DBMS_XMLINDEX	PL/SQL API for `XMLIndex`.
`DBMS_XMLPARSER`	*Oracle Database PL/SQL Packages and Types Reference*, Chapter "DBMS_XMLPARSER"	PL/SQL implementation of the DOM Parser API for `XMLType`.

Table 1–1 (Cont.) APIs Related to XML

API	Documentation	Description
DBMS_XMLQUERY	*Oracle Database PL/SQL Packages and Types Reference*, Chapter "DBMS_XMLQUERY"	PL/SQL API providing database-to-XMLType functionality. (Where possible, use DBMS_XMLGEN instead.)
DBMS_XMLSAVE	*Oracle Database PL/SQL Packages and Types Reference*, Chapter "DBMS_XMLSAVE"	PL/SQL API providing XML- to-database type functionality.
DBMS_XMLSCHEMA	*Oracle Database PL/SQL Packages and Types Reference*, Chapter "DBMS_XMLSCHEMA	PL/SQL API for managing XML schemas within Oracle Database – schema registration, deletion.
DBMS_XMLSTORE	*Oracle Database PL/SQL Packages and Types Reference*, Chapter "DBMS_XMLSTORE"	PL/SQL API for storing XML data in relational tables.
DBMS_XSLPROCESSOR	*Oracle Database PL/SQL Packages and Types Reference*, Chapter "DBMS_XSLPROCESSOR"	PL/SQL implementation of an XSLT processor.

Catalog Views Related to XML

Table 1–2 lists the catalog views related to XML. Information about a given view can be obtained by using SQL command DESCRIBE.

```
DESCRIBE USER_XML_SCHEMAS
```

Table 1–2 Catalog Views Related to XML

Schema	Description
USER_XML_SCHEMAS	Registered XML schemas *owned* by the current *user*
ALL_XML_SCHEMAS	Registered XML schemas *usable* by the current *user*
DBA_XML_SCHEMAS	Registered XML schemas in Oracle XML DB
USER_XML_TABLES	XMLType tables *owned* by the current *user*
ALL_XML_TABLES	XMLType tables *usable* by the current *user*
DBA_XML_TABLES	XMLType tables in Oracle XML DB
USER_XML_TAB_COLS	XMLType table columns *owned* by the current *user*
ALL_XML_TAB_COLS	XMLType table columns *usable* by the current *user*
DBA_XML_TAB_COLS	XMLType table columns in Oracle XML DB
USER_XML_VIEWS	XMLType views *owned* by the current *user*
ALL_XML_VIEWS	XMLType views *usable* by the current *user*
DBA_XML_VIEWS	XMLType views in Oracle XML DB
USER_XML_VIEW_COLS	XMLType view columns *owned* by the current *user*
ALL_XML_VIEW_COLS	XMLType view columns *usable* by the current *user*
DBA_XML_VIEW_COLS	XMLType view columns in Oracle XML DB

In addition to the views ALL_XML_TABLES, DBA_XML_TABLES, and USER_XML_TABLES, views ALL_OBJECT_TABLES, DBA_OBJECT_TABLES, and USER_OBJECT_TABLES provide tablespace and other storage information for XMLType data stored object-relationally.

See Also:

- *Oracle Database Reference*
- *Oracle Database PL/SQL Packages and Types Reference*

Overview of Oracle XML DB Repository

Oracle XML DB Repository is a component of Oracle Database that lets you handle XML data using a file/folder/URL metaphor. The repository contains **resources**, which can be either **folders** (directories, containers) or files.

A resource has these properties:

- It is identified by a *path* and *name*.
- It has *content* (data), which can be XML data but need not be.
- It has a set of **system-defined metadata** (properties), such as `Owner` and `CreationDate`, in addition to its content. Oracle XML DB uses this information to manage the resource.
- It might also have **user-defined metadata**. Like system-defined metadata, this is information that is not part of the content, but is associated with it.
- It has an associated **access control list** that determines who can access the resource, and for what operations.

Although Oracle XML DB Repository treats XML content specially, you can use the repository to store other kinds of data besides XML. You can use the repository to access any data that is stored in Oracle Database.

You can access data in the repository in the following ways (see Figure 1–1):

- SQL – Using views `RESOURCE_VIEW` and `PATH_VIEW`
- PL/SQL – Using package `DBML_XDB`
- Java – Using the Oracle XML DB resource API for Java

See Also:

- Part V. "Oracle XML DB Repository"
- Chapter 28, "Accessing the Repository using Protocols" for information about accessing XML data in `XMLType` tables and columns using external protocols
- Chapter 29, "User-Defined Repository Metadata"

XML Services

Besides providing APIs for accessing and manipulating repository data, Oracle XML DB provides APIs for the following repository services:

- Versioning – Oracle XML DB uses PL/SQL package `DBMS_XDB_VERSION` to version resources in Oracle XML DB Repository. Updating a resource creates a new version. Previous versions are retained. Versioning support is based on the IETF WebDAV standard.
- ACL Security – Repository security is based on access control lists (ACLs). Each resource has an associated ACL that lists the privileges required to use it in various ways. When a resource is accessed or manipulated, its ACL determines whether the requested operation is allowed. An ACL is an XML document that contains a set of access control entries (ACEs). Each ACE grants or revokes a set of

permissions to a particular user or group (database role). This access control mechanism is based on the WebDAV specification.

- Foldering – Oracle XML DB Repository manages a persistent hierarchy of folder (that is, directory) resources that contain other resources (files or folders). Oracle XML DB modules such as protocol servers, the XML schema manager, and the Oracle XML DB `RESOURCE_VIEW` API use foldering to map repository path names to the resources they target.

Views RESOURCE_VIEW and PATH_VIEW

Views `RESOURCE_VIEW` and `PATH_VIEW` provide SQL access to data in Oracle XML DB Repository through protocols such as FTP and WebDAV. View `PATH_VIEW` has one row for each unique path in the repository. View `RESOURCE_VIEW` has one row for each resource in the repository.

The Oracle XML DB resource API for PL/SQL, `DBMS_XDB`, provides query and DML functions. It is based on `RESOURCE_VIEW` and `PATH_VIEW`.

See Also:

- Chapter 25, "Accessing the Repository using RESOURCE_VIEW and PATH_VIEW"
- *Oracle Database Reference* for more information about view `PATH_VIEW`
- *Oracle Database Reference* for more information about view `RESOURCE_VIEW`

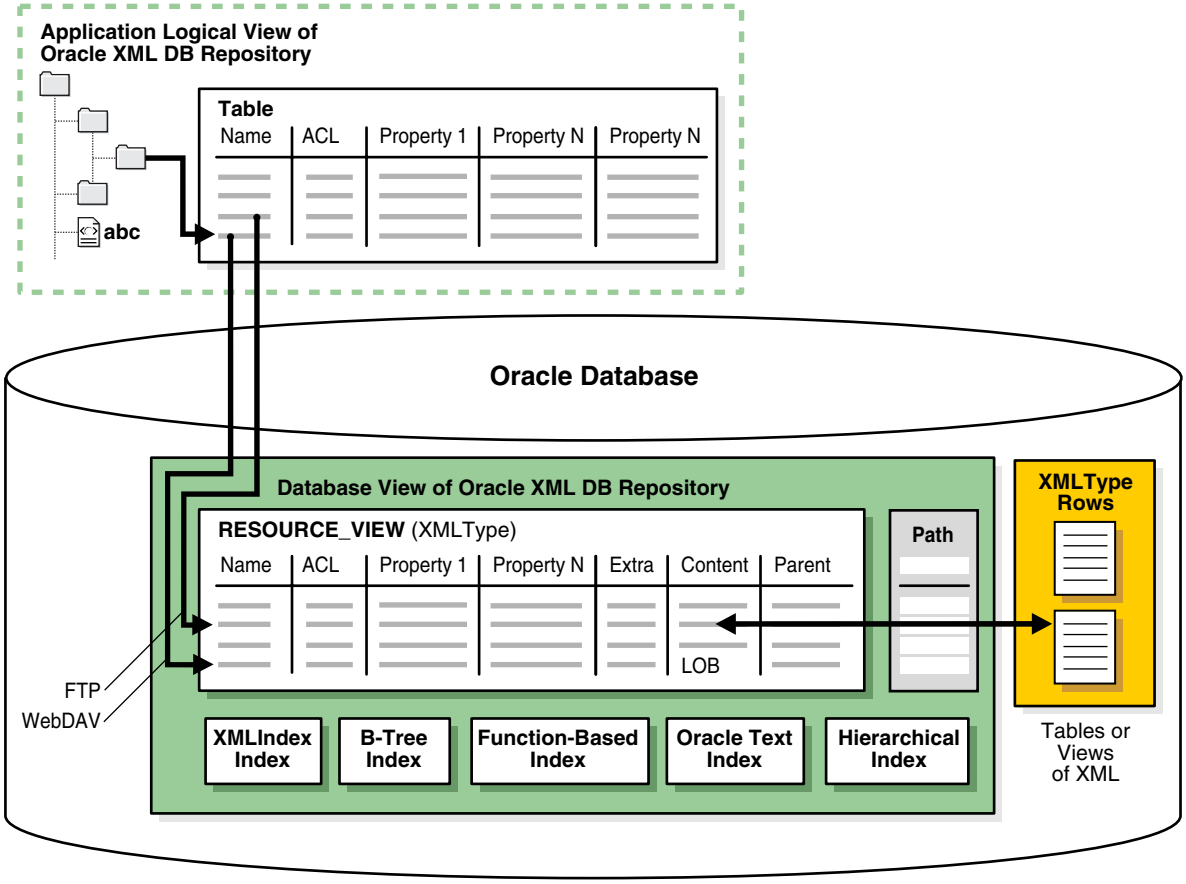
Oracle XML DB Repository Architecture

Figure 1–3 illustrates the architecture of Oracle XML DB Repository.

See Also:

- Chapter 21, "Accessing Oracle XML DB Repository Data"
- Chapter 25, "Accessing the Repository using RESOURCE_VIEW and PATH_VIEW"

Figure 1–3 Oracle XML DB Repository Architecture



e

****

Files and Folders

Relational databases are traditionally poor at managing hierarchical structures and traversing a path or a URL. Oracle XML DB Repository provides you with a hierarchical organization of XML content in the database. You can query and manage it as if it were organized using files and folders.

The relational table-row-column metaphor is an effective model for managing highly structured data. It can be less effective for managing semi-structured and unstructured data, such as document-oriented XML data.

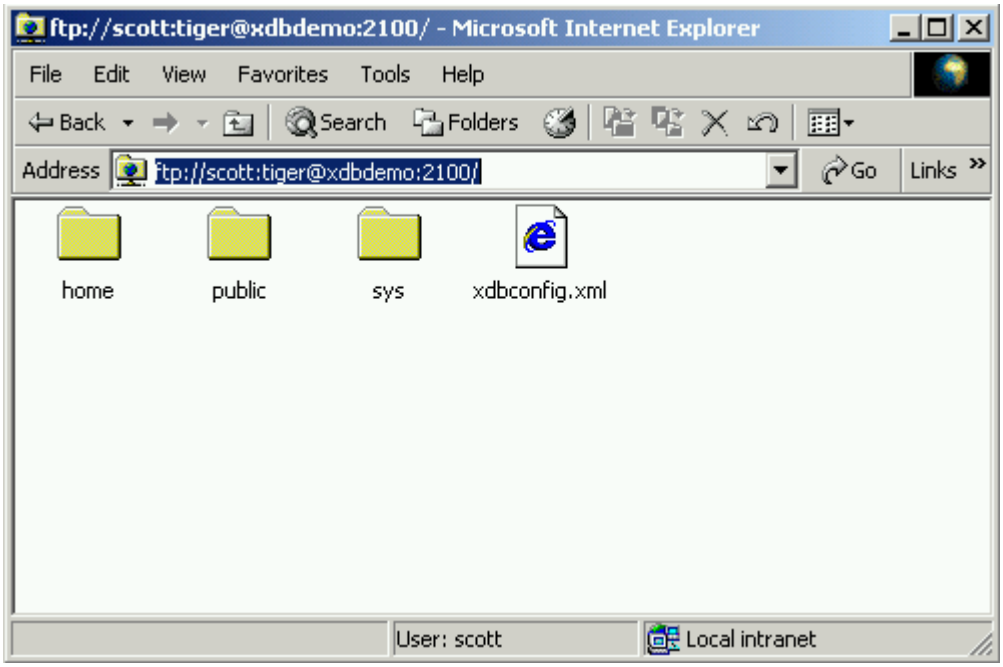
For example, a book is not easily represented as a set of rows in a table. It might be more natural to represent a book as a hierarchy, book—chapter—section—paragraph, and to represent the hierarchy as a set of folders and subfolders.

- A hierarchical repository index speeds up folder and path traversals. Oracle XML DB includes a patented hierarchical index that speeds up folder and path traversals in Oracle XML DB Repository. The hierarchical repository index is transparent to end users, and lets Oracle XML DB perform folder and path traversals at speeds comparable to or faster than conventional file systems.
- You can access XML documents in Oracle XML DB Repository using standard connect-access protocols such as FTP, HTTP(S), and WebDAV, in addition to

languages SQL, PL/SQL, Java, and C. The repository provides content authors and editors direct access to XML content stored in Oracle Database.

- A resource in this context is a file or folder, identified by a URL. WebDAV is an IETF standard that defines a set of extensions to the HTTP protocol. It lets an HTTP server act as a file server for a DAV-enabled client. For example, a WebDAV-enabled editor can interact with an HTTP/WebDAV server as if it were a file system. The WebDAV standard uses the term **resource** to describe a file or a folder. Each resource managed by a WebDAV server is identified by a URL. Oracle XML DB adds native support to Oracle Database for these protocols. The protocols were designed for document-centric operations. By providing support for these protocols, Oracle XML DB lets Microsoft Windows Explorer, Microsoft Office, and products from vendors such as Altova and Adobe work directly with XML content stored in Oracle XML DB Repository. Figure 1–4 shows the root-level directory of the repository as seen from a Web browser.

Figure 1–4 Web Browser View of Oracle XML DB Repository



See Also: Chapter 3, "Using Oracle XML DB"

Hence, WebDAV clients such as Microsoft Windows Explorer can connect directly to Oracle XML DB Repository. No additional Oracle Database or Microsoft-specific software or other complex middleware is needed. End users can work directly with Oracle XML DB Repository using familiar tools and interfaces.

Oracle XML DB Protocol Architecture

One key feature of the Oracle XML DB architecture is that protocols HTTP(S), WebDAV, and FTP are supported, including in a shared server configuration. When the Listener receives an HTTP(S) or FTP request, it hands it off to an Oracle Database shared server process which services it and sends the appropriate response back to the client.

You can use the TNS Listener command, `lsnrctl status`, to verify that HTTP(S) and FTP support has been enabled. Example 1–1 illustrates this.

Example 1–1 Listener Status with FTP and HTTP(S) Protocol Support Enabled

```
LSNRCTL for 32-bit Windows: Version 11.1.0.5.0 - Production on 20-AUG-2007 16:02:34

Copyright (c) 1991, 2007, Oracle.  All rights reserved.

Connecting to (DESCRIPTION=(ADDRESS=(PROTOCOL=IPC)(KEY=EXTPROC1521))) STATUS of the LISTENER
------------------------------------------------------------------------------------------
Alias                         LISTENER
Version                       TNSLSNR for 32-bit Windows: Version 11.1.0.5.0 - Beta
Start Date                    20-JUN-2007 15:35:40
Uptime                        0 days 16 hr. 47 min. 42 sec
Trace Level                   off
Security                      ON: Local OS Authentication
SNMP                          OFF
Listener Parameter File       C:\oracle\product\11.1.0\db_1\network\admin\listener.ora
Listener Log File             c:\oracle\diag\tnslsnr\quine-pc\listener\alert\log.xml

Listening Endpoints Summary...
(DESCRIPTION=(ADDRESS=(PROTOCOL=ipc)(PIPENAME=\\.\pipe\EXTPROC1521ipc)))
(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(HOST=quine-pc.example.com)(PORT=1521)))
(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(HOST=quine-pc.example.com)
             (PORT=21))(Presentation=FTP)(Session=RAW))
(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(HOST=quine-pc.example.com)
             (PORT=443))(Presentation=HTTP)(Session=RAW))
Services Summary...
Service "orcl.example.com" has 1 instance(s).
  Instance "orcl", status READY, has 1 handler(s) for this service...
Service "orclXDB.example.com" has 1 instance(s).
  Instance "orcl", status READY, has 1 handler(s) for this service...
Service "orcl_XPT.example.com" has 1 instance(s).
  Instance "orcl", status READY, has 1 handler(s) for this service...
The command completed successfully
```

See Also: Chapter 28, "Accessing the Repository using Protocols"

Programmatic Access to Oracle XML DB (Java, PL/SQL, and C)

All Oracle XML DB functionality is accessible from C, PL/SQL, and Java. You can build Web-based applications in various ways, including these:

- Using servlets and Java Server Pages (JSP). A typical API accesses data using Java Database Connectivity (JDBC).
- Using Extensible Stylesheet Language (XSL) plus XML Server Pages (XSP). A typical API accesses data in the form of XML documents that are processed using a Document Object Model (DOM) API implementation.

Oracle XML DB supports such styles of application development. It provides Java, PL/SQL, and C implementations of the DOM API.

Applications that use JDBC, such as those based on servlets, need prior knowledge of the data structure they are processing. Oracle JDBC drivers allow you to access and update `XMLType` tables and columns, and call PL/SQL procedures that access Oracle XML DB Repository.

Applications that use DOM, such as those based on XSLT transformations, typically require less knowledge of the data structure. DOM-based applications use string names to identify pieces of content, and must dynamically walk through the DOM tree to find the required information. For this, Oracle XML DB supports the use of the DOM API to access and update `XMLType` columns and tables. Programming to a DOM

API is more flexible than programming through JDBC, but it may require more resources at run time.

Oracle XML DB Features

Any database used for managing XML must be able to persist XML documents. Oracle XML DB is capable of much more than this. It provides standard database features such as transaction control, data integrity, replication, reliability, availability, security, and scalability, while also allowing for efficient indexing, querying, updating, and searching of XML documents in an XML-centric manner.

The hierarchical nature of XML presents the traditional relational database with some challenges:

- In a relational database, the *table-row* metaphor locates content. Primary-Key Foreign-Key relationships help define the relationships between content. Content is accessed and updated using the table-row-column metaphor.
- XML, on the other hand, uses *hierarchical* techniques to achieve the same functionality. A URL is used to locate an XML document. URL-based standards such as XLink are used to define relationships between XML documents. W3C Recommendations such as XPath are used to access and update content contained within XML documents. Both URLs and XPath expressions are based on *hierarchical* metaphors. A URL uses a path through a *folder hierarchy* to identify a document, whereas XPath uses a path through the *node hierarchy* of an XML document to access part of an XML document.

Oracle XML DB addresses these challenges by introducing SQL functions and methods that allow the use of XML-centric metaphors, such as XQuery and XPath expressions for querying and updating XML Documents.

These are the major features of Oracle XML DB:

- XMLType Data Type
- XML Schema Support
- XMLType Storage Models
- XML/SQL Duality
- SQL/XML Standard Functions
- Automatic Rewriting of XQuery and XPath Expressions
- Overview of Oracle XML DB Repository. This was described on page 1-7.

XMLType Data Type

`XMLType` is an abstract native SQL data type for XML data. It provides methods that allow operations such as XML Schema validation and XSL transformation of XML content. You can use `XMLType` as you would any other SQL data type. For example, you can use `XMLType` when you do any of the following:

- Create a column in a relational table
- Declare a PL/SQL variable
- Define or call a PL/SQL procedure or function

`XMLType` is an Oracle Database *object* type, so you can also create a table of `XMLType` object instances. By default, an `XMLType` table or column can contain any well-formed XML document.

See Also: *Oracle Database Object-Relational Developer's Guide* for information about Oracle Database object types and object-relational storage

XMLType Tables and Columns Can Conform to an XML Schema

`XMLType` tables or columns can be constrained to conform to an XML schema. This has several advantages:

- The database ensures that only XML documents that validate against the XML schema are stored in the column or table. Invalid documents are excluded.
- Because XML schema-based data conforms to a predefined XML structure, Oracle XML DB can use the information contained in the XML schema to optimize querying and updating of the data.
- If you store XML schema-based data using structured storage, Oracle XML DB automatically decomposes it and stores it as a set of object-relational objects. The object-relational model used to store the document is derived from the XML schema.

XMLType API

Data type `XMLType` provides the following:

- *Constructors*, which you can use to create an `XMLType` instance from a `VARCHAR`, `CLOB`, `BLOB`, or `BFILE` value.
- XML-specific *methods* that operate on `XMLType` instances. These include the following:
  - `extract()`– Extract a subset of nodes contained in the `XMLType` instance.
  - `existsNode()` – Check whether or not a particular node exists in the `XMLType` instance.
  - `schemaValidate()` – Validate the content of the `XMLType` instance against an XML schema.
  - `transform()` – Perform an XSL transformation on the content of an `XMLType` instance.

See Also: Chapter 4, "XMLType Operations" and Chapter 11, "Transforming and Validating XMLType Data"

XML Schema Support

Support for the Worldwide Web Consortium (W3C) XML Schema Recommendation is a key feature in Oracle XML DB. XML Schema specifies the structure, content, and certain semantics of XML documents. It is described in detail at `http://www.w3.org/TR/xmlschema-0/`.

The W3C Schema Working Group publishes a particular XML schema, often referred to as the **schema for schemas**, that provides the definition, or vocabulary, of the XML Schema language. An **XML schema definition** (**XSD**[1]), also called an **XML schema**, is an XML document that is compliant with the vocabulary defined by the schema for schemas.

[1] `xsd` is the prefix used in the schema of schemas for the XML Schema namespace, hence it is also the namespace prefix used for the XML Schema data types, such as `xsd:string`. `xsd` is also used often as the file extension of XML schema files.

An XML schema uses vocabulary defined by the schema for schemas to create a collection of XML Schema type definitions and element declarations that comprise a vocabulary for describing the contents and structure of a new class of XML documents, the **XML instance documents** that conform to that XML schema.

> **Note:** This manual uses the term "XML schema" (lower-case "s") to reference any XML schema that conforms to the W3C XML Schema (upper-case "S") Recommendation. Since an XML schema is used to define a class of XML documents, the term "instance document" is often used to describe an XML document that conforms to a particular XML schema.

The XML Schema language provides strong typing of elements and attributes. It defines numerous scalar data types. This base set of data types can be extended to define more complex types, using object-oriented techniques such as inheritance and extension. The XML Schema vocabulary also includes constructs that you can use to define complex types, substitution groups, repeating sets, nesting, ordering, and so on. Oracle XML DB supports all of the constructs defined by the XML Schema Recommendation, except for `redefines`.

XML schemas are commonly used as a mechanism for checking (validating) whether XML instance documents conform with their specifications. Oracle XML DB includes `XMLType` methods and SQL functions that you can use to validate XML documents against an XML schema.

In Oracle XML DB, you can use a standard data model for all of your data, regardless of how structured it is. You can use XML Schema to automatically create database tables for storing your XML data. XML schema-based data maintains DOM fidelity and allows for significant database optimizations.

XML schema-based data can be stored using any Oracle XML DB `XMLType` storage model: binary XML storage, structured (object-relational) storage, or unstructured (`CLOB`) storage. Non-schema-based XML data can be stored using binary XML storage or unstructured storage.

You can also wrap existing relational and object-relational data as `XMLType` views, which can optionally be XML schema-based. You can map from incoming XML documents to `XMLType` storage, specifying the mapping using a registered XML schema.

> **See Also:** Chapter 7, "XML Schema Storage and Query: Basic" for more information about using XML schemas with Oracle XML DB

XMLType Storage Models

`XMLType` is an *abstract* data type that provides different *storage models* to best fit your data and your use of it. As an abstract data type, your applications and database queries gain in flexibility: the same interface is available for all `XMLType` operations. Because different storage (persistence) models are available, you can tailor performance and functionality to best fit the kind of XML data you have and the pattern of its use. One key decision to make when using Oracle XML DB for persisting XML data as `XMLType` is thus which storage model to use for which XML data.

You can change `XMLType` storage from one model to another, using database import/export (see Chapter 36, "Exporting and Importing XMLType Tables"). Your application code need not change. You can change XML storage options when tuning your application.

`XMLType` tables and columns can be stored in these ways:

- **Structured** storage – `XMLType` data is stored as a set of objects. This is also referred to as **object-relational** storage and **object-based persistence**.
- **Unstructured** storage – `XMLType` data is stored in Character Large Object (`CLOB`) instances. This is also referred to as **CLOB** storage and **text-based persistence**.
- **Binary XML** storage – `XMLType` data is stored in a post-parse, binary format specifically designed for XML data. Binary XML is compact, post-parse, XML schema-aware XML. This is also referred to as **post-parse persistence**.

Oracle Database provides two *LOB storage options*, SecureFile and BasicFile. Either of these can be used with unstructured (CLOB-based) `XMLType` storage. BasicFile LOB storage is the default for unstructured storage.

For binary XML data, SecureFile is the default storage option.[2] However, if either of the following is true then it is not possible to use SecureFile LOB storage. In that case, BasicFile is the default option for binary XML data:

- The tablespace for the `XMLType` table does not use automatic segment space management.
- A setting in file `init.ora` prevents SecureFile LOB storage. For example, see parameter `DB_SECUREFILE`.

See Also:

- *Oracle Database SQL Language Reference*, section "CREATE TABLE", clause "LOB_storage_clause"
- *Oracle Database SecureFiles and Large Objects Developer's Guide* for information about LOB storage options SecureFile and BasicFile
- *Oracle Database Administrator's Guide* for information about automatic segment space management
- *Oracle Database Reference* for information about parameter `DB_SECUREFILE`

You can mix storage models, using one model for one part of an XML document and a different model for another part. The mixture of structured and unstructured storage is sometimes called **hybrid** storage. What is true about structured storage is true about the structured part of hybrid storage. What is true about unstructured storage is true about the unstructured part of hybrid storage.

`XMLType` has multiple storage models, and some models can be configured in more than one way. Each model has its advantages, depending on the context. Each model has one or more types of index that are appropriate for it.

The first thing to consider, when choosing an `XMLType` storage model, is the nature of your XML data and the ways you use it. A spectrum exists, with data-centric use of highly structured data at one end, and document-centric use of highly unstructured data at the other. The first question to ask yourself is this: *Is your use case primarily data-centric or document-centric?*

- **Data-centric** – Your data is, in general, highly structured, with relatively static and predictable structure, and your applications take advantage of this structure. Your data conforms to an XML schema.

[2] Prior to Oracle Database 11*g* Release 2 (11.2.0.2) the BasicFile option was the default for binary XML storage. Use of the BasicFile option for binary XML data is deprecated.

- **Document-centric** – Two cases:
  - Your data is generally without structure or of variable structure. Document structure can vary over time (evolution). Content is **mixed** (**semi-structured**): many elements contain both text nodes and child elements. Many XML elements can be absent or can appear in different orders. Documents might or might not conform to an XML schema.
  - Your data relatively structured, but your applications do not take advantage of that structure: they treat the data as if it were without structure.

    Note: Please be aware of the context, so as not to confuse discussion of storage options with discussion of the structure of the XML content to be stored. In this book, "structured" and "unstructured" generally refer to `XMLType` storage options. They refer less often to the nature of your data. "Hybrid" refers to object-relational storage with some embedded `CLOB` storage. "Semi-structured" refers to XML content, regardless of storage. Unstructured storage is `CLOB`-based storage, and structured storage is object-relational storage.

Once you've located the data-centric or document-centric half of the spectrum that is appropriate for your use case and data, consider whether your case is at an end of the spectrum or closer to the middle. That is, just how data-centric or document-centric is your case?

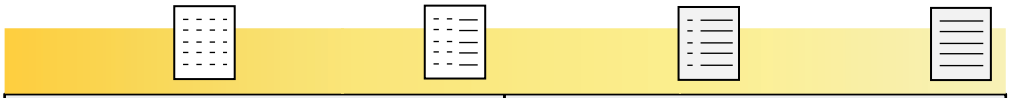
- Employ *object-relational* (structured) storage for purely data-centric uses. A typical example of this use case would be an employee record (fields employee number, name, address, and so on). Use B-tree indexing with object-relational storage.
- Employ *hybrid* storage if your data is composed primarily of invariable XML structures, but it does contain some variable data; that is, it contains a predictably few mixed-content elements. A typical example of this use case would be an employee record that includes a free-form resume. Index the structured and unstructured parts of your data separately, using appropriate indexes for each part.
- Employ *binary XML* storage or `CLOB`-*based* (unstructured) storage for all document-centric use cases. `XMLIndex` is the indexing method of choice here.
  - For general indexing of document-centric XML data, use `XMLIndex` indexes with unstructured components. A typical example of this use case would be an XML Web document or a book chapter.
  - If your data contains some predictable, fixed structures that you query frequently, then you can use `XMLIndex` indexes with structured components on those parts. A typical example of this use case would be a free-form specification, with author, date, and title fields.

  A single `XMLIndex` index can have both structured and unstructured components, to handle islands of structure within generally unstructured content. A use case where you might use both components would be to support queries that extract an XML fragment from a document whenever some structured data is present. The unstructured index component is used for the fragment extraction. The structured component is used for the SQL `WHERE` clause that checks for the structured data.

In all cases, you can additionally use Oracle Text indexing for full-text queries. This is especially useful for document-centric cases.

These considerations are summarized in Figure 1–5. The figure shows the spectrum of use cases, from most data-centric, at the left, to most document-centric, at the right. The table in the figure classifies use cases and shows the corresponding storage models and indexing methods.

Figure 1–5 XML Use Cases and XMLType Storage Models



	Data-Centric		Document-Centric	
Use Case	XML schema-based data, with little variation and little structural change over time	XML schema-based data, with some embedded variable data	Variable, free-form data, with some fixed embedded structures	Variable, free-form data
Typical Data	Employee record	Employee record that includes a free-form resume	Technical article, with author, date, and title fields	Web document or book chapter
Storage Model	Object-Relational (Structured)	Hybrid	`CLOB` (Unstructured) or Binary XML	
Indexing	B-tree index	• B-tree index • `XMLIndex` index with unstructured component	`XMLIndex` index with structured and unstructured components	`XMLIndex` index with unstructured component

****

See Chapter 6, "Indexing XMLType Data" for more information about indexing XML data. In particular, note that some types of indexing are complementary or orthogonal, so you can use them together.

The following list and Table 1–3 outline some of the advantages of each storage model.

- *Structured* (object-relational) storage advantages over the other storage models include near-relational query and update performance, optimized memory management, reduced storage requirements, B-tree indexing, and in-place updates. These advantages are at a cost of increased processing overhead during ingestion and full retrieval of XML data, and reduced flexibility in the structure of the XML that can be managed by a given `XMLType` table or column. Structural flexibility is reduced because data and metadata (such as column names) are separated in object-relational storage. Instance structures cannot vary easily. Structured storage is particularly appropriate for highly structured data whose structure does not vary, if this maps to a manageable number of database tables and joins.
- *Unstructured* (`CLOB`) storage enables higher throughput than structured storage when inserting and retrieving entire XML documents. No data conversion is needed, so the same format can be used outside the database. Unstructured storage also provides greater flexibility than structured storage in the structure of the XML that can be stored. Unstructured storage is particularly appropriate for document-centric use cases. These advantages can come at the expense of certain aspects of intelligent processing: in the absence of indexing, there is little that the database can do to optimize queries or updates on XML data that is stored in a `CLOB` instance. In particular, the cost of XML parsing (often implicit) can significantly impact query performance. Indexing with `XMLIndex` can improve the performance of queries against unstructured storage.

- *Binary XML* storage provides more efficient database storage, updating, indexing, and fragment extraction than unstructured storage. It can provide better query performance than unstructured storage—it does not suffer from the XML parsing bottleneck (it is a post-parse persistence model). Like structured storage, binary XML storage is aware of XML Schema data types and can take advantage of native database data types. Like structured storage, binary XML storage allows for piecewise updates. Because binary XML data can also be used outside the database, it can serve as an efficient XML exchange medium, and you can off load work from the database to increase overall performance in many cases. Like unstructured storage, binary XML data is kept in document order. Like structured storage, data and metadata can, using binary storage, be separated at the database level, for efficiency. Like unstructured storage, however, binary XML storage allows for intermixed data and metadata, which lets instance structures vary. Binary XML storage allows for very complex and variable data, which in the structured-storage model could necessitate using many database tables. Unlike the other `XMLType` storage models, you can use binary storage for XML schema-based data even if the XML schema is not known beforehand, and you can store multiple XML schemas in the same table and query across common elements.

Table 1–3 XMLType Storage Models: Relative Advantages

Quality	Structured (Object-Relational) Storage	Binary XML Storage	Unstructured (CLOB) Storage
Throughput	– XML decomposition can result in reduced throughput when ingesting or retrieving the entire content of an XML document.	+ High throughput. Fast DOM loading. There is a slight overhead from the binary XML encoder/decoder.	++ High throughput when ingesting and retrieving the entire content of an XML document.
Queries	++ Extremely fast: relational query performance. You can create B-tree indexes on the underlying object-relational columns.	+ Streaming XPath evaluation avoids DOM construction and allows evaluation of multiple XPath expressions in a single pass. Navigational XPath evaluation is significantly faster than with unstructured storage. `XMLIndex` indexing can improve performance of XPath-based queries.	– XPath operations are evaluated by constructing a DOM from the `CLOB` data and using functional evaluation. Expensive when performing operations on large documents or large collections of documents. `XMLIndex` indexing can improve performance of XPath-based queries.
Update operations (DML)	++ Extremely fast: relational columns are updated in place.	+ In-place, piecewise update for SecureFile LOB storage.	– When any part of the document is updated, the entire document must be written back to disk.
Space efficiency (disk)	++ Extremely space-efficient.	+ Space-efficient.	– Consumes the most disk space, due to insignificant whitespace and repeated tags.
Data flexibility	– Limited flexibility. Only documents that conform to the XML schema can be stored in the `XMLType` table or column.	+ Flexibility in the structure of the XML documents that can be stored in an `XMLType` column or table.	+ Flexibility in the structure of the XML documents that can be stored in an `XMLType` column or table.
XML schema flexibility	– An `XMLType` table can only store documents that conform to the same XML schema. In-place XML schema evolution is available, with some restrictions.	++ Can store XML schema-based or non-schema-based documents. An `XMLType` table can store documents that conform to any registered XML schemas.	++ Can store XML schema-based or non-schema-based documents. Cannot use multiple XML schemas for the same `XMLType` table.
XML fidelity	DOM fidelity: A DOM created from an XML document that has been stored in the database is identical to a DOM created from the original document. However, insignificant whitespace may be discarded.	DOM fidelity (see structured storage description).	+ Document fidelity: Maintains the original XML data, byte for byte. In particular, all original whitespace is preserved.

Table 1–3 (Cont.) XMLType Storage Models: Relative Advantages

Quality	Structured (Object-Relational) Storage	Binary XML Storage	Unstructured (CLOB) Storage
Indexing support	+ B-tree, bitmap, Oracle Text, `XMLIndex`, and function-based indexes.	`XMLIndex`, function-based, and Oracle Text indexes.	`XMLIndex`, function-based, and Oracle Text indexes.
Optimized memory management	+ XML operations can be optimized to reduce memory requirements.	+ XML operations can be optimized to reduce memory requirements.	– XML operations on the document require creating a DOM from the document.
Validation upon insert	XML data is partially validated when it is inserted.	+ XML schema-based data is fully validated when it is inserted.	XML schema-based data is partially validated when it is inserted.
Partitioning	+ Available.[1]	Partition based on virtual columns.	`XMLType` columns can be partitioned if the partitioning key is a relational column.
Streams-based replication	– Not available.	– Not available.	++ Available.
Compression	++ XML elements and attributes can be compressed individually.	+ XML data that uses SecureFile LOB storage can be compressed.	– Not available.

[1] Partitioning of ordered collection tables (OCTs) reflects the partitioning of the top-level `XMLType` tables. Partition maintenance operations on the top-level tables are cascaded to the associated OCTs. See "Partitioning XMLType Tables and Columns Stored Object-Relationally" on page 9-10.

> **Note:** When you insert XML schema-based data into binary `XMLType` columns or tables, the data is fully *validated* against the XML schema. Insertion fails if the data is invalid.

When `XMLType` is stored object-relationally, the `XMLType` instances contain hidden columns that store information about the XML data that does not fit into the SQL object model.

XML/SQL Duality

A key objective of Oracle XML DB is to provide XML/SQL duality. XML programmers can leverage the power of the relational model when working with XML content and SQL programmers can leverage the flexibility of XML when working with relational content. This lets you use the most appropriate tools for a particular business problem.

XML/SQL duality means that the same data can be exposed as rows in a table and manipulated using SQL or exposed as nodes in an XML document and manipulated using techniques such as DOM and XSL transformation. Access and processing techniques are independent of the underlying storage format.

These features provide simple solutions to common business problems. For example:

- You can use Oracle XML DB SQL functions to generate XML data directly from a SQL query. You can then transform the XML data into other formats, such as HTML, using the database-resident XSLT processor.
- You can access XML content without converting between different data formats, using SQL queries, on-line analytical processing (OLAP), and business-intelligence/data warehousing operations.
- You can perform text, spatial data, and multimedia operations on XML content.

SQL/XML Standard Functions

Oracle XML DB provides the SQL functions that are defined in the SQL/XML standard. SQL/XML functions fall into two groups:

- Functions that you can use to *generate* XML data from the result of a SQL query. In this book, these are called **SQL/XML publishing functions**. They are also sometimes called **SQL/XML generation functions**.
- Functions that you can use to *query* and *access* XML content as part of normal SQL operations. In this book, these are called **SQL/XML query and access functions**.

Using SQL/XML functions you can address XML content in any part of a SQL statement. These functions use XQuery or XPath expressions to traverse the XML structure and identify the nodes on which to operate. The ability to embed XQuery and XPath expressions in SQL statements greatly simplifies XML access.

See Also:

- *Oracle Database SQL Language Reference* for information about Oracle support for the SQL/XML standard
- Chapter 4, "XMLType Operations" and Chapter 5, "Using XQuery with Oracle XML DB" for detailed descriptions of the SQL/XML standard functions for *querying* XML data
- Generating XML using SQL Functions on page 18-2 for information about SQL/XML standard functions for *generating* XML data
- Chapter 3, "Using Oracle XML DB" for additional examples that use SQL/XML standard functions
- "Standards Supported by Oracle XML DB" on page 1-28

Automatic Rewriting of XQuery and XPath Expressions

SQL/XML functions and `XMLType` methods use XQuery or XPath expressions to search collections of XML documents and to access a subset of the nodes contained within an XML document. In many cases, Oracle XML DB is able to automatically rewrite such expressions to code that executes directly against the underlying database objects.

How XPath Expressions Are Evaluated by Oracle XML DB

Oracle XML DB provides the following ways of evaluating XPath expressions that operate on `XMLType` columns and tables, depending on the XML storage method used:

- Structured storage – Oracle XML DB attempts to translate the XPath expression in a SQL/XML function into an equivalent SQL query. The SQL query references the object-relational data structures that underpin a schema-based `XMLType`. This process is referred to as **XPath rewrite**. It can occur when performing queries and `UPDATE` operations. In addition, B-tree indexes on the underlying object-relational tables can be used to evaluate XPath expressions for structured storage.
- Unstructured storage – `XMLIndex` indexes can be used to evaluate XPath expressions for unstructured storage. Use `XMLIndex` by preference.
  - If an `XMLIndex` index can be used, then it is used instead of functional evaluation.

  - In the absence of such an index, Oracle XML DB evaluates the XPath expression using functional evaluation. Functional evaluation builds a DOM tree for each XML document, and then resolves the XPath programmatically using the methods provided by the DOM API. If the operation involves updating the DOM tree, the entire XML document must be written back to disk when the operation is completed.

- Binary XML storage – Oracle XML DB can evaluate XPath expressions in different ways: using `XMLIndex` and using single-pass streaming. Single-pass streaming means evaluating a set of XPath expressions in a single scan of binary XML data. During query compilation, the cost-based optimizer picks the fastest combination of evaluation methods.

  See Also: Table 1–3, " XMLType Storage Models: Relative Advantages"

Rewriting SQL Code That Contains XQuery and XPath Expressions

For XML data that is stored object-relationally, Oracle XML DB can rewrite SQL statements that contain XQuery and XPath expressions to purely relational SQL statements, which are then processed in an optimal manner. In this way, Oracle XML DB insulates the database optimizer from needing to understand the XQuery and XPath languages and the XML data model. The database optimizer processes a rewritten SQL statement the same way it processes other SQL statements. The general term applied to this rewriting process is **XPath rewrite**.

The database optimizer can thus derive an execution plan based on conventional relational algebra. This in turn means that Oracle XML DB can leverage all of the features of the database, and ensure that SQL statements containing XQuery and XPath expressions are executed in a highly performant and efficient manner. There is little overhead with this rewriting, and Oracle XML DB executes XQuery-based and XPath-based queries at near-relational speed, while preserving the XML abstraction.

When Can XPath Rewrite Occur?

XPath rewrite is possible when all of the following conditions are met:

- An `XMLType` column or table uses structured storage techniques to provide the underlying storage model.
- An `XMLType` column or table is associated with a registered XML schema.
- A SQL statement contains SQL/XML functions or `XMLType` methods that use XPath expressions to refer to one or more nodes within a set of XML documents.
- The nodes referenced by an XPath expression can be mapped, using the XML schema, to attributes of the underlying SQL object model.

What is the XPath-Rewrite Process?

XPath rewrite performs the following tasks:

1. Identify the set of XPath expressions included in the SQL statement.
2. Translate each XPath expression into an object relational SQL expression that references the tables, types, and attributes of the underlying SQL: 1999 object model.
3. Rewrite the original SQL statement into an equivalent object relational SQL statement.

4. Pass the new SQL statement to the database optimizer for plan generation and query execution.

In certain cases, XPath rewrite is not possible. This normally occurs when there is no SQL equivalent of the XPath expression. In this situation, Oracle XML DB performs a functional evaluation of the XPath expressions.

In general, functional evaluation of a SQL statement is more expensive than XPath rewrite, particularly if the number of documents to be processed is large. The advantage of functional evaluation is that it is always possible, regardless of whether the `XMLType` data is stored using structured storage and regardless of the complexity of the XPath expression.

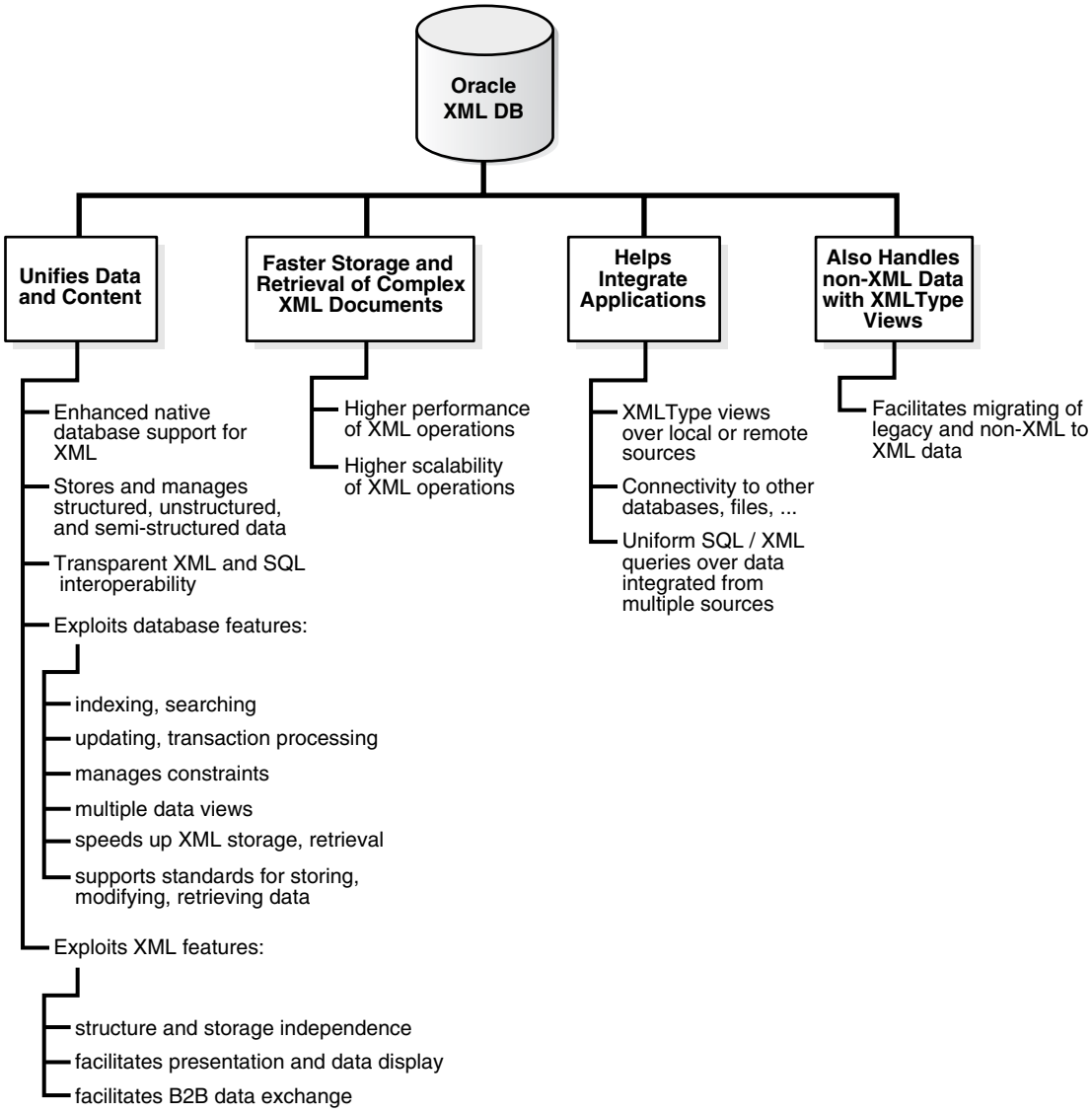
Understanding the concept of XPath rewrite, and the conditions under which it can take place, is a key step in developing Oracle XML DB applications that deliver the required levels of scalability and performance.

See Also: Chapter 8, "XPath Rewrite for Structured Storage"

Oracle XML DB Benefits

This section describes several benefits of using Oracle XML DB. Figure 1–6 presents an overview.

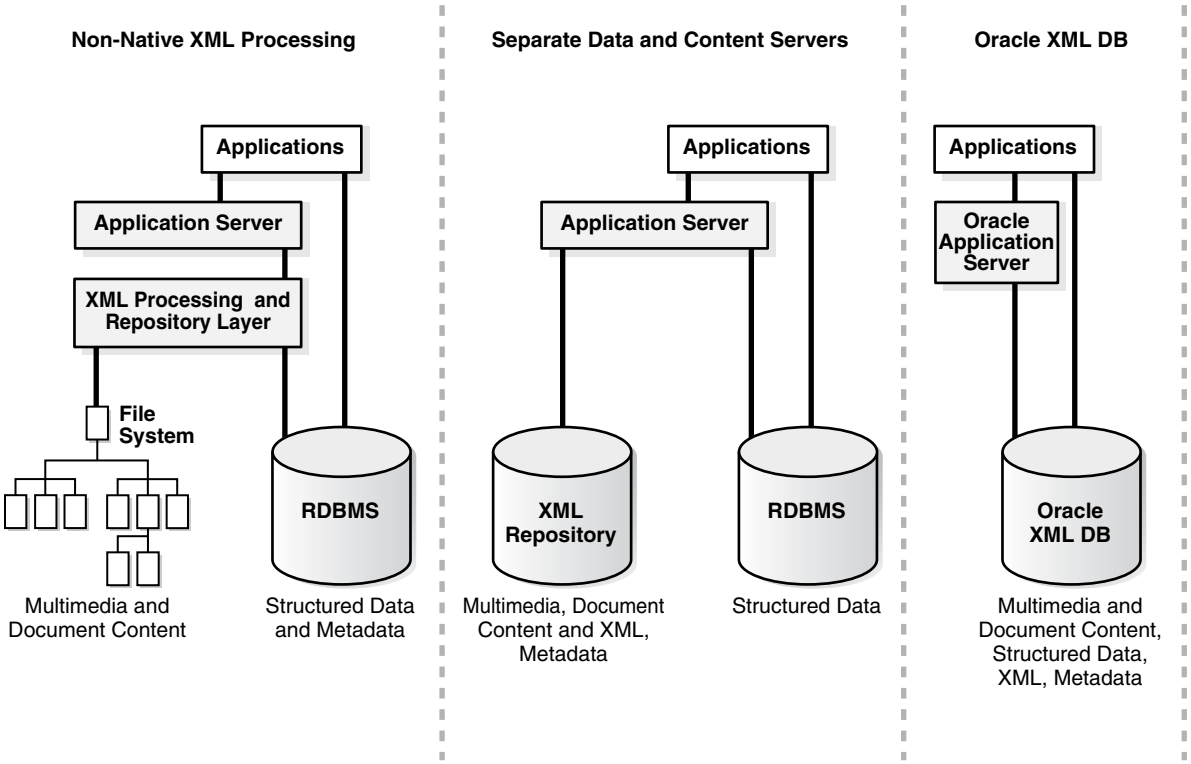
Figure 1–6 Oracle XML DB Benefits



Unifying Data and Content

Most application data and Web content is stored in a relational database, a file system, or both. XML data is often used for data exchange, and it can be generated from a relational database or a file system. As the volume of XML data exchanged grows, the cost of regenerating this data grows, and these storage methods become less effective at accommodating XML content.

Figure 1–7 Unifying Data and Content: Some Common XML Architectures



Organizations often manage their structured data and unstructured data differently:

- Unstructured data, in tables, makes document access transparent and table access complex.
- Structured data, often in binary large objects (such as in `BLOB` instances), makes access more complex and table access transparent.

With Oracle XML DB, you can store and manage data that is structured, unstructured, and semi-structured using a standard data model and standard SQL and XML. You can perform SQL operations on XML documents and XML operations on object-relational (such as table) data.

Exploiting Database Capabilities

Oracle Database has the following key database capabilities for working with XML:

- Indexing and search – Applications use queries such as "find all the product definitions created between March and April 2002", a query that is typically supported by a B-tree index on a date column. Oracle XML DB can enable efficient structured searches on XML data, saving content-management vendors the need to build proprietary query APIs to handle such queries.

  See Also:

  - Chapter 4, "XMLType Operations"
  - Chapter 12, "Full-Text Search Over XML Data"
  - Chapter 18, "Generating XML Data from the Database"

- Updates and transaction processing – Commercial relational databases use fast updates of subparts of records, with minimal contention between users trying to

update. As traditionally document-centric data participate in collaborative environments through XML, this requirement becomes more important. File or CLOB storage cannot provide the granular concurrency control that Oracle XML DB does.

See Also: Chapter 4, "XMLType Operations"

- Managing relationships – Data with any structure typically has foreign-key constraints. XML data stores generally lack this feature, so you must implement any constraints in application code. Oracle XML DB enables you to constrain XML data according to XML schema definitions, and hence achieve control over relationships that structured data has always enjoyed.

  See Also:

  - Chapter 7, "XML Schema Storage and Query: Basic"
  - The purchase-order examples in Chapter 4, "XMLType Operations"

- Multiple views of data – Most enterprise applications need to group data together in different ways for different modules. This is why relational views are necessary—to allow for these multiple ways to combine data. By allowing views on XML, Oracle XML DB creates different logical abstractions on XML for, say, consumption by different types of applications.

  See Also: Chapter 19, "XMLType Views"

- Performance and scalability – Users expect data storage, retrieval, and query to be fast. Loading a file or CLOB value, and parsing, are typically slower than relational data access. Oracle XML DB dramatically speeds up XML storage and retrieval.

  See Also:

  - Chapter 2, "Getting Started with Oracle XML DB"
  - Chapter 3, "Using Oracle XML DB"

- Ease of development – Databases are foremost an application platform that provides standard, easy ways to manipulate, transform, and modify individual data elements. While typical XML parsers give standard read access to XML data they do not provide an easy way to modify and store individual XML elements. Oracle XML DB supports several standard ways to store, modify, and retrieve data. These include XML Schema, XQuery, XPath, DOM, and Java.

  See Also:

  - Chapter 15, "Java DOM API for XMLType"
  - Chapter 25, "Accessing the Repository using RESOURCE_VIEW and PATH_VIEW"
  - Chapter 26, "Accessing the Repository using PL/SQL"

Exploiting XML Capabilities

If the drawbacks of XML file storage force you to break down XML into database tables and columns, there are several XML advantages you have left:

- Structure independence: The open content model of XML cannot be captured easily in the pure tables-and-columns world. XML schemas allow global element declarations, not just scoped to a container. Hence you can find a particular data item regardless of where in the XML document it moves to as your application evolves.

  See Also: Chapter 7, "XML Schema Storage and Query: Basic"

- Storage independence: When you use relational design, your client programs must know where your data is stored, in what format, what table, and what the relationships are among those tables. `XMLType` enables you to write applications without that knowledge and lets database administrators map structured data to physical table and column storage.

  See Also:

  - Chapter 7, "XML Schema Storage and Query: Basic"
  - Chapter 21, "Accessing Oracle XML DB Repository Data"

- Ease of presentation: XML is understood natively by Web browsers, many popular desktop applications, and most Internet applications. Relational data is generally not accessible directly from applications. Additional programming is required to make relational data accessible to standard clients. Oracle XML DB stores data as XML and makes it available as XML outside the database. No extra programming is required to display database content.

  See Also:

  - Chapter 11, "Transforming and Validating XMLType Data".
  - Chapter 18, "Generating XML Data from the Database".
  - Chapter 19, "XMLType Views".
  - *Oracle XML Developer's Kit Programmer's Guide*, in the chapter, "XSQL Pages Publishing Framework". It includes `XMLType` examples.

- Ease of interchange – XML is the language of choice in business-to-business (B2B) data exchange. If you are forced to store XML in an arbitrary table structure, you are using some kind of proprietary translation. Whenever you translate a language, information is lost and interchange suffers. By natively understanding XML and providing DOM fidelity in the storage/retrieval process, Oracle XML DB enables a clean interchange.

  See Also:

  - Chapter 11, "Transforming and Validating XMLType Data"
  - Chapter 19, "XMLType Views"

Efficient Storage and Retrieval of Complex XML Documents

Users today face a performance barrier when storing and retrieving complex, large, or many XML documents. Oracle XML DB provides high performance and scalability for XML operations. The major performance features are:

- Native `XMLType`. See Chapter 4, "XMLType Operations".
- A lazily evaluated virtual DOM. See Chapter 13, "PL/SQL APIs for XMLType".

- XQuery, XPath, and XSLT support. This is described in several chapters, including Chapter 4, "XMLType Operations", Chapter 11, "Transforming and Validating XMLType Data", and Chapter 5, "Using XQuery with Oracle XML DB".
- XML schema caching support. See Chapter 7, "XML Schema Storage and Query: Basic".
- Indexing, both full-text and XML. See Chapter 6, "Indexing XMLType Data" and Chapter 12, "Full-Text Search Over XML Data".
- A hierarchical index over Oracle XML DB Repository. See Chapter 21, "Accessing Oracle XML DB Repository Data".

Use XMLType Views If Your Data Is Not XML

`XMLType` views provide a way for you wrap existing relational and object-relational data in XML format. This is especially useful if, for example, your legacy data is not in XML format but you must migrate it to XML format. Using `XMLType` views, you need not alter your application code.

See Also: Chapter 19, "XMLType Views"

To use `XMLType` views, you must first register an XML schema with annotations that represent a bidirectional mapping between XML Schema data types and either SQL data types or binary XML encoding types. You can then create an `XMLType` view conforming to this mapping, by providing an underlying query that constructs instances of the appropriate types.

Search XML Data using Oracle Text

Oracle Database enables special indexing on XML data, including Oracle Text indexes for section searching, SQL functions to process XML data, aggregation of XML data, and special optimization of queries involving XML data. Oracle SQL functions `hasPath` and `inPath` are designed to optimize XML data searches where you can search within XML text for substring matches.

See Also:

- Chapter 12, "Full-Text Search Over XML Data"
- "Oracle Text Indexes on XML Data" on page 6-48
- *Oracle Text Application Developer's Guide*
- *Oracle Text Reference*

Build Messaging Applications using Oracle Streams Advanced Queuing

Oracle Streams Advanced Queuing supports the use of:

- `XMLType` as a message/payload type, including XML schema-based `XMLType`
- Queuing or dequeuing of `XMLType` messages

See Also:

- *Oracle Streams Advanced Queuing User's Guide* for information about using `XMLType` with Oracle Streams Advanced Queuing
- Chapter 37, "Exchanging XML Data using Oracle Streams AQ"

Standards Supported by Oracle XML DB

Oracle XML DB supports all major XML, SQL, Java, and Internet standards:

- W3C XML Schema 1.0 Recommendation. You can register XML schemas, validate stored XML content against XML schemas, or constrain XML stored in the server to XML schemas.
- W3C XQuery 1.0 Recommendation and W3C XPath 2.0 Recommendation. You can search or traverse XML stored inside the database using XQuery and XPath, either from HTTP(S) requests or from SQL.
- SQL/XML.
- Java Database Connectivity (JDBC) API. JDBC access to XML is available for Java programmers.
- W3C XSL 1.0 Recommendation. You can transform XML documents at the server using XSLT.
- W3C DOM Recommendation Levels 1.0 and 2.0 Core. You can retrieve XML stored in the server as an XML DOM, for dynamic access.
- Protocol support. You can store or retrieve XML data from Oracle XML DB using Oracle Net or standard protocols such as HTTP(S), FTP, and IETF WebDAV.
- Java Servlet version 2.2, (except: the servlet WAR file, `web.xml`, is not supported in its entirety; only one `ServletContext`; one `web-app` are currently supported; and stateful servlets are not supported).
- Web services: SOAP 1.1. You can access XML stored in the server from SOAP requests. You can build, publish, or find Web Services using Oracle XML DB and Oracle9*i*AS, using WSDL and UDDI. You can use Oracle Streams Advanced Queuing IDAP, the SOAP specification for queuing operations, on XML stored in Oracle Database.
- W3C XML Linking Language (Xlink) 1.0 Recommendation. You can define various types of links between XML documents.
- W3C XML Pointer Language (XPointer) Recommendation and XPointer Framework. You can include the content of multiple XML documents or fragments in a single infoset.

See Also:

- "SQL/XML Standard Functions" on page 1-20 for more information about the SQL/XML functions
- *Oracle Database SQL Language Reference* for information about Oracle support for the SQL/XML standard
- Chapter 23, "Using XLink and XInclude with Oracle XML DB" for more information about XLink and XPointer support
- Chapter 28, "Accessing the Repository using Protocols" for more information about protocol support
- Chapter 32, "Writing Oracle XML DB Applications in Java" for information about using the Java servlet
- Chapter 37, "Exchanging XML Data using Oracle Streams AQ" and *Oracle Streams Advanced Queuing User's Guide* for information about using SOAP

Oracle XML DB Technical Support

Besides your regular channels of support through your customer representative or consultant, technical support for Oracle Database XML-enabled technologies is available free through the Discussions option on Oracle Technology Network (OTN):

http://www.oracle.com/technology/tech/xml/

Oracle XML DB Examples Used in This Manual

This manual contains examples that illustrate the use of Oracle XML DB and `XMLType`. The examples are based on various database schemas, sample XML documents, and sample XML schemas.

> **See Also:** Appendix A, "Oracle-Supplied XML Schemas and Examples"

Further Oracle XML DB Case Studies and Demonstrations

Visit OTN to view Oracle XML DB examples, white papers, case studies, and demonstrations:

http://www.oracle.com/technology/tech/xml/

Comprehensive XML classes on how to use Oracle XML DB are also available. See the Oracle University link on OTN.

Several detailed Oracle XML DB case studies are available on OTN and include the following:

- Oracle XML DB Downloadable Demonstration. This detailed demonstration illustrates how to use many Oracle XML DB features. Parts of this demonstration are also included in Chapter 3, "Using Oracle XML DB".
- Content Management System (CMS) application. This illustrates how you can store files on the database using Oracle XML DB Repository in hierarchically organized folders, place the files under version control, provide security using ACLs, transform XML content to a desired format, search content using Oracle Text, and exchange XML messages using Oracle Streams Advanced Queuing. See: http://www.oracle.com/technology/sample_code/tech/xml/xmldb/cmsxdb/content.html
- XML Dynamic News. This is a complete J2EE 1.3 based application that demonstrates Java and Oracle XML DB features for an online news portal. News feeds are stored and managed persistently in Oracle XML DB. Various other news portals can customize this application to provide static or dynamic news services to end users. End users can personalize their news pages by setting their preferences. The application also demonstrates the use of Model View Controller (MVC) architecture and various J2EE design patterns. See http://www.oracle.com/technology/sample_code/tech/xml/xmlnews/content.html
- SAX Loader Application. This demonstrates an efficient way to break up large files containing multiple XML documents outside the database and insert them into the database as a set of separate documents. This is provided as a standalone and a Web-based application.
- Oracle XML DB Utilities package. This highlights the subprograms provided with the `XDB_Utilities` package. These subprograms operate on `BFILE` values, `CLOB` values, DOM, and Oracle XML DB Resource APIs. With this package, you

can perform basic Oracle XML DB foldering operations, read and load XML files into a database, and perform basic DOM operations through PL/SQL.

- Card Payment Gateway Application. This application uses Oracle XML DB to store all your data in XML format and enables access to the resulting XML data using SQL. It illustrates how a credit card company can store its account and transaction data in the database and also maintain XML fidelity.
- Survey Application. This application determines what members want from Oracle products. OTN posts the online surveys and studies the responses. This Oracle XML DB application demonstrates how a company can create dynamic, interactive HTML forms, deploy them to the Internet, store the responses as XML, and analyze them using the XML enabled Oracle Database.

2

Getting Started with Oracle XML DB

This chapter provides some preliminary design criteria for consideration when planning your Oracle XML DB solution.

This chapter contains these topics:

- Oracle XML DB Installation
- Oracle XML DB Use Cases
- Application Design Considerations for Oracle XML DB
- Oracle XML DB Performance

Oracle XML DB Installation

Oracle XML DB is installed automatically in the following situations:

- If Database Configuration Assistant (DBCA) is used to build Oracle Database using the general-purpose template
- If you use SQL script `catqm` to install Oracle Database

You can determine whether or not Oracle XML DB is already installed. If it is installed, then the following are true:

- Database schema (user account) `XDB` exists. To check that, run this query:

  ```
  SELECT * FROM ALL_USERS;
  ```

- View `RESOURCE_VIEW` exists. To check that, use this command:

  ```
  DESCRIBE RESOURCE_VIEW
  ```

See Also:

- Chapter 34, "Administering Oracle XML DB" for information about installing and uninstalling Oracle XML DB *manually*
- *Oracle Database 2 Day + Security Guide* for information about database schema `XDB`

Oracle XML DB Use Cases

Oracle XML DB is suited for any application where some or all of the data processed by the application is represented using XML. Oracle XML DB provides for high-performance database ingestion, storage, processing and retrieval of XML data. It also lets you quickly and easily generate XML from existing relational data.

Applications for which Oracle XML DB is particularly suited include the following:

- Business-to-business (B2B) and application-to-application (A2A) integration
- Internet
- Content-management
- Messaging
- Web Services

A typical Oracle XML DB application has one or more of the following characteristics:

- Large numbers of XML documents must be ingested or generated
- Large XML documents must be processed or generated
- High-performance searching is needed, both within a document and across large collections of documents
- High levels of security are needed
- Fine-grained security is needed
- Data processing must use XML documents, and data must be stored in relational tables
- Programming must support open standards such as SQL, XML, XQuery, XPath, and XSL
- Information must be accessed using standard Internet protocols such as FTP, HTTP(S)/WebDAV, and Java Database Connectivity (JDBC)
- XML data must be queried from SQL
- Analytic capabilities must be applied to XML data
- XML documents must be validated against an XML schema

Oracle XML DB lets you fine-tune how XML documents are stored and processed in Oracle Database. Depending on the nature of the application, XML storage must have at least one of the following features

- High performance ingestion and retrieval of XML documents
- High performance indexing and searching of XML documents
- Ability to update sections of an XML document
- Management of structured or unstructured XML documents

Application Design Considerations for Oracle XML DB

This section mentions some preliminary design criteria that you can consider when planning your Oracle XML DB application.

Structure of Your Data

Is your data be highly structured (mostly XML), semi-structured, or mostly unstructured? If highly structured, are your tables XML schema-based or non-schema-based?

If your XML data is *not* XML schema-based, then, regardless of how structured it is, you can store it in an `XMLType` table or view as binary XML or as a `CLOB` instance, or you can store it as a file in an Oracle XML DB Repository folder.

If your XML data is XML schema-based then you can use unstructured, structured (object-relational), or binary XML storage for its structured parts. For the unstructured parts, you have the same options as for data that is not XML schema-based.

See Also: Chapter 3, "Using Oracle XML DB"

Oracle XML DB Repository Access

This section pertains to data that is stored as resources in Oracle XML DB Repository.

How do other applications and users need to access your XML and other data? How secure must the access be? Do you need versioning?

There are two main repository access methods:

- Navigation-based access or path-based access. This is suitable for both content/document and data oriented applications. Oracle XML DB provides the following languages and access APIs:
  - SQL access through resource and path views. See Chapter 25, "Accessing the Repository using RESOURCE_VIEW and PATH_VIEW".
  - PL/SQL access through `DBMS_XDB`. See Chapter 26, "Accessing the Repository using PL/SQL".
  - Protocol-based access using HTTP(S)/WebDAV or FTP, most suited to content-oriented applications. See Chapter 28, "Accessing the Repository using Protocols".
- Query-based access. This can be most suited to data oriented applications. Oracle XML DB provides access using SQL queries through the following APIs:
  - Java access (through JDBC). See Java DOM API for XMLType.
  - PL/SQL access. See Chapter 13, "PL/SQL APIs for XMLType".

These options for accessing repository data are also discussed in Chapter 21, "Accessing Oracle XML DB Repository Data".

You can also consider the following access criteria:

- What levels of security do you need? See Chapter 27, "Repository Access Control".
- What kind of indexing best suits your application? Do you need to use Oracle Text indexing and querying? See Chapter 4, "XMLType Operations", Chapter 6, "Indexing XMLType Data", and Chapter 12, "Full-Text Search Over XML Data".
- Do you need to version the data? If yes, see Chapter 24, "Managing Resource Versions".

Application Language

In which languages do you program your application?

You can program your Oracle XML DB applications in the following languages:

- Java (JDBC, Java Servlets)

  See Also:

  - Chapter 15, "Java DOM API for XMLType"
  - Chapter 32, "Writing Oracle XML DB Applications in Java"

- PL/SQL

See Also:

- Chapter 13, "PL/SQL APIs for XMLType"
- Chapter 26, "Accessing the Repository using PL/SQL"
- "APIs for XML" on page 1-4

Processing

Do you need to generate XML data? See Chapter 18, "Generating XML Data from the Database".

How often are XML documents accessed, updated, and manipulated? Do you need to update fragments or whole documents?

Do you need to transform XML data to HTML, WML, or other languages? If so, how does your application do this? See Chapter 11, "Transforming and Validating XMLType Data".

Must your application be primarily database-resident or must it work in both the database and middle tier?

Is your application data-centric, document-centric (content-centric), or both?

The following processing options are available and should be considered when designing your Oracle XML DB application:

- XSLT. Do you need to transform the XML data to HTML, WML, or other languages, and, if so, how does your application transform the XML data? While storing XML documents in Oracle XML DB, you can optionally ensure that their structure complies with (validates against) specific XML schemas. See Chapter 11, "Transforming and Validating XMLType Data".
- DOM fidelity, document fidelity. Use unstructured storage to preserve document fidelity. Use binary XML or structured storage for XML schema-based data to preserve DOM fidelity. See Chapter 13, "PL/SQL APIs for XMLType" and "DOM Fidelity" on page 7-16.
- XPath searching. You can use XPath syntax embedded in a SQL statement or as part of an HTTP(S) request to query XML content in the database. See Chapter 4, "XMLType Operations", Chapter 12, "Full-Text Search Over XML Data", Chapter 21, "Accessing Oracle XML DB Repository Data", and Chapter 25, "Accessing the Repository using RESOURCE_VIEW and PATH_VIEW".
- XML Generation and `XMLType` views. Do you need to generate or regenerate XML data? If yes, see Chapter 18, "Generating XML Data from the Database".

How often are XML documents accessed, updated, and manipulated? See Chapter 4, "XMLType Operations" and Chapter 25, "Accessing the Repository using RESOURCE_VIEW and PATH_VIEW".

Do you need to update fragments or whole documents? You can use XPath expressions to specify individual elements and attributes of your document during updates, without rewriting the entire document. This is more efficient, especially for large XML documents. Chapter 7, "XML Schema Storage and Query: Basic".

Is your application data-centric, document- and content-centric, or *integrated* (is both data- and document-centric)? See Chapter 3, "Using Oracle XML DB".

Messaging

Does your application exchange XML data with other applications across gateways? Do you need Oracle Streams Advanced Queuing (AQ) or SOAP compliance? See Chapter 37, "Exchanging XML Data using Oracle Streams AQ".

Advanced Queuing (AQ) supports XML and `XMLType` applications. You can create queues with payloads that contain `XMLType` attributes. These can be used for transmitting and storing messages that contain XML documents. By defining Oracle Database objects with `XMLType` attributes, you can do the following:

- Store more than one type of XML document in the same queue. The documents are stored internally as `CLOB` values.
- Selectively dequeue messages with `XMLType` attributes using an XPath or XQuery expression.
- Define rule-based subscribers that query message content using an XPath or XQuery expression.
- Define transformations to convert Oracle Database objects to `XMLType`.

See Also:

- Chapter 37, "Exchanging XML Data using Oracle Streams AQ"
- *Oracle Streams Advanced Queuing User's Guide*

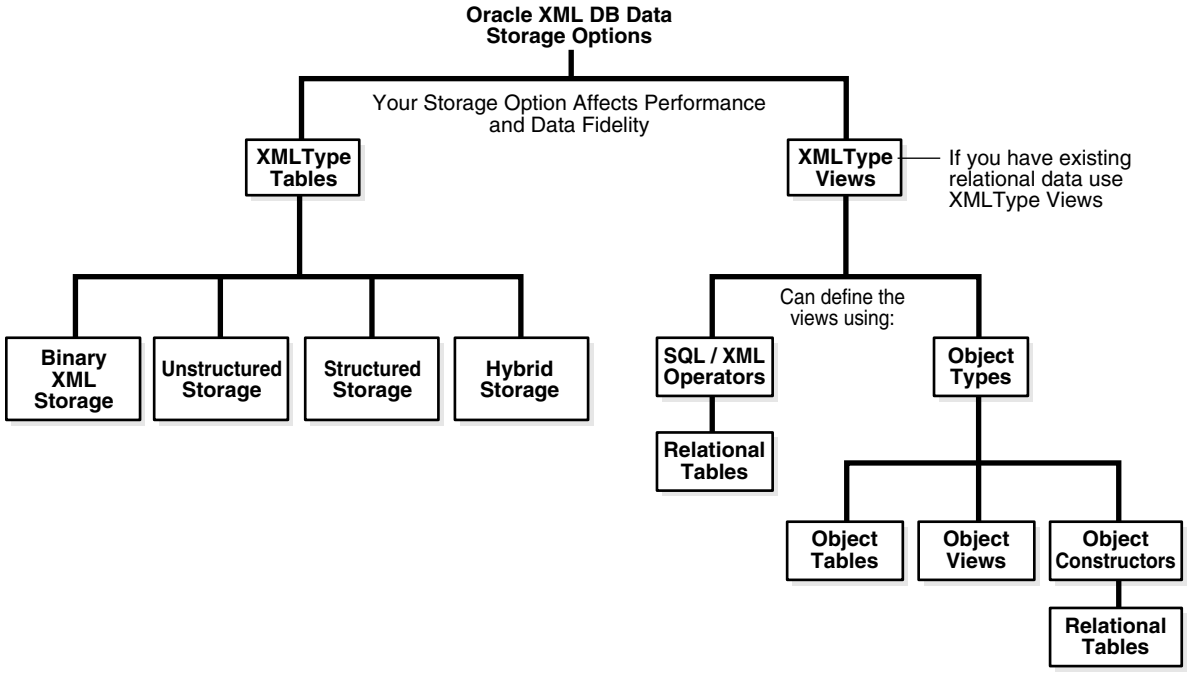
Storage

How and where do you store your relational data, XML data, XML schemas, and so on?

Note: The choices you make for data structure, access, language, and processing are typically interdependent, but they are not dependent on the storage model you choose.

Figure 2–1 shows the Oracle XML DB storage options for `XMLType` tables and views.

Figure 2–1 Oracle XML DB Storage Options for XML Data



If you have existing relational data, you can access it as XML data by creating `XMLType` views over it. You can use the following to define the `XMLType` views:

- SQL/XML functions. See Chapter 18, "Generating XML Data from the Database" and Chapter 5, "Using XQuery with Oracle XML DB".
- Object types: object tables, object constructors, and object views.

Regardless of which storage options you choose for your application, Oracle XML DB provides the same functionality. Though the storage model you use can affect your application performance and XML data fidelity, it is totally independent of all of the following:

- How, and how often, you query or update your data.
- How you access your data. This is determined only by your application processing requirements.
- What language(s) your application uses. This is determined only by your application processing requirements.

See Also:

- "XMLType Storage Models" on page 1-14
- "DOM Fidelity" on page 7-16

Oracle XML DB Performance

One objection to using XML to represent data is that it generates higher overhead than other representations. Oracle XML DB incorporates several features specifically designed to address this issue by significantly improving the performance of XML processing. These are described in the following sections:

- XML Storage Requirements

- XML Memory Management
- XML Parsing Optimizations
- Node-Searching Optimizations
- XML Schema Optimizations
- Load Balancing Through Cached XML Schema
- Reduced Bottlenecks From Code That Is Not Native
- Reduced Java Type Conversion Bottlenecks

XML Storage Requirements

Data represented in XML and stored in a text file averages three times the size of the same data in a Java object or in relational tables. There are two main reasons for this:

- Tag names (metadata describing the data) and white space (formatting characters) take up a significant amount of space in the document, particularly for highly structured, data-centric XML.
- All data in an XML file is represented in human readable (string) format.

The string representation of a numeric value needs about twice as many bytes as the native (binary) representation. When XML documents are stored in Oracle XML DB using structured or binary XML storage, the storage process discards all tags and white space in the document.

The amount of space saved by this optimization depends on the ratio of tag names to data, and the number of collections in the document. For highly-structured, data-centric XML data, the savings can be significant. When a document is printed, or when node-based operations such as XPath evaluation take place, Oracle XML DB uses the information contained in the associated XML schema to dynamically reconstruct any necessary tag information.

XML Memory Management

Document Object Model (DOM) is the dominant programming model for XML documents. DOM APIs are easy to use but the DOM Tree that underpins them is expensive to generate, in terms of memory. A typical DOM implementation maintains approximately 80 to 120 bytes of system overhead for each node in the DOM tree. For highly structured data, the DOM tree can require 10 to 20 times more memory than the document on which it is based.

A conventional DOM implementation requires the entire contents of an XML document to be loaded into the DOM tree before any operations can take place. If an application only needs to process a small percentage of the nodes in the document, this is extremely inefficient in terms of memory and processing overhead. The alternative Simple API for XML (SAX) approach reduces the amount of memory required to process an XML document, but its disadvantage is that it only allows linear processing of nodes in the XML document.

See Also:

- `http://www.w3.org/DOM/` for information about DOM
- `http://www.saxproject.org/` for information about SAX

Use of XOBs Reduces Memory Overhead for XML Schema-Based Documents

Oracle XML DB reduces memory overhead associated with DOM programming by managing XML schema-based XML documents using an internal structure in dynamic memory called an XML Object (XOB). A XOB is much smaller than the equivalent DOM since it does not duplicate information like tag names and node types, that can easily be obtained from the associated XML schema. Oracle XML DB automatically uses a XOB whenever an application works with the contents of a schema-based `XMLType`. The use of the XOB is transparent to you. It is hidden behind the `XMLType` data type and the C, PL/SQL, and Java APIs.

XOB Uses a Lazily-Loaded Virtual DOM

The XOB can also reduce the amount of memory required to work with an XML document using the Lazily-Loaded Virtual DOM feature. This lets Oracle XML DB defer loading the dynamic memory representation of nodes that are part of sub-elements or collection until code attempts to operate on a node in that object. Consequently, if an application only operates on a few nodes in a document, only those nodes and their immediate siblings are loaded into memory.

The XOB can only used when an XML document is based on an XML schema. If the contents of the XML document are not based on an XML schema, a traditional DOM is used instead of the XOB.

XML Parsing Optimizations

To populate a DOM tree the application must parse the XML document. The process of creating a DOM tree from an XML file is very CPU- intensive. In a typical DOM-based application, where the XML documents are stored as text, every document has to be parsed and loaded into the DOM tree before the application can work with it. If the contents of the DOM tree are updated the entire tree must be serialized back into a text format and written out to disk.

Oracle XML DB eliminates the need to parse documents over and over again. No parsing is needed when an XML document is loaded from disk into memory, if the document is stored as structured or binary XML storage. Oracle XML DB maps directly between the format on disk and the format in dynamic memory using information derived from the associated XML schema. When changes are made to XML schema-based data, Oracle XML DB is able to write just the updated data back to disk. When XML data is not based on an XML schema, a traditional DOM is used instead.

Node-Searching Optimizations

Most DOM implementations use string comparisons when searching for a particular node in the DOM tree. Even a simple search of a DOM tree can require hundreds or thousands of instruction cycles. Searching for a node in a XOB is much more efficient than searching for a node in a DOM. A XOB is based on a computed offset model, similar to a C/C++ object, and uses dynamic hashtables rather than string comparisons to perform node searches.

XML Schema Optimizations

Making use of the powerful features associated with XML schema in a conventional XML application can generate significant amounts of additional overhead. For example, before an XML document can be validated against an XML schema, the schema itself must be located, parsed, and validated.

Oracle XML DB minimizes the overhead associated with using XML schema. When an XML schema is registered with the database, it is loaded in the Oracle XML DB schema cache, together with all of the metadata required to map between the textual, XOB and on- disk representations of the data. After the XML schema has been registered with the database no additional parsing or validation of the XML schema is required before it can be used. The schema cache is shared by all users of the database. Whenever an Oracle XML DB operation requires information contained in the XML schema, it can access the required information directly from the cache.

Load Balancing Through Cached XML Schema

Some operations, such as performing a full schema validation, or serializing an XML document back into text form, can still require significant memory and CPU resources. Oracle XML DB let these operations be off-loaded to the client or middle tier processor. Both Oracle Call Interface (OCI) interface and the OCI driver for JDBC allow the XOB to be managed by the client.

The cached representation of the XML schema can also be downloaded to the client. This lets operations such as XML printing, and XML schema validation be performed using client or middle tier resources, rather than server resources.

Reduced Bottlenecks From Code That Is Not Native

Another bottleneck for XML-based Java applications happens when parsing an XML file. Even natively compiled or JIT compiled Java performs XML parsing operations twice as slowly compared to using native C language. One of the major performance bottlenecks in implementing XML applications is the cost of transforming data in an XML document between text, Java, and native server representations. The cost of performing these transformations is proportional to the size and complexity of the XML file and becomes severe even in moderately sized files.

Oracle XML DB addresses these issues by implementing all of the Java and PL/SQL interfaces as thin facades over a native implementation in the C language. Java, C, PL/SQL, and SQL all use the same underlying implementation. This provides for language-neutral XML support and higher performance XML parsing and DOM processing.

Reduced Java Type Conversion Bottlenecks

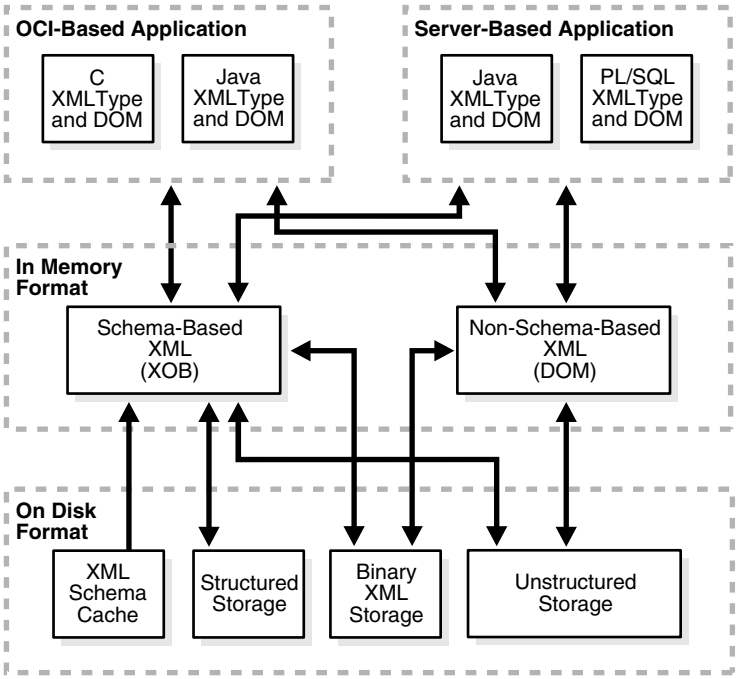
One of the biggest bottlenecks when using Java and XML is with type conversions. Internally Java uses UCS-2 to represent character data. Most XML files and databases do not contain UCS-2 encoded data. All data contained in an XML file must be converted from 8-Bit or UTF-8 encoding to UCS-2 encoding before it can be manipulated in a Java program.

Oracle XML DB addresses these problems with lazy type conversions. With lazy type conversions, the content of a node is not converted into the format required by Java until the application attempts to access the contents of the node. Data remains in the internal representation till the last moment. Avoiding unnecessary type conversions can result in significant performance improvements when an application only needs to access a few nodes in an XML document.

Consider a JSP that loads a name from the Oracle Database and prints it out in the generated HTML output. Typical JSP implementations read the name from the database (that probably contains data in the ASCII or ISO8859 character sets), convert the data to UCS-2, and return it to Java as a string. The JSP would not look at the string content, but only print it out after printing the enclosing HTML, probably converting

back to the same ASCII or ISO8859 for the client browser. Oracle XML DB provides a write interface on `XMLType` so that any element can write itself directly to a stream (such as a `ServletOutputStream`) without conversion through Java character sets. Figure 2–2 shows the Oracle XML DB Application Program Interface (API) stack.

Figure 2–2 Oracle XML DB Application Program Interface (API) Stack



3

Using Oracle XML DB

This chapter is an overview of how to use Oracle XML DB. The examples presented here illustrate techniques for accessing and managing XML content in purchase-order documents. Purchase orders are highly structured documents, but you can use the techniques shown here to also work with XML documents that have little structure.

This chapter contains these topics:

- Storing XML Data as XMLType
- Creating XMLType Tables and Columns
- Partitioning or Constraining Binary XML Data using Virtual Columns
- Loading XML Content into Oracle XML DB
- Character Sets of XML Documents
- Overview of the W3C XML Schema Recommendation
- Using XML Schema with Oracle XML DB
- Identifying XML Schema Instance Documents
- Enforcing XML Data Integrity using the Database
- DML Operations on XML Content using Oracle XML DB
- Querying XML Content Stored in Oracle XML DB
- Accessing XML Data in Oracle XML DB using Relational Views
- Updating XML Content Stored in Oracle XML DB
- Namespace Support in Oracle XML DB
- How Oracle XML DB Processes XMLType Methods and SQL Functions
- Generating XML Data from Relational Data
- XSL Transformation and Oracle XML DB
- Using Oracle XML DB Repository
- Viewing Relational Data as XML From a Browser
- XSL Transformation using DBUri Servlet

Storing XML Data as XMLType

Before the introduction of Oracle XML DB, there were two ways to store XML content in Oracle Database:

- Use Oracle XML Developer's Kit (XDK) to parse the XML document outside Oracle Database, and store the extracted XML data as rows in one or more tables in the database.
- Store the XML document in Oracle Database using a Character Large Object (`CLOB`), Binary Large Object (`BLOB`), Binary File (`BFILE`), or `VARCHAR` column.

In both cases, Oracle Database is unaware that it is managing XML content.

Oracle XML DB and the `XMLType` abstract data type make Oracle Database XML-aware. Storing XML data as an `XMLType` column or table lets the database perform XML-specific operations on the content. This includes XML validation and optimization. `XMLType` storage allows highly efficient processing of XML content in the database.

What is XMLType?

`XMLType` is an abstract data type for native handling of XML data in the database.

- `XMLType` has built-in methods to create, extract, and index database XML data.
- `XMLType` provides SQL access to XML data.
- `XMLType` functionality is also available through a set of Application Program Interfaces (APIs) provided in PL/SQL and Java. `XMLType` can be used in PL/SQL stored procedures for parameters, return values, and variables.

Using `XMLType`, SQL developers can leverage the power of the relational database while working in the context of XML. XML developers can leverage the power of XML standards while working in the context of a relational database.

`XMLType` can be used as the data type of columns in tables and views. `XMLType` variables can be used in PL/SQL stored procedures as parameters and return values. You can also use `XMLType` in SQL, PL/SQL, C, Java (through JDBC), and Oracle Data Provider for .NET (ODP.NET).

The `XMLType` API provides several useful methods that operate on XML content. For example, method `extract()` extracts one or more nodes from an `XMLType` instance.

Oracle XML DB functionality is based on the Oracle XML Developer's Kit C implementations of the relevant XML standards such as XML Parser, XML DOM, and XML Schema Validator.

See Also:

- "XMLType Data Type" on page 1-12
- "XMLType Storage Models" on page 1-14 for the available `XMLType` storage options and their relative advantages

Benefits of XMLType Data Type and API

The `XMLType` data type and application programming interface (API) enable SQL operations on XML content and XML operations on SQL content:

- Versatile API – `XMLType` has a versatile API for application development that includes built-in functions, indexing, and navigation support.
- `XMLType` and SQL – You can use `XMLType` in SQL statements, combined with other data types. For example, you can query `XMLType` columns and join the result of the extraction with a relational column. Oracle Database determines an optimal way to run such queries.

- Indexing – You can created several kinds of indexes to improve the performance of queries on XML data.
  - For structured storage of `XMLType` data, you can create B-tree indexes and function-based indexes on the object-relational tables that underlie `XMLType` tables and columns. Create function-based indexes only on scalar data, that is, columns that represent singleton elements or attributes.
  - For unstructured and binary XML storage of `XMLType` data, you can create an `XMLIndex` index, which specifically targets the XML structure of a document.
  - You can index the textual content of XML data with an Oracle Text `CONTEXT` index, for use in full-text search. This applies to all `XMLType` storage models.

Creating XMLType Tables and Columns

`XMLType` is an abstract data type, so it is straightforward to create an `XMLType` table or column. The basic `CREATE TABLE` statement, specifying no storage options and no XML schema, stores `XMLType` data as binary XML.[1]

Example 3–1 creates an `XMLType` column, and Example 3–2 creates an `XMLType` table.

Example 3–1 Creating a Table with an XMLType Column

```
CREATE TABLE mytable1 (key_column VARCHAR2(10) PRIMARY KEY, xml_column XMLType);
```

Example 3–2 Creating a Table of XMLType

```
CREATE TABLE mytable2 OF XMLType;
```

See Also: "Creating XMLType Tables and Columns Based on XML Schemas" on page 7-27

Note: To create an `XMLType` table in a different database schema from your own, you must have not only privilege `CREATE ANY TABLE` but also privilege `CREATE ANY INDEX`. This is because a unique index is created on column `OBJECT_ID` when you create the table. Column `OBJECT_ID` stores a system-generated object identifier.

Partitioning or Constraining Binary XML Data using Virtual Columns

XML data has its own structure, which, except for object-relational storage of `XMLType`, is not reflected directly in database data structure. That is, individual XML elements and attributes are not mapped to individual database columns or tables.

Therefore, to constrain or partition XML data according to the values of individual elements or attributes, the standard approach for relational data does not apply. Instead, you must create *virtual columns* that represent the XML data of interest, and then use those virtual columns to define the constraints or partitions that you need.

This approach applies only to XML data that is stored as *binary XML*. For XML data that uses unstructured storage, the database has no knowledge of the XML structure—the data is treated as flat text, but for binary XML storage that structure is

[1] The `XMLType` storage model for XML schema-based data is whatever was specified during registration of the referenced XML schema. If no storage model was specified during registration, then binary XML storage is used.

known. You can exploit this structural knowledge to create virtual columns, which the database can then use with constraints or partitions.

The technique is as follows:

1. Define virtual columns that correspond to the XML data that you are interested in.
2. Use those columns to partition or constrain the `XMLType` data as a whole.

You create virtual columns on `XMLType` data as you would create virtual columns using any other type of data, but using a slightly different syntax. In particular, you cannot specify any constraints in association with the column definition.

Because `XMLType` is an abstract data type, if you create virtual columns on an `XMLType` table then those columns are *hidden*. They do not show up in `DESCRIBE` statements, for example. This hiding enables tools that use operations such as `DESCRIBE` to function normally and not be misled by the virtual columns.

Note:

- Partitioning of binary XML tables is supported starting with 11g Release 2 (11.2). It is supported only if the database compatibility (parameter `compatible` in file `init.ora`) is 11.2 or higher.
- Range, hash, and list partitioning are supported.
- You can partition an `XMLType` *table* using a virtual column. You cannot partition a relational table that has an `XMLType` *column*, using that column to define virtual columns of XML data.

You create a virtual column based on an XML element or attribute by defining it in terms of a SQL expression that involves that element or attribute. The column is thus function-based. You use SQL/XML functions `XMLCast` and `XMLQuery` to do this, as shown in Example 3–3. The XQuery expression argument to function `XMLQuery` must be a simple XPath expression that uses only the child and attribute axes.

Example 3–3 Partitioning a Binary XML Table using Virtual Columns

```
CREATE TABLE po_binaryxml OF XMLType
  XMLTYPE STORE AS BINARY XML
  VIRTUAL COLUMNS
    (DATE_COL AS (XMLCast(XMLQuery('/PurchaseOrder/@orderDate'
                                   PASSING OBJECT_VALUE RETURNING CONTENT)
                          AS DATE)))
  PARTITION BY RANGE (DATE_COL)
    (PARTITION orders2001 VALUES LESS THAN (to_date('01-JAN-2002')),
     PARTITION orders2002 VALUES LESS THAN (MAXVALUE));
```

Example 3–3 partitions an `XMLType` table using a virtual column, `DATE_COL`, which targets the `orderDate` element in a purchase-order document.

To use a virtual column for partitioning, its data type must be constant. In the case where the `XMLType` data in the column or table is mixed, some documents being encoded using an XML schema and others being encoded without using any schema, you must cast the functional expression, to ensure that the same data type is used for all rows in the virtual column.

Note: For best performance, choose, as the partitioning key, an XPath expression whose target occurs within 32 K bytes of the beginning of the XML document.

You define constraints on binary XML data similarly. See Example 3–20 on page 3-35.

See Also:

- "XMLIndex Partitioning and Parallelism" on page 6-36
- "Enforcing Referential Integrity using SQL Constraints" on page 3-34
- *Oracle Database SQL Language Reference* for information about creating tables with virtual columns

Loading XML Content into Oracle XML DB

You can load XML content into Oracle XML DB using these techniques:

- Table-based loading:
  - Loading XML Content using SQL or PL/SQL
  - Loading XML Content using Java
  - Loading XML Content using C
  - Loading Large XML Files That Contain Small XML Documents
  - Loading Large XML Files using SQL*Loader
- Path-based repository loading techniques:
  - Loading XML Documents into the Repository using DBMS_XDB
  - Loading Documents into the Repository using Protocols

Loading XML Content using SQL or PL/SQL

You can use a simple `INSERT` operation in SQL or PL/SQL to load an XML document into the database. Before the document can be stored as an `XMLType` column or table, you must convert it into an `XMLType` instance using one of the `XMLType` constructors.

See Also:

- Chapter 4, "XMLType Operations"
- "APIs for XML" on page 1-4
- *Oracle Database PL/SQL Packages and Types Reference* for a description of the `XMLType` constructors

`XMLType` **constructors** allow an `XMLType` instance to be created from different sources, including `VARCHAR`, `CLOB`, and `BFILE` values. The constructors accept additional arguments that reduce the amount of processing associated with `XMLType` creation. For example, if you are sure that a given source XML document is valid, you can provide an argument to the constructor that disables the type-checking that is otherwise performed.

In addition, if the source data is not encoded in the database character set, an `XMLType` instance can be constructed using a `BFILE` or `BLOB` value. The encoding of the source data is specified through the character set id (`csid`) argument of the constructor.

Example 3–5 shows how to insert XML content into an `XMLType` table. Before making this insertion, you must create a database directory object that points to the directory containing the file to be processed. To do this, you must have the `CREATE ANY DIRECTORY` privilege.

> **See Also:** *Oracle Database SQL Language Reference*, Chapter 18, under `GRANT`

Example 3–4 Creating a Database Directory

```
CREATE DIRECTORY xmldir AS path_to_folder_containing_XML_file;
```

Example 3–5 Inserting XML Content into an XMLType Table

```
INSERT INTO mytable2 VALUES (XMLType(bfilename('XMLDIR', 'purchaseOrder.xml'),
                                     nls_charset_id('AL32UTF8')));
```

The value passed to `nls_charset_id` indicates that the encoding for the file to be read is UTF-8.

When you use SQL `INSERT` to insert a large document containing collections into `XMLType` tables (but not into `XMLType` columns), Oracle XML DB optimizes load time and memory usage.

> **See Also:** "Loading and Retrieving Large Documents with Collections" on page 9-28

Loading XML Content using Java

Example 3–6 shows how to load XML content into Oracle XML DB by first creating an `XMLType` instance in Java, given a Document Object Model (DOM).

Example 3–6 Inserting Content into an XMLType Table using Java

```
public void doInsert(Connection conn, Document doc)
throws Exception
{
   String SQLTEXT = "INSERT INTO purchaseorder VALUES (?)";
   XMLType xml = null;
   xml = XMLType.createXML(conn,doc);
   OraclePreparedStatement sqlStatement = null;
   sqlStatement = (OraclePreparedStatement) conn.prepareStatement(SQLTEXT);
   sqlStatement.setObject(1,xml);
   sqlStatement.execute();
}
```

The "Simple Bulk Loader Application" available on the Oracle Technology Network (OTN) site at `http://www.oracle.com/technology/sample_code/tech/xml/xmldb/content.html` demonstrates how to load a directory of XML files into Oracle XML DB using Java Database Connectivity (JDBC). JDBC is a set of Java interfaces to Oracle Database.

Loading XML Content using C

Example 3–7 shows how to insert XML content into an `XMLType` table using C code, by creating an `XMLType` instance given a DOM.

Example 3–7 Inserting Content into an XMLType Table using C

```
#include "stdio.h"
#include <xml.h>
#include <stdlib.h>
#include <string.h>
#include <ociximldb.h>
OCIEnv *envhp;
OCIError *errhp;
OCISvcCtx *svchp;
OCIStmt *stmthp;
OCIServer *srvhp;
OCIDuration dur;
OCISession *sesshp;
oratext *username = "QUINE";
oratext *password = "*************";         /* Replace with the real password. */
oratext *filename = "AMCEWEN-20021009123336171PDT.xml";
oratext *schemaloc = "http://localhost:8080/source/schemas/poSource/xsd/purchaseOrder.xsd";

/*-------------------------------------------------------*/
/* Execute a SQL statement that binds XML data           */
/*-------------------------------------------------------*/

sword exec_bind_xml(OCISvcCtx *svchp, OCIError *errhp, OCIStmt *stmthp,
                    void *xml,        OCIType *xmltdo, OraText *sqlstmt)
{
  OCIBind *bndhp1 = (OCIBind *) 0;
  sword  status = 0;
  OCIInd ind = OCI_IND_NOTNULL;
  OCIInd *indp = &ind;
  if(status = OCIStmtPrepare(stmthp, errhp, (OraText *)sqlstmt,
                             (ub4)strlen((const char *)sqlstmt),
                             (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT))
    return OCI_ERROR;
  if(status = OCIBindByPos(stmthp, &bndhp1, errhp, (ub4) 1, (dvoid *) 0,
                           (sb4) 0, SQLT_NTY, (dvoid *) 0, (ub2 *)0,
                           (ub2 *)0, (ub4) 0, (ub4 *) 0, (ub4) OCI_DEFAULT))
    return OCI_ERROR;
  if(status = OCIBindObject(bndhp1, errhp, (CONST OCIType *) xmltdo,
                            (dvoid **) &xml, (ub4 *) 0,
                            (dvoid **) &indp, (ub4 *) 0))
    return OCI_ERROR;
  if(status = OCIStmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
                             (CONST OCISnapshot*) 0, (OCISnapshot*) 0,
                             (ub4) OCI_DEFAULT))
    return OCI_ERROR;
  return OCI_SUCCESS;
}

/*-------------------------------------------------------*/
/* Initialize OCI handles, and connect                   */
/*-------------------------------------------------------*/

sword init_oci_connect()
{
. . .
}

/*-------------------------------------------------------*/
/* Free OCI handles, and disconnect                      */
/*-------------------------------------------------------*/

void free_oci()
{
. . .
}
```

```
void main()
{
  OCIType *xmltdo;
  xmldocnode  *doc;
  ocixmldbparam params[1];
  xmlerr       err;
  xmlctx  *xctx;
  oratext *ins_stmt;
  sword    status;
  xmlnode *root;
  oratext buf[10000];

  /* Initialize envhp, svchp, errhp, dur, stmthp */
  init_oci_connect();

  /* Get an XML context */
  params[0].name_ocixmldbparam = XCTXINIT_OCIDUR;
  params[0].value_ocixmldbparam = &dur;
  xctx = OCIXmlDbInitXmlCtx(envhp, svchp, errhp, params, 1);
  if (!(doc = XmlLoadDom(xctx, &err, "file", filename,
                         "schema_location", schemaloc, NULL)))
    {
      printf("Parse failed.\n");
      return;
    }
  else
    printf("Parse succeeded.\n");
  root = XmlDomGetDocElem(xctx, doc);
  printf("The xml document is :\n");
  XmlSaveDom(xctx, &err, (xmlnode *)doc, "buffer", buf, "buffer_length", 10000, NULL);
  printf("%s\n", buf);

  /* Insert the document into my_table */
  ins_stmt = (oratext *)"insert into purchaseorder values (:1)";
  status = OCITypeByName(envhp, errhp, svchp, (const text *) "SYS",
                         (ub4) strlen((const char *)"SYS"), (const text *) "XMLTYPE",
                         (ub4) strlen((const char *)"XMLTYPE"), (CONST text *) 0,
                         (ub4) 0, OCI_DURATION_SESSION, OCI_TYPEGET_HEADER,
                         (OCIType **) &xmltdo);
  if (status == OCI_SUCCESS)
    {
      status = exec_bind_xml(svchp, errhp, stmthp, (void *)doc,
                             xmltdo, ins_stmt);
    }
  if (status == OCI_SUCCESS)
    printf ("Insert successful\n");
  else
    printf ("Insert failed\n");

  /* Free XML instances */
  if (doc)
    XmlFreeDocument((xmlctx *)xctx, (xmldocnode *)doc);
  /* Free XML CTX */
  OCIXmlDbFreeXmlCtx(xctx);
  free_oci();
}
```

Note: For simplicity in demonstrating this feature, this example does not perform the password management techniques that a deployed system normally uses. In a production environment, follow the Oracle Database password management guidelines, and disable any sample accounts. See *Oracle Database Security Guide* for password management guidelines and other security recommendations.

See Also: Appendix A, "Oracle-Supplied XML Schemas and Examples" for a complete listing of this example

Loading Large XML Files That Contain Small XML Documents

When loading large XML files consisting of a collection of smaller XML documents, it is often more efficient to use Simple API for XML (SAX) parsing to break the file into a set of smaller documents, and then insert those documents. SAX is an XML standard interface provided by XML parsers for event-based applications.

You can use SAX to load a database table from very large XML files in the order of 30 MB or larger, by creating individual documents from a collection of nodes. You can also bulk load XML files.

See Also:

- `http://www.saxproject.org/` for information about SAX
- `http://www.oracle.com/technology/sample_code/tech/xml/xmldb/content.html`, "SAX Loader Application", for an example of how to load large files using SAX

Loading Large XML Files using SQL*Loader

Use SQL*Loader to load large amounts of XML data into Oracle Database. SQL*Loader loads in one of two modes, conventional or direct path. Table 3–1 compares these modes.

Table 3–1 SQL*Loader – Conventional and Direct-Path Load Modes

Conventional Load Mode	Direct-Path Load Mode
Uses SQL to load data into Oracle Database. This is the *default* mode.	Bypasses SQL and streams the data directly into Oracle Database.
Advantage: Follows SQL semantics. For example triggers are fired and constraints are checked.	*Advantage:* This loads data much faster than the conventional load mode.
Disadvantage: This loads data slower than with the direct load mode.	*Disadvantage:* SQL semantics is not obeyed. For example triggers are not fired and constraints are not checked.

When loading LOBs with SQL*Loader direct-path load, much memory can be used. If the message `SQL*Loader 700 (out of memory)` appears, then it is likely that more rows are being included in each load call than can be handled by your operating system and process memory. *Workaround:* use the `ROWS` option to read a smaller number of rows in each data save.

See Also: Chapter 35, "Loading XML Data using SQL*Loader"

Loading XML Documents into the Repository using DBMS_XDB

You can also store XML documents in Oracle XML DB Repository, and access these documents using path-based rather than table-based techniques. To load an XML document into the repository under a given path, use PL/SQL function `DBMS_XDB.createResource`. Example 3–8 illustrates this.

Example 3–8 Inserting XML Content into the Repository using CREATERESOURCE

```
DECLARE
```

```
  res BOOLEAN;
BEGIN
  res := DBMS_XDB.createResource('/home/QUINE/purchaseOrder.xml',
                                 bfilename('XMLDIR', 'purchaseOrder.xml'),
                                 nls_charset_id('AL32UTF8'));
END;
/
```

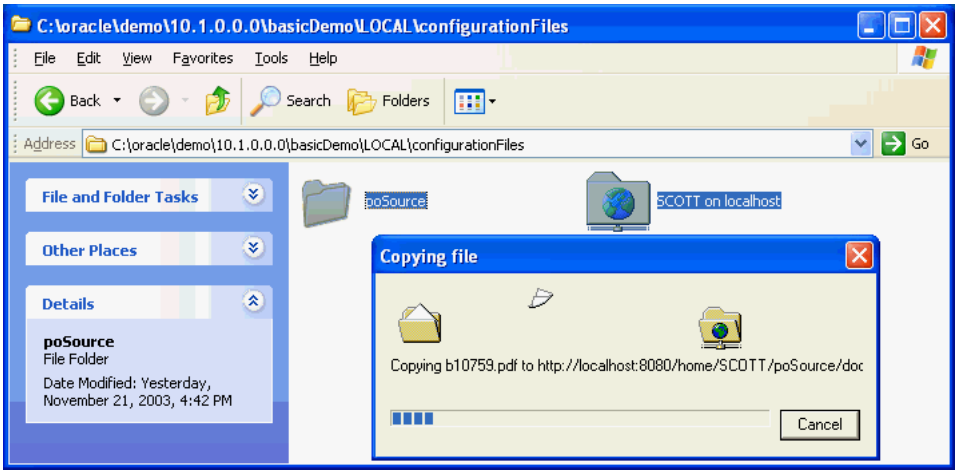
Many operations for configuring and using Oracle XML DB are based on processing one or more XML documents. Examples include registering an XML schema and performing an XSL transformation. The easiest way to make these XML documents available to Oracle Database is to load them into Oracle XML DB Repository.

Loading Documents into the Repository using Protocols

Oracle XML DB Repository can store XML documents that are either XML schema-based or non-schema-based. It can also store content that is not XML data, such as HTML files, image files, and Microsoft Word documents.

You can load XML documents from a local file system into Oracle XML DB Repository using protocols such as WebDAV, from Windows Explorer or other tools that support WebDAV. Figure 3–1 shows a simple drag and drop operation for copying the contents of the `SCOTT` folder from the local hard drive to folder `poSource` in the Oracle XML DB Repository.

Figure 3–1 Loading Content into the Repository using Windows Explorer



The copied folder might contain, for example, an XML schema document, an HTML page, and some XSLT style sheets.

Character Sets of XML Documents

This section describes how character sets of XML documents are determined.

Caution: *AL32UTF8* is the Oracle Database character set that is appropriate for `XMLType` data. It is equivalent to the IANA registered standard UTF-8 encoding, which supports all valid XML characters.

Do not confuse Oracle Database database character set UTF8 (no hyphen) with database character set AL32UTF8 or with character *encoding* UTF-8. Database character set UTF8 has been *superseded* by AL32UTF8. Do *not* use UTF8 for XML data. Character set UTF8 supports only Unicode version 3.1 and earlier. It does not support all valid XML characters. AL32UTF8 has no such limitation.

Using database character set UTF8 for XML data could potentially *stop a system or affect security negatively*. If a character that is not supported by the database character set appears in an input-document element name, a replacement character (usually "?") is substituted for it. This terminates parsing and raises an exception. It can cause an irrecoverable error.

XML Encoding Declaration

Each XML document is composed of units called entities. Each entity in an XML document may use a different encoding for its characters. Entities that are stored in an encoding other than UTF-8 or UTF-16 must begin with an XML declaration containing an encoding specification indicating the character encoding in use. For example:

```
<?xml version='1.0' encoding='EUC-JP' ?>
```

Entities encoded in UTF-16 must begin with the Byte Order Mark (BOM), as described in Appendix F of the XML 1.0 Reference. For example, on big-endian platforms, the BOM required of a UTF-16 data stream is `#xFEFF`.

In the absence of both the encoding declaration and the BOM, the XML entity is assumed to be encoded in UTF-8. Because ASCII is a subset of UTF-8, ASCII entities do not require an encoding declaration.

In many cases, external sources of information are available, besides the XML data, to provide the character encoding in use. For example, the encoding of the data can be obtained from the `charset` parameter of the `Content-Type` field in an HTTP(S) request as follows:

```
Content-Type: text/xml; charset=ISO-8859-4
```

Character-Set Determination When Loading XML Documents into the Database

In releases prior to Oracle Database 10*g* release 1, all XML documents were assumed to be in the *database* character set, regardless of the document encoding declaration. Starting with Oracle Database 10*g* release 1, the document encoding is detected from the encoding declaration when the document is loaded into the database.

However, if the XML data is obtained from a `CLOB` or `VARCHAR` value, then the encoding declaration is *ignored*, because these two data types are always encoded in the database character set.

In addition, when loading data into Oracle XML DB, either through programmatic APIs or transfer protocols, you can provide external encoding to override the document encoding declaration. An error is raised if you try to load a schema-based XML document that contains characters that are not legal in the determined encoding.

The following examples show different ways to specify external encoding:

- Using PL/SQL function `DBMS_XDB.createResource` to create a file resource from a `BFILE`, you can specify the file encoding with the *`CSID`* argument. If a zero *`CSID`* is specified then the file encoding is auto-detected from the document encoding declaration.

```
CREATE DIRECTORY xmldir AS '/private/xmldir';
CREATE OR REPLACE PROCEDURE loadXML(filename VARCHAR2, file_csid NUMBER) IS
  xbfile  BFILE;
  RET     BOOLEAN;
BEGIN
  xbfile := bfilename('XMLDIR', filename);
  ret := DBMS_XDB.createResource('/public/mypurchaseorder.xml',
                                 xbfile,
                                 file_csid);
END;
/
```

- Use the FTP protocol to load documents into Oracle XML DB. Use the `quote set_charset` FTP command to indicate the encoding of the files to be loaded.

```
ftp> quote set_charset Shift_JIS
ftp> put mypurchaseorder.xml
```

- Use the HTTP(S) protocol to load documents into Oracle XML DB. Specify the encoding of the data to be transmitted to Oracle XML DB in the request header.

```
Content-Type: text/xml; charset= EUC-JP
```

Character-Set Determination When Retrieving XML Documents from the Database

XML documents stored in Oracle XML DB can be retrieved using a SQL client, programmatic APIs, or transfer protocols. You can specify the encoding of the retrieved data (except in Oracle Database releases prior to 10*g*, where XML data is retrieved only in the database character set).

When XML data is stored as a `CLOB` or `VARCHAR2` value, the encoding declaration, if present, is always ignored for retrieval, just as for storage. The encoding of a retrieved document can thus be different from the encoding explicitly declared in that document.

The character set for an XML document retrieved from the database is determined in the following ways:

- SQL client – If a SQL client (such as SQL*Plus) is used to retrieve XML data, then the character set is determined by the client-side environment variable `NLS_LANG`. In particular, this setting overrides any explicit character-set declarations in the XML data itself.

  For example, if you set the client side `NLS_LANG` variable to `AMERICAN_AMERICA.AL32UTF8` and then retrieve an XML document with encoding `EUC_JP` provided by declaration `<?xml version="1.0" encoding="EUC-JP"?>`, the character set of the retrieved document is `AL32UTF8`, *not* `EUC_JP`.

  See Also: *Oracle Database Globalization Support Guide* for information about `NLS_LANG`

- PL/SQL and APIs – Using PL/SQL or programmatic APIs, you can retrieve XML data into VARCHAR, CLOB, or XMLType data types. As for SQL clients, you can control the encoding of the retrieved data by setting NLS_LANG.

  You can also retrieve XML data into a BLOB value using XMLType and URIType methods. These let you specify the character set of the returned BLOB value. Here is an example:

```
CREATE OR REPLACE FUNCTION getXML(pathname VARCHAR2, charset VARCHAR2)
  RETURN BLOB IS
  xblob BLOB;
BEGIN
  SELECT XMLSERIALIZE(DOCUMENT e.RES AS BLOB ENCODING charset) INTO xblob
    FROM RESOURCE_VIEW e WHERE equals_path(e.RES, pathname) = 1;
  RETURN xblob;
END;
/
```

- FTP – You can use the FTP quote set_nls_locale command to set the character set:

```
ftp> quote set_nls_locale EUC-JP
ftp> get mypurchaseorder.xml
```

  See Also: FTP Quote Methods on page 28-10

- HTTP(S) – You can use the Accept-Charset parameter in an HTTP(S) request:

```
/httptest/mypurchaseorder.xml  1.1 HTTP/Host: localhost:2345
Accept: text/*
Accept-Charset:  iso-8859-1, utf-8
```

  See Also: Controlling Character Sets for HTTP(S) on page 28-20

Overview of the W3C XML Schema Recommendation

The W3C XML Schema Recommendation defines a standardized language for specifying the structure, content, and certain semantics of a set of XML documents. An XML schema can be considered the metadata that describes a class of XML documents. The XML Schema Recommendation is described at:
http://www.w3.org/TR/xmlschema-0/

XML Instance Documents

Documents conforming to a given XML schema can be considered as members or instances of the class defined by that XML schema. Consequently the term *instance document* is often used to describe an XML document that conforms to a given XML schema. The most common use of an XML schema is to validate that a given instance document conforms to the rules defined by the XML schema.

XML Schema for Schemas

The W3C Schema working group publishes an XML schema, often referred to as the "Schema for Schemas". This XML schema provides the definition, or vocabulary, of the XML Schema language. All valid XML schemas can be considered to be members of the class defined by this XML schema. An XML schema is thus an XML document that

conforms to the class defined by the XML schema published at http://www.w3.org/2001/XMLSchema.

Editing XML Schemas

XML schemas can be authored and edited using any of the following:

- A simple text editor, such as emacs or vi
- An XML schema-aware editor, such as the XML editor included with Oracle JDeveloper
- An explicit XML schema-authoring tool, such as XMLSpy from Altova Corporation

XML Schema Features

The XML Schema language defines 47 scalar data types. This provides for strong typing of elements and attributes. The W3C XML Schema Recommendation also supports object-oriented techniques such as inheritance and extension, hence you can design XML schema with complex objects from base data types defined by the XML Schema language. The vocabulary includes constructs for defining and ordering, default values, mandatory content, nesting, repeated sets, and redefines. Oracle XML DB supports all the constructs, except for redefines.

Text Representation of the Purchase Order XML Schema

Example 3–9 shows the purchase order XML schema as an XML file, `purchaseOrder.xsd`.

Example 3–9 Purchase-Order XML Schema, purchaseOrder.xsd

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" version="1.0">
  <xs:element name="PurchaseOrder" type="PurchaseOrderType"/>
  <xs:complexType name="PurchaseOrderType">
    <xs:sequence>
      <xs:element name="Reference" type="ReferenceType"/>
      <xs:element name="Actions" type="ActionsType"/>
      <xs:element name="Reject" type="RejectionType" minOccurs="0"/>
      <xs:element name="Requestor" type="RequestorType"/>
      <xs:element name="User" type="UserType"/>
      <xs:element name="CostCenter" type="CostCenterType"/>
      <xs:element name="ShippingInstructions" type="ShippingInstructionsType"/>
      <xs:element name="SpecialInstructions" type="SpecialInstructionsType"/>
      <xs:element name="LineItems" type="LineItemsType"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="LineItemsType">
    <xs:sequence>
      <xs:element name="LineItem" type="LineItemType" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="LineItemType">
    <xs:sequence>
      <xs:element name="Description" type="DescriptionType"/>
      <xs:element name="Part" type="PartType"/>
    </xs:sequence>
    <xs:attribute name="ItemNumber" type="xs:integer"/>
  </xs:complexType>
  <xs:complexType name="PartType">
```

```
  <xs:attribute name="Id">
    <xs:simpleType>
      <xs:restriction base="xs:string">
        <xs:minLength value="10"/>
        <xs:maxLength value="14"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:attribute>
  <xs:attribute name="Quantity" type="moneyType"/>
  <xs:attribute name="UnitPrice" type="quantityType"/>
</xs:complexType>
<xs:simpleType name="ReferenceType">
  <xs:restriction base="xs:string">
    <xs:minLength value="18"/>
    <xs:maxLength value="30"/>
  </xs:restriction>
</xs:simpleType>
<xs:complexType name="ActionsType">
  <xs:sequence>
    <xs:element name="Action" maxOccurs="4">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="User" type="UserType"/>
          <xs:element name="Date" type="DateType" minOccurs="0"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="RejectionType">
  <xs:all>
    <xs:element name="User" type="UserType" minOccurs="0"/>
    <xs:element name="Date" type="DateType" minOccurs="0"/>
    <xs:element name="Comments" type="CommentsType" minOccurs="0"/>
  </xs:all>
</xs:complexType>
<xs:complexType name="ShippingInstructionsType">
  <xs:sequence>
    <xs:element name="name" type="NameType" minOccurs="0"/>
    <xs:element name="address" type="AddressType" minOccurs="0"/>
    <xs:element name="telephone" type="TelephoneType" minOccurs="0"/>
  </xs:sequence>
</xs:complexType>
<xs:simpleType name="moneyType">
  <xs:restriction base="xs:decimal">
    <xs:fractionDigits value="2"/>
    <xs:totalDigits value="12"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="quantityType">
  <xs:restriction base="xs:decimal">
    <xs:fractionDigits value="4"/>
    <xs:totalDigits value="8"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="UserType">
  <xs:restriction base="xs:string">
    <xs:minLength value="0"/>
    <xs:maxLength value="10"/>
  </xs:restriction>
```

```
</xs:simpleType>
<xs:simpleType name="RequestorType">
  <xs:restriction base="xs:string">
    <xs:minLength value="0"/>
    <xs:maxLength value="128"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="CostCenterType">
  <xs:restriction base="xs:string">
    <xs:minLength value="1"/>
    <xs:maxLength value="4"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="VendorType">
  <xs:restriction base="xs:string">
    <xs:minLength value="0"/>
    <xs:maxLength value="20"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="PurchaseOrderNumberType">
  <xs:restriction base="xs:integer"/>
</xs:simpleType>
<xs:simpleType name="SpecialInstructionsType">
  <xs:restriction base="xs:string">
    <xs:minLength value="0"/>
    <xs:maxLength value="2048"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="NameType">
  <xs:restriction base="xs:string">
    <xs:minLength value="1"/>
    <xs:maxLength value="20"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="AddressType">
  <xs:restriction base="xs:string">
    <xs:minLength value="1"/>
    <xs:maxLength value="256"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="TelephoneType">
  <xs:restriction base="xs:string">
    <xs:minLength value="1"/>
    <xs:maxLength value="24"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="DateType">
  <xs:restriction base="xs:date"/>
</xs:simpleType>
<xs:simpleType name="CommentsType">
  <xs:restriction base="xs:string">
    <xs:minLength value="1"/>
    <xs:maxLength value="2048"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="DescriptionType">
  <xs:restriction base="xs:string">
    <xs:minLength value="1"/>
    <xs:maxLength value="256"/>
  </xs:restriction>
```

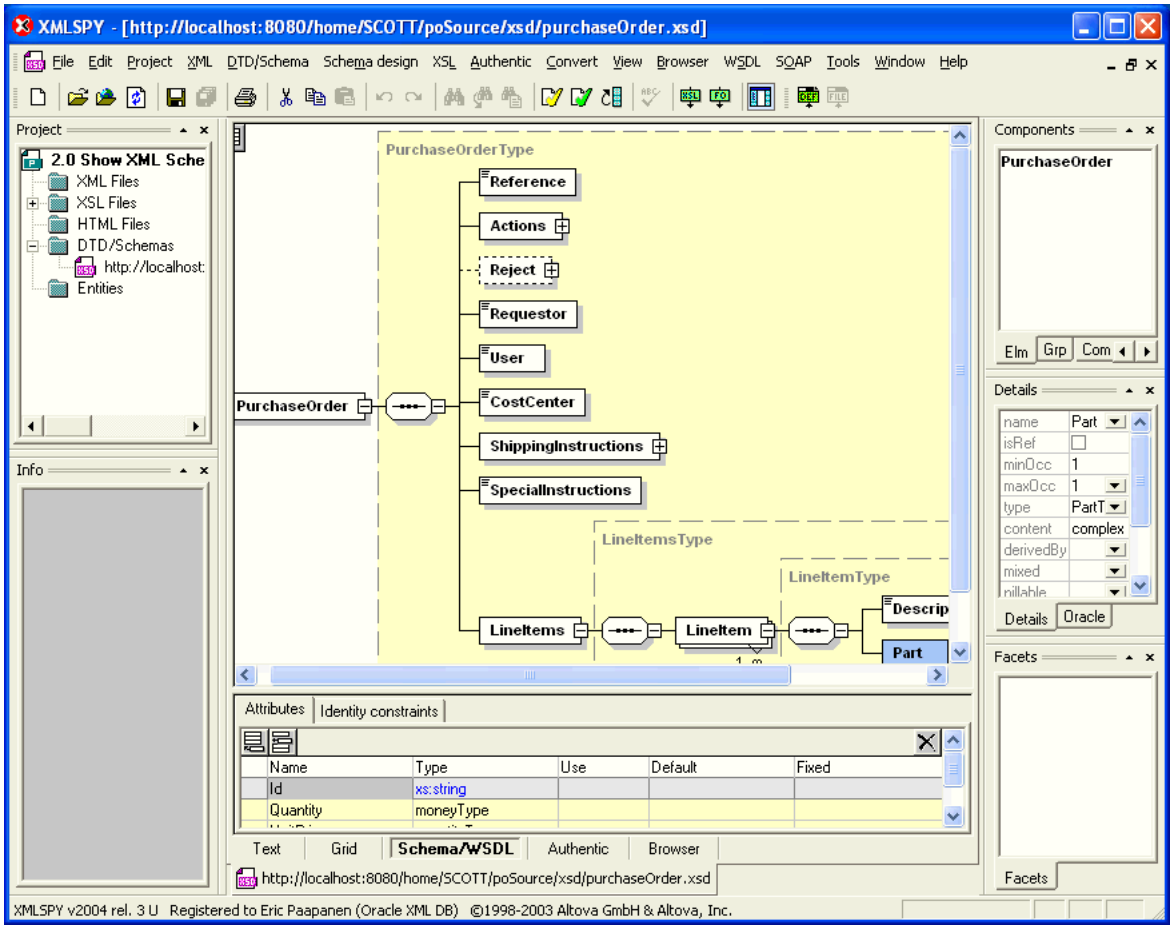
```
  </xs:simpleType>
</xs:schema>
```

See Also: Example 3–10, "Annotated Purchase-Order XML Schema, purchaseOrder.xsd" on page 3-20

Graphical Representation of the Purchase-Order XML Schema

Figure 3–2 shows the purchase-order XML schema displayed using XMLSpy. XMLSpy is a graphical and user-friendly tool from Altova Corporation for creating and editing XML schema and XML documents. See `http://www.altova.com` for details. XMLSpy also supports WebDAV and FTP protocols hence can directly access and edit content stored in Oracle XML DB Repository.

Figure 3–2 XMLSpy Graphical Representation of the PurchaseOrder XML Schema



The purchase order XML schema demonstrates some key features of a typical XML document:

- Global element `PurchaseOrder` is an instance of the `complexType` `PurchaseOrderType`
- `PurchaseOrderType` defines the set of nodes that make up a `PurchaseOrder` element
- `LineItems` element consists of a collection of `LineItem` elements
- Each `LineItem` element consists of two elements: `Description` and `Part`

- `Part` element has attributes `Id`, `Quantity`, and `UnitPrice`

Using XML Schema with Oracle XML DB

This section describes the use of XML Schema with Oracle XML DB.

Why Use XML Schema with Oracle XML DB?

The following paragraphs describe the main reasons for using XML schema with Oracle XML DB.

Validating Instance Documents with XML Schema

The most common usage of XML Schema is as a mechanism for validating that instance documents conform to a given XML schema. The `XMLType` methods `isSchemaValid()` and `schemaValidate()` validate the contents of an instance document stored as `XMLType`.

Constraining Instance Documents for Business Rules or Format Compliance

An XML schema can also be used as a constraint when creating tables or columns of `XMLType`. For example, the `XMLType` is constrained to storing XML documents compliant with one of the global elements defined by the XML schema.

Defining How XMLType Contents Must be Stored in the Database

Oracle XML DB also uses XML Schema as a mechanism for defining how the contents of an `XMLType` instance should be stored inside the database. All storage models support the use of XML Schema: binary XML, structured, unstructured, and hybrid (a combination of structured and unstructured). See "XMLType Storage Models" on page 1-14 for information on the available storage models for `XMLType`.

Structured Storage of XML Documents

Structured storage of XML documents is based on decomposing the content of the document into a set of SQL objects. These SQL objects are based on the SQL 1999 Type framework. When an XML schema is registered with Oracle XML DB, the required SQL type definitions are automatically generated from the XML schema.

A SQL type definition is generated from each `complexType` defined by the XML schema. Each element or attribute defined by the `complexType` becomes a SQL attribute in the corresponding SQL type. Oracle XML DB automatically maps the 47 scalar data types defined by the XML Schema Recommendation to the 19 scalar data types supported by SQL. A varray type is generated for each element and this can occur multiple times.

The generated SQL types allow XML content, compliant with the XML schema, to be decomposed and stored in the database as a set of objects without any loss of information. When the document is ingested the constructs defined by the XML schema are mapped directly to the equivalent SQL types.

This lets Oracle XML DB leverage the full power of Oracle Database when managing XML and can lead to significant reductions in the amount of space required to store the document. It can also reduce the amount of memory required to query and update XML content.

Annotating an XML Schema to Control Naming, Mapping, and Storage

The W3C XML Schema Recommendation defines an annotation mechanism that lets vendor-specific information be added to an XML schema. Oracle XML DB uses this mechanism to control the mapping between the XML schema and database features.

You can use XML schema annotations to do the following:

- Specify which database tables are used to store the XML data.
- Override the default mapping between XML Schema data types and either binary XML encoding types or, for structured storage, SQL data types.
- Name the database objects and attributes that are created to store XML data (for structured storage).

Controlling How Collections Are Stored for Object-Relational XMLType Storage

When you register an XML schema for data that is stored object-relationally and you set registration parameter `GENTABLES` to `TRUE`, default tables are created automatically to store the associated XML instance documents.

Order is preserved among XML collection elements when they are stored. The result is an **ordered collection**.[2] You can store data in an ordered collection in these ways:

- **Varray in a table**. Each element in the collection is mapped to a SQL object. The collection of SQL objects is stored as a set of rows in a table, called an **ordered collection table** (**OCT**). By default, all collections are stored in OCTs. This default behavior corresponds to the XML schema annotation `xdb:storeVarrayAsTable = "true"` (default value).
- **Varray in a LOB**. Each element in the collection is mapped to a SQL object. The entire collection of SQL objects is serialized as a varray and stored in a LOB column. To store a given collection as a varray in a LOB, use XML schema annotation `xdb:storeVarrayAsTable = "false"`.

You can also use out-of-line storage for an ordered collection. This corresponds to XML schema annotation `SQLInline = "false"`, and it means that a varray of `REF`s in the collection table or LOB tracks the collection content, which is stored out of line.

There is no requirement to annotate an XML schema before using it. Oracle XML DB uses a set of default assumptions when processing an XML schema that contains no annotations.

If you do not supply any of the annotations mentioned in this section, then Oracle XML DB stores a collection as a *heap-based* OCT. You can force OCTs to be stored as **index-organized tables** (**IOTs**) instead, by passing **`REGISTER_NT_AS_IOT`** in the `OPTIONS` parameter of `DBMS_XMLSCHEMA.registerSchema`.

[2] If you use XML schema annotation `maintainOrder = "false"`, then an unordered collection is used instead of an ordered collection. Oracle recommends that you use ordered collections (`maintainOrder = "true"`) for XML data, to preserve document order. By default, attribute `maintainOrder` is `true`.

Note: Use heap-based OCTs, *not* IOTs, unless you are explicitly advised by Oracle to use IOTs. IOT storage has these significant limitations:

- It disables partitioning of the collection tables (IOTs).
- It supports only document-level Oracle Text indexes. It disables indexes that are element-specific or attribute-specific.

See also: Chapter 12, "Full-Text Search Over XML Data" for information about using Oracle Text with XML data.

Note: In releases prior to Oracle Database 11g Release 1:

- OCTs were stored as IOTs by default.
- The default value for `xdb:storeVarrayAsTable` was `false`.

See Also:

- "Structured Storage of XML Schema-Based Data" on page 7-32 for information about collection storage when you create `XMLType` tables and columns manually using structured storage
- Chapter 7, "XML Schema Storage and Query: Basic"
- "Setting Annotation Attribute SQLInline to false for Out-Of-Line Storage" on page 9-4
- Partitioning XMLType Tables and Columns Stored Object-Relationally on page 9-10

Declaring the Oracle XML DB Namespace

Before annotating an XML schema you must first declare the Oracle XML DB namespace. The Oracle XML DB namespace is defined as:

```
http://xmlns.oracle.com/xdb
```

The namespace is declared in the XML schema by adding a namespace declaration such as the following to the root element of the XML schema:

```
xmlns:xdb="http://xmlns.oracle.com/xdb"
```

Note the use of a namespace prefix (`xdb`). This makes it possible to abbreviate the namespace to `xdb` when adding annotations.

Example 3–10 shows the beginning of the `PurchaseOrder` XML schema with annotations. See Example A–1 on page A-30 for the complete schema listing.

Example 3–10 Annotated Purchase-Order XML Schema, purchaseOrder.xsd

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:xdb="http://xmlns.oracle.com/xdb"
           version="1.0"
           xdb:storeVarrayAsTable="true">
  <xs:element name="PurchaseOrder" type="PurchaseOrderType" xdb:defaultTable="PURCHASEORDER"/>
  <xs:complexType name="PurchaseOrderType" xdb:SQLType="PURCHASEORDER_T">
    <xs:sequence>
      <xs:element name="Reference" type="ReferenceType" minOccurs="1" xdb:SQLName="REFERENCE"/>
```

```
    <xs:element name="Actions" type="ActionsType" xdb:SQLName="ACTIONS"/>
    <xs:element name="Reject" type="RejectionType" minOccurs="0" xdb:SQLName="REJECTION"/>
    <xs:element name="Requestor" type="RequestorType" xdb:SQLName="REQUESTOR"/>
    <xs:element name="User" type="UserType" minOccurs="1" xdb:SQLName="USERID"/>
    <xs:element name="CostCenter" type="CostCenterType" xdb:SQLName="COST_CENTER"/>
    <xs:element name="ShippingInstructions" type="ShippingInstructionsType"
                xdb:SQLName="SHIPPING_INSTRUCTIONS"/>
    <xs:element name="SpecialInstructions" type="SpecialInstructionsType"
                xdb:SQLName="SPECIAL_INSTRUCTIONS"/>
    <xs:element name="LineItems" type="LineItemsType" xdb:SQLName="LINEITEMS"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="LineItemsType" xdb:SQLType="LINEITEMS_T">
  <xs:sequence>
    <xs:element name="LineItem" type="LineItemType" maxOccurs="unbounded"
                xdb:SQLName="LINEITEM" xdb:SQLCollType="LINEITEM_V"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="LineItemType" xdb:SQLType="LINEITEM_T">
  <xs:sequence>
    <xs:element name="Description" type="DescriptionType"
                xdb:SQLName="DESCRIPTION"/>
    <xs:element name="Part" type="PartType" xdb:SQLName="PART"/>
  </xs:sequence>
  <xs:attribute name="ItemNumber" type="xs:integer" xdb:SQLName="ITEMNUMBER"
                xdb:SQLType="NUMBER"/>
</xs:complexType>
<xs:complexType name="PartType" xdb:SQLType="PART_T">
  <xs:attribute name="Id" xdb:SQLName="PART_NUMBER" xdb:SQLType="VARCHAR2">
    <xs:simpleType>
      <xs:restriction base="xs:string">
        <xs:minLength value="10"/>
        <xs:maxLength value="14"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:attribute>
  <xs:attribute name="Quantity" type="moneyType" xdb:SQLName="QUANTITY"/>
  <xs:attribute name="UnitPrice" type="quantityType" xdb:SQLName="UNITPRICE"/>
</xs:complexType>
<xs:simpleType name="ReferenceType">
  <xs:restriction base="xs:string">
    <xs:minLength value="18"/>
    <xs:maxLength value="30"/>
  </xs:restriction>
</xs:simpleType>
<xs:complexType name="ActionsType" xdb:SQLType="ACTIONS_T">
  <xs:sequence>
    <xs:element name="Action" maxOccurs="4" xdb:SQLName="ACTION" xdb:SQLCollType="ACTION_V">
      <xs:complexType xdb:SQLType="ACTION_T">
        <xs:sequence>
          <xs:element name="User" type="UserType" xdb:SQLName="ACTIONED_BY"/>
          <xs:element name="Date" type="DateType" minOccurs="0" xdb:SQLName="DATE_ACTIONED"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="RejectionType" xdb:SQLType="REJECTION_T">
  <xs:all>
    <xs:element name="User" type="UserType" minOccurs="0" xdb:SQLName="REJECTED_BY"/>
    <xs:element name="Date" type="DateType" minOccurs="0" xdb:SQLName="DATE_REJECTED"/>
    <xs:element name="Comments" type="CommentsType" minOccurs="0" xdb:SQLName="REASON_REJECTED"/>
  </xs:all>
</xs:complexType>
<xs:complexType name="ShippingInstructionsType" xdb:SQLType="SHIPPING_INSTRUCTIONS_T">
  <xs:sequence>
```

```
    <xs:element name="name" type="NameType" minOccurs="0" xdb:SQLName="SHIP_TO_NAME"/>
    <xs:element name="address" type="AddressType" minOccurs="0" xdb:SQLName="SHIP_TO_ADDRESS"/>
    <xs:element name="telephone" type="TelephoneType" minOccurs="0" xdb:SQLName="SHIP_TO_PHONE"/>
  </xs:sequence>
</xs:complexType>
<xs:simpleType name="moneyType">
  <xs:restriction base="xs:decimal">
    <xs:fractionDigits value="2"/>
    <xs:totalDigits value="12"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="quantityType">
  <xs:restriction base="xs:decimal">
    <xs:fractionDigits value="4"/>
    <xs:totalDigits value="8"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="UserType">
  <xs:restriction base="xs:string">
    <xs:minLength value="0"/>
    <xs:maxLength value="10"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="RequestorType">
  <xs:restriction base="xs:string">
    <xs:minLength value="0"/>
    <xs:maxLength value="128"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="CostCenterType">
  <xs:restriction base="xs:string">
    <xs:minLength value="1"/>
    <xs:maxLength value="4"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="VendorType">
  <xs:restriction base="xs:string">
    <xs:minLength value="0"/>
    <xs:maxLength value="20"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="PurchaseOrderNumberType">
  <xs:restriction base="xs:integer"/>
</xs:simpleType>
<xs:simpleType name="SpecialInstructionsType">
  <xs:restriction base="xs:string">
    <xs:minLength value="0"/>
    <xs:maxLength value="2048"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="NameType">
  <xs:restriction base="xs:string">
    <xs:minLength value="1"/>
    <xs:maxLength value="20"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="AddressType">
  <xs:restriction base="xs:string">
    <xs:minLength value="1"/>
    <xs:maxLength value="256"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="TelephoneType">
  <xs:restriction base="xs:string">
    <xs:minLength value="1"/>
    <xs:maxLength value="24"/>
```

```
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="DateType">
    <xs:restriction base="xs:date"/>
  </xs:simpleType>
  <xs:simpleType name="CommentsType">
    <xs:restriction base="xs:string">
      <xs:minLength value="1"/>
      <xs:maxLength value="2048"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="DescriptionType">
    <xs:restriction base="xs:string">
      <xs:minLength value="1"/>
      <xs:maxLength value="256"/>
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

The `PurchaseOrder` XML schema defines the following two *namespaces*:

- `http://www.w3c.org/2001/XMLSchema`. This is reserved by W3C for the Schema for Schemas.
- `http://xmlns.oracle.com/xdb`. This is reserved by Oracle for the Oracle XML DB schema annotations.

The `PurchaseOrder` schema uses several *annotations*, including the following:

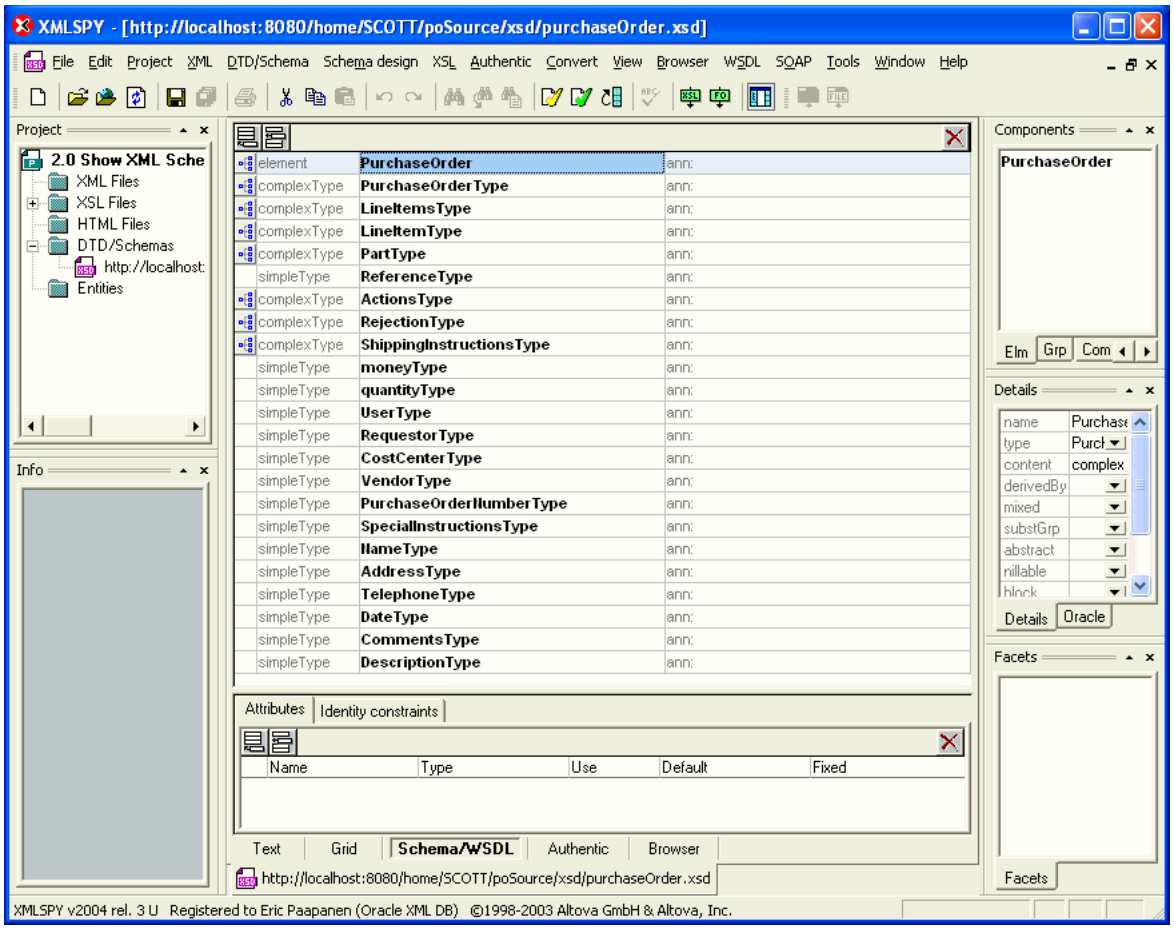
- `defaultTable` annotation in the `PurchaseOrder` element. This specifies that XML documents, compliant with this XML schema are stored in a database table called `purchaseorder`.
- `SQLType` annotation.

  The first occurrence of `SQLType` specifies that the name of the SQL type generated from `complexType` element `PurchaseOrderType` is `purchaseorder_t`.

  The second occurrence of `SQLType` specifies that the name of the SQL type generated from the `complexType` element `LineItemType` is `lineitem_t` and the SQL type that manages the collection of `LineItem` elements is `lineitem_v`.
- `SQLName` annotation. This provides an explicit name for each SQL attribute of `purchaseorder_t`.

Figure 3–3 shows the XMLSpy Oracle tab, which facilitates adding Oracle XML DB schema annotations to an XML schema while working in the graphical editor.

Figure 3–3 XMLSpy Showing Support for Oracle XML DB Schema Annotations



Registering an XML Schema with Oracle XML DB

For an XML schema to be useful to Oracle XML DB you must first register it with Oracle XML DB. After it has been registered, it can be used for validating XML documents and for creating `XMLType` tables and columns bound to the XML schema.

Two items are required to register an XML schema with Oracle XML DB:

- The XML schema document
- A string that can be used as a unique identifier for the XML schema, after it is registered with Oracle Database. Instance documents use this unique identifier to identify themselves as members of the class defined by the XML schema. The identifier is typically in the form of a URL, and is often referred to as the **schema location hint** or **document location hint**.

You register an XML schema using PL/SQL procedure `DBMS_XMLSCHEMA.registerSchema`. Example 3–11 illustrates this. By default, when an XML schema is registered, Oracle XML DB automatically generates all of the SQL object types and `XMLType` tables required to manage the instance documents. An XML schema can be registered as global or local.

See Also:

- "Delete and Reload Documents Before Registering Their XML Schema" on page 7-8 for considerations to keep in mind when you register an XML schema
- "Local and Global XML Schemas" on page 7-14
- *Oracle Database PL/SQL Packages and Types Reference* for information about `DBMS_XMLSCHEMA.registerSchema`

Example 3–11 Registering an XML Schema using DBMS_XMLSCHEMA.REGISTERSCHEMA

```
BEGIN
  DBMS_XMLSCHEMA.registerSchema(
    SCHEMAURL => 'http://localhost:8080/source/schemas/poSource/xsd/purchaseOrder.xsd',
    SCHEMADOC => XDBURIType('/source/schemas/poSource/xsd/purchaseOrder.xsd').getCLOB(),
    LOCAL     => TRUE,
    GENTYPES  => TRUE,
    GENTABLES => TRUE);
END;
/
```

In Example 3–11, the unique identifier for the XML schema is:

```
http://localhost:8080/source/schemas/poSource/xsd/purchaseOrder.xsd
```

The XML schema document was previously loaded into Oracle XML DB Repository at this path: `/source/schemas/poSource/xsd/purchaseOrder.xsd`.

During XML schema registration, an `XDBURIType` accesses the content of the XML schema document, based on its location in the repository. Options passed to procedure `registerSchema` specify that the schema in Example 3–11 is to be registered as a local XML schema, and that SQL objects, and that tables are to be generated during the registration process.

PL/SQL procedure `DBMS_XMLSCHEMA.registerSchema` performs the following operations:

- Parses and validates the XML schema.
- Creates a set of entries in Oracle Data Dictionary that describe the XML schema.
- Creates a set of SQL object definitions, based on `complexType` elements defined in the XML schema.
- Creates an `XMLType` table for each global element defined by the XML schema.

See Also: "Local and Global XML Schemas" on page 7-14

SQL Types and Tables Created During XML Schema Registration

Example 3–12 illustrates the creation of object types during XML schema registration with Oracle XML DB.

Example 3–12 Objects Created During XML Schema Registration

```
DESCRIBE purchaseorder_t
 purchaseorder_t is NOT FINAL
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 SYS_XDBPD$                                         XDB.XDB$RAW_LIST_T
 REFERENCE                                          VARCHAR2(30 CHAR)
 ACTIONS                                            ACTIONS_T
```

```
 REJECTION                                          REJECTION_T
 REQUESTOR                                          VARCHAR2(128 CHAR)
 USERID                                             VARCHAR2(10 CHAR)
 COST_CENTER                                        VARCHAR2(4 CHAR)
 SHIPPING_INSTRUCTIONS                              SHIPPING_INSTRUCTIONS_T
 SPECIAL_INSTRUCTIONS                               VARCHAR2(2048 CHAR)
 LINEITEMS                                          LINEITEMS_T

DESCRIBE lineitems_t
 lineitems_t is NOT FINAL
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 SYS_XDBPD$                                         XDB.XDB$RAW_LIST_T
 LINEITEM                                           LINEITEM_V

DESCRIBE lineitem_v
 lineitem_v VARRAY(2147483647) OF LINEITEM_T
 LINEITEM_T is NOT FINAL
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 SYS_XDBPD$                                         XDB.XDB$RAW_LIST_T
 ITEMNUMBER                                         NUMBER(38)
 DESCRIPTION                                        VARCHAR2(256 CHAR)
 PART                                               PART_T
```

This example shows that SQL type definitions were created when the XML schema was registered with Oracle XML DB. These SQL type definitions include:

- `purchaseorder_t`. This type is used to persist the SQL objects generated from a `PurchaseOrder` element. When an XML document containing a `PurchaseOrder` element is stored in Oracle XML DB the document is broken up, and the contents of the document are stored as an instance of `purchaseorder_t`.
- `lineitems_t`, `lineitem_v`, and `lineitem_t`. These types manage the collection of `LineItem` elements that may be present in a `PurchaseOrder` document. Type `lineitems_t` consists of a single attribute `lineitem`, defined as an instance of type `lineitem_v`. Type `lineitem_v` is defined as a varray of `linteitem_t` objects. There is one instance of the `lineitem_t` object for each `LineItem` element in the document.

Working with Large XML Schemas

Several issues can arise when working with large, complex XML schemas.

Sometimes, you encounter one of these errors when you register an XML schema or you create a table that is based on a global element defined by an XML schema:

- `ORA-01792: maximum number of columns in a table or view is 1000`
- `ORA-04031: unable to allocate` *`string`* `bytes of shared memory ("`*`string`*`","`*`string`*`","`*`string`*`","`*`string`*`")`

These errors are raised when an attempt is made to create an `XMLType` table or column based on a global element and the global element is defined as a `complexType` that contains a very large number of element and attribute definitions.

The errors are raised only when creating an `XMLType` table or column that uses object-relational storage. In this case, the table or column is persisted using a SQL type, and each object attribute defined by the SQL type counts as one column in the underlying table. If the SQL type contains object attributes that are based on other SQL types, then the attributes defined by those types also count as columns in the underlying table.

If the total number of object attributes in all of the SQL types exceeds the Oracle Database limit of 1000 columns in a table, then the storage table cannot be created. When the total number of elements and attributes defined by a `complexType` reaches 1000, it is not possible to create a single table that can manage the SQL objects that are generated when an instance of that type is stored in the database.

> **Tip:** You can use the following query to determine the number of columns for a given `XMLType` table stored object-relationally:
>
> ```
> SELECT count(*) FROM USER_TAB_COLS WHERE TABLE_NAME = '<the table>'
> ```
>
> where *<the table>* is the table you want to check.

Error `ORA-01792` reports that the 1000-column limit has been exceeded. Error `ORA-04031` reports that memory is insufficient during the processing of a large number of element and attribute definitions.

To resolve this problem of having too many element and attribute definitions, you must reduce the total number of object attributes in the SQL types that are used to create the storage tables.

There are two ways to achieve this reduction:

- Use a top-down technique, with *multiple* `XMLType` *tables* that manage the XML documents. This reduces the number of SQL attributes in the SQL type hierarchy for a given storage table. As long as none of the tables need to manage more than 1000 object attributes, the problem is resolved.
- Use a bottom-up technique, which reduces the number of SQL attributes in the SQL type hierarchy, *collapsing some elements and attributes* defined by the XML schema so that they are stored as a single `CLOB` value.

Both techniques rely on annotating the XML schema to define how a particular `complexType` is stored in the database.

For the top-down technique, annotations `SQLInline = "false"` and `defaultTable` force some subelements in the XML document to be stored as rows in a separate `XMLType` table. Oracle XML DB maintains the relationship between the two tables using a `REF` of `XMLType`. Good candidates for this approach are XML schemas that do either of the following:

- Define a *choice*, where each element within the choice is defined as a `complexType`
- Define an element based on a `complexType` that contains a *large number of element and attribute definitions*

The bottom-up technique involves reducing the total number of attributes in the SQL object types by choosing to store some of the lower-level `complexType` elements as `CLOB` values, rather than as objects. This is achieved by annotating the `complexType` or the usage of the `complexType` with `SQLType = "CLOB"`.

Which technique you use depends on the application and the type of queries and updates to be performed against the data.

Working with Global Elements

By default, when an XML schema is registered with the database, Oracle XML DB generates a *default table for each global element* defined by the XML schema.

You can use attribute `xdb:defaultTable` to specify the name of the default table for a given global element. Each `xdb:defaultTable` attribute value you provide must

be *unique* among *all schemas* registered by a given database user. If you do *not* supply a nonempty default table name for some element, then a unique name is provided automatically.

In practice, however, you do *not* want to create a default table for most global elements. Elements that never serve as the root element for an XML instance document do not need default tables—such tables are never used. Creating default tables for all global elements can lead to significant overhead in processor time and space used, especially if an XML schema contains a large number of global element definitions.

As a general rule, then, you want to prevent the creation of a default table for any global element (or any local element stored out of line) that you are sure will *not* be used as a root element in any document. You can do this in one of the following ways:

- Add the annotation `xdb:defaultTable = ""` (empty string) to the definition of *each* global element that will *not* appear as the root element of an XML instance document. Using this approach, you allow automatic default-table creation, in general, and you prohibit it explicitly where needed, using `xdb:defaultTable = ""`.
- Set parameter `GENTABLES` to `false` when registering the XML schema, and then *manually create the default table* for each global element that can legally appear as the root element of an instance document. Using this approach, you inhibit automatic default-table creation, and you create only the tables that are needed, by hand.

Creating XML Schema-Based XMLType Columns and Tables

After an XML schema has been registered with Oracle XML DB, it can be referenced when defining tables that contain `XMLType` columns or creating `XMLType` tables.

If you specify no storage model when creating an XMLType table or column for XML schema-based data, then the storage model used is that specified during registration of the referenced XML schema. If no storage model was specified for the XML schema registration, then object-relational storage is used.

Example 3–13 shows how to manually create table `purchaseorder`, the default table for `PurchaseOrder` elements.

Example 3–13 Creating an XMLType Table that Conforms to an XML Schema

```
CREATE TABLE purchaseorder OF XMLType
  XMLSCHEMA "http://localhost:8080/source/schemas/poSource/xsd/purchaseOrder.xsd"
  ELEMENT "PurchaseOrder"
  VARRAY "XMLDATA"."ACTIONS"."ACTION"
    STORE AS TABLE action_table
                   ((PRIMARY KEY (NESTED_TABLE_ID, SYS_NC_ARRAY_INDEX$)))
  VARRAY "XMLDATA"."LINEITEMS"."LINEITEM"
    STORE AS TABLE lineitem_table
                   ((PRIMARY KEY (NESTED_TABLE_ID, SYS_NC_ARRAY_INDEX$)));
```

Each member of the varray that manages the collection of `Action` elements is stored in the ordered collection table `action_table`. Each member of the varray that manages the collection of `LineItem` elements is stored as a row in ordered collection table `lineitem_table`. The ordered collection tables are heap-based. Because of the `PRIMARY KEY` specification, they automatically contain pseudocolumn `NESTED_TABLE_ID` and column `SYS_NC_ARRAY_INDEX$`, which are required to link them back to the parent column.

This CREATE TABLE statement is equivalent to the CREATE TABLE statement that is generated automatically by Oracle XML DB when you set parameter GENTABLES to TRUE during XML schema registration. By default, the value of XML schema annotation storeVarrayAsTable is true, which automatically generates ordered collection tables (OCTs) for collections during XML schema registration. These OCTs are given system-generated names, which can be difficult to work with. You can give them more meaningful names using the SQL statement RENAME TABLE.

The CREATE TABLE statement in Example 3–13 corresponds to a purchase-order document with a single level of nesting: The varray that manages the collection of LineItem elements is ordered collection table lineitem_table.

What if you had a different XML schema that had, say, a collection of Shipment elements inside a Shipments element that was, in turn, inside a LineItem element? In that case, you could create the table manually as shown in Example 3–14.

Example 3–14 Creating an XMLType Table for Nested Collections

```
CREATE TABLE purchaseorder OF XMLType
  XMLSCHEMA "http://localhost:8080/source/schemas/poSource/xsd/purchaseOrder.xsd"
  ELEMENT "PurchaseOrder"
  VARRAY "XMLDATA"."ACTIONS"."ACTION"
    STORE AS TABLE action_table
                   ((PRIMARY KEY (NESTED_TABLE_ID, SYS_NC_ARRAY_INDEX$)))
                   VARRAY "XMLDATA"."LINEITEMS"."LINEITEM"
    STORE AS TABLE lineitem_table
                   ((PRIMARY KEY (NESTED_TABLE_ID, SYS_NC_ARRAY_INDEX$))
                    VARRAY "SHIPMENTS"."SHIPMENT"
                      STORE AS TABLE shipments_table
                                     ((PRIMARY KEY (NESTED_TABLE_ID,
                                                    SYS_NC_ARRAY_INDEX$))));
```

A SQL*Plus DESCRIBE statement can be used to view information about an XMLType table, as shown in Example 3–15.

Example 3–15 Using DESCRIBE with an XML Schema-Based XMLType Table

```
DESCRIBE purchaseorder
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
TABLE of SYS.XMLTYPE(XMLSchema
"http://localhost:8080/source/schemas/poSource/xsd/purchaseOrder.xsd"
Element "PurchaseOrder") STORAGE Object-relational TYPE "PURCHASEORDER_T"
```

The output of the DESCRIBE statement of Example 3–15 shows the following information about table purchaseorder:

- The table is an XMLType table
- The table is constrained to storing PurchaseOrder documents as defined by the PurchaseOrder XML schema
- Rows in this table are stored as a set of objects in the database
- SQL type purchaseorder_t is the base object for this table

Default Tables

The XML schema in Example 3–13 specifies that the PurchaseOrder table is the default table for PurchaseOrder elements. When an XML document compliant with the XML schema is inserted into Oracle XML DB Repository using protocols or

PL/SQL, the content of the XML document is stored as a row in the `purchaseorder` table.

When an XML schema is registered as a global schema, you must grant the appropriate access rights on the default table to all other users of the database, before they can work with instance documents that conform to the globally registered XML schema.

See Also: "Local and Global XML Schemas" on page 7-14

Identifying XML Schema Instance Documents

Before an XML document can be inserted into an XML schema-based `XMLType` table or column the document must identify the associated XML schema. There are two ways to do this:

- Explicitly identify the XML schema when creating the `XMLType`. This can be done by passing the name of the XML schema to the `XMLType` constructor, or by invoking `XMLType` method `createSchemaBasedXML()`.
- Use the `XMLSchema-instance` mechanism to explicitly provide the required information in the XML document. This option can be used when working with Oracle XML DB.

The advantage of the `XMLSchema-instance` mechanism is that it lets the Oracle XML DB protocol servers recognize that an XML document inserted into Oracle XML DB Repository is an instance of a registered XML schema. The content of the instance document is automatically stored in the default table specified by that XML schema.

The `XMLSchema-instance` mechanism is defined by the W3C XML Schema working group. It is based on adding attributes that identify the target XML schema to the root element of the instance document. These attributes are defined by the `XMLSchema-instance` namespace.

To identify an instance document as a member of the class defined by a particular XML schema you must declare the `XMLSchema-instance` namespace by adding a namespace declaration to the root element of the instance document. For example:

```
xmlns:xsi = http://www.w3.org/2001/XMLSchema-instance
```

Once the `XMLSchema-instance` namespace has been declared and given a `namespace` prefix, attributes that identify the XML schema can be added to the root element of the instance document. In the preceding example, the namespace prefix for the `XMLSchema-instance` namespace was defined as `xsi`. This prefix can then be used when adding the `XMLSchema-instance` attributes to the root element of the instance document.

Which attributes must be added depends on several factors. There are two possibilities, `noNamespaceSchemaLocation` and `schemaLocation`. Depending on the XML schema, one or both of these attributes is required to identify the XML schemas that the instance document is associated with.

Attributes noNamespaceSchemaLocation and schemaLocation

If the target XML schema does not declare a target namespace, the `noNamespaceSchemaLocation` attribute is used to identify the XML schema. The value of the attribute is the *schema location hint*. This is the unique identifier passed to PL/SQL procedure `DBMS_XMLSCHEMA.registerSchema` when the XML schema is registered with the database.

For XML schema `purchaseOrder.xsd`, the correct definition of the root element of the instance document would read as follows:

```
<PurchaseOrder
  xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
  xsi:noNamespaceSchemaLocation=
    "http://localhost:8080/source/schemas/poSource/xsd/purchaseOrder.xsd">
```

If the target XML schema declares a target namespace, then the `schemaLocation` attribute is used to identify the XML schema. The value of this attribute is a pair of values separated by a space:

- the value of the *target namespace* declared in the XML schema
- the *schema location hint*, the unique identifier passed to procedure `DBMS_XMLSCHEMA.registerSchema` when the schema is registered with the database

For example, assume that the `PurchaseOrder` XML schema includes a target namespace declaration. The root element of the schema would look like this:

```
<xs:schema targetNamespace="http://demo.oracle.com/xdb/purchaseOrder"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:xdb="http://xmlns.oracle.com/xdb"
           version="1.0" xdb:storeVarrayAsTable="true">
   <xs:element name="PurchaseOrder" type="PurchaseOrderType"
               xdb:defaultTable="PURCHASEORDER"/>
```

In this case, the correct form of the root element of the instance document would read as follows:

```
<PurchaseOrder
    xnlns="http://demo.oracle.com/xdb/purchaseOrder"
    xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
    xsi:schemaLocation=
      "http://demo.oracle.com/xdb/purchaseOrder
       http://mdrake-lap:8080/source/schemas/poSource/xsd/purchaseOrder.xsd">
```

Dealing with Multiple Namespaces

When the XML schema includes elements defined in multiple namespaces, an entry must occur in the `schemaLocation` attribute for each of the XML schemas. Each entry consists of the namespace declaration and the *schema location hint*. The entries are separated from each other by one or more whitespace characters. If the primary XML schema does not declare a target namespace, then the instance document also needs to include a `noNamespaceSchemaLocation` attribute that provides the *schema location hint* for the primary XML schema.

Enforcing XML Data Integrity using the Database

One advantage of using Oracle XML DB to manage XML content is that SQL can be used to supplement the functionality provided by XML schema. Combining the power of SQL and XML with the ability of the database to enforce rules makes the database a powerful framework for managing XML content.

Only well-formed XML documents can be stored in `XMLType` tables or columns. A **well-formed** XML document is one that conforms to the syntax of the XML version declared in its XML declaration. This includes having a single root element, properly nested tags, and so forth. Additionally, if the `XMLType` table or column is constrained to an XML schema, only documents that conform to that XML schema can be stored in

that table or column. Any attempt to store or insert any other kind of XML document in an XML schema-based `XMLType` raises an error. Example 3–16 illustrates this.

Example 3–16 Error From Attempting to Insert an Incorrect XML Document

```
INSERT INTO purchaseorder
  VALUES (XMLType(bfilename('XMLDIR', 'Invoice.xml'), nls_charset_id('AL32UTF8')))
  VALUES (XMLType(bfilename('XMLDIR', 'Invoice.xml'), nls_charset_id('AL32UTF8')))
         *
ERROR at line 2:
ORA-19007: Schema - does not match expected
http://localhost:8080/source/schemas/poSource/xsd/purchaseOrder.xsd.
```

Such an error only occurs when content is inserted directly into an `XMLType` table. It indicates that Oracle XML DB did not recognize the document as a member of the class defined by the XML schema. For a document to be recognized as a member of the class defined by the schema, the following conditions must be true:

- The name of the XML document root element must match the name of global element used to define the `XMLType` table or column.
- The XML document must include the appropriate attributes from the `XMLSchema-instance` namespace, or the XML document must be explicitly associated with the XML schema using the `XMLType` constructor or `XMLType` method `createSchemaBasedXML()`.

If the constraining XML schema declares a `targetNamespace`, then the instance documents must contain the appropriate namespace declarations to place the root element of the document in the `targetNamespace` defined by the XML schema.

> **Note:** XML constraints are enforced only within individual XML documents. Database (SQL) constraints are enforced across sets of XML documents.

Comparing Partial to Full XML Schema Validation

This section describes the differences between partial and full XML schema validation used when inserting XML documents into the database.

Partial Validation

For binary XML storage, Oracle XML DB performs a full validation whenever an XML document is inserted into an XML schema-based `XMLType` table or column. For all other models of XML storage, Oracle XML DB performs only a partial validation of the document. This is because, except for binary XML storage, complete XML schema validation is quite costly, in terms of performance.

Partial validation ensures only that all of the mandatory elements and attributes are present, and that there are no unexpected elements or attributes in the document. That is, it ensures only that the structure of the XML document conforms to the SQL data type definitions that were derived from the XML schema. Partial validation does not ensure that the instance document is fully compliant with the XML schema.

Example 3–17 provides an example of failing partial validation while inserting an XML document into table `PurchaseOrder`, which is stored object-relationally.

Example 3–17 Error When Inserting Incorrect XML Document (Partial Validation)

```
INSERT INTO purchaseorder
```

```
  VALUES(XMLType(bfilename('XMLDIR', 'InvalidElement.xml'),
                 nls_charset_id('AL32UTF8')));
  VALUES(XMLType(bfilename('XMLDIR', 'InvalidElement.xml'),
         *
ERROR at line 2:
ORA-30937: No schema definition for 'UserName' (namespace '##local') in parent
'/PurchaseOrder'
```

Full Validation

Loading XML data into XML schema-based binary XML storage causes full validation against the target XML schemas. Otherwise, regardless of storage model, you can force full validation of XML instance documents against an XML schema at any time, using either of the following:

- Table level `CHECK` constraint
- PL/SQL `BEFORE INSERT` trigger

Both approaches ensure that only valid XML documents can be stored in the `XMLType` table.

The advantage of a `TABLE CHECK` constraint is that it is easy to code. The disadvantage is that it is based on Oracle SQL function `XMLisValid`, so it can only indicate whether or not the XML document is valid. If an XML document is invalid, a `TABLE CHECK` constraint cannot provide any information as to *why* it is invalid.

A `BEFORE INSERT` trigger requires slightly more code. The trigger validates the XML document by invoking `XMLType` method `schemaValidate()`. The advantage of using `schemaValidate()` is that the exception raised provides additional information about what was wrong with the instance document. Using a `BEFORE INSERT` trigger also makes it possible to attempt corrective action when an invalid document is encountered.

Full XML Schema Validation Costs Processing Time and Memory Usage Unless you are using binary XML storage, full XML schema validation costs processing time and memory. You should thus perform full XML schema validation only when necessary. If you can rely on your application to validate an XML document, you can obtain higher overall throughput with non-binary XML storage, by avoiding the overhead associated with full validation. If you cannot be sure about the validity of incoming XML documents, you can rely on the database to ensure that an `XMLType` table or column contains only schema-valid XML documents.

Example 3–18 shows how to force a full XML schema validation by adding a `CHECK` constraint to an `XMLType` table. In Example 3–18, the XML document `InvalidReference` is a not valid with respect to the XML schema. The XML schema defines a minimum length of 18 characters for the text node associated with the `Reference` element. In this document, the node contains the value `SBELL-20021009`, which is only 14 characters long. Partial validation would not catch this error. Unless the constraint or trigger is present, attempts to insert this document into the database would succeed.

Example 3–18 Forcing Full XML Schema Validation using a CHECK Constraint

```
ALTER TABLE purchaseorder
  ADD CONSTRAINT validate_purchaseorder
  CHECK (XMLIsValid(OBJECT_VALUE) = 1);

Table altered.
```

```
INSERT INTO purchaseorder
  VALUES (XMLType(bfilename('XMLDIR', 'InvalidReference.xml'),
                  nls_charset_id('AL32UTF8')));

INSERT INTO purchaseorder
*

ERROR at line 1:
ORA-02290: check constraint (QUINE.VALIDATE_PURCHASEORDER) violated
```

Pseudocolumn `OBJECT_VALUE` can be used to access the content of an `XMLType` table from within a trigger. Example 3–19 illustrates this, showing how to use a `BEFORE INSERT` trigger to validate that the data being inserted into the `XMLType` table conforms to the specified XML schema.

Example 3–19 Enforcing Full XML Schema Validation using a BEFORE INSERT Trigger

```
CREATE OR REPLACE TRIGGER validate_purchaseorder
   BEFORE INSERT ON purchaseorder
   FOR EACH ROW
BEGIN
  IF (:new.OBJECT_VALUE IS NOT NULL) THEN :new.OBJECT_VALUE.schemavalidate();
  END IF;
END;
/

INSERT INTO purchaseorder
  VALUES (XMLType(bfilename('XMLDIR', 'InvalidReference.xml'),
                  nls_charset_id('AL32UTF8')));
  VALUES (XMLType( bfilename('XMLDIR', 'InvalidReference.xml'),
          *
ERROR at line 2:
ORA-31154: invalid XML document
ORA-19202: Error occurred in XML processing
LSX-00221: "SBELL-20021009" is too short (minimum length is 18)
ORA-06512: at "SYS.XMLTYPE", line 354
ORA-06512: at "QUINE.VALIDATE_PURCHASEORDER", line 3
ORA-04088: error during execution of trigger 'QUINE.VALIDATE_PURCHASEORDER'
```

Enforcing Referential Integrity using SQL Constraints

The W3C XML Schema Recommendation defines a powerful language for defining the contents of an XML document. However, there are some simple data management concepts that are not currently addressed by the W3C XML Schema Recommendation. These include the ability to ensure that the value of an element or attribute has either of these properties:

- It is unique across a set of XML documents (a `UNIQUE` constraint).
- It exists in a particular data source that is outside of the current document (`FOREIGN KEY` constraint).

With Oracle XML DB, however, you can enforce such constraints. The mechanisms that you use to enforce integrity on XML data are the same mechanisms that you use to enforce integrity on relational data. Simple rules, such as uniqueness and foreign-key relationships, can be enforced by specifying constraints. More complex rules can be enforced by specifying database triggers.

Oracle XML DB lets you use the database to enforce business rules on XML content, in addition to enforcing rules that can be specified using XML Schema constructs. The

database enforces these business rules regardless of whether XML is inserted directly into a table or uploaded using one of the protocols supported by Oracle XML DB Repository.

Example 3–20, Example 3–21, and Example 3–22 illustrate how you can use SQL constraints to enforce referential integrity. Example 3–20 defines a uniqueness constraint on an `XMLType` table that is stored as binary XML. It defines a virtual column, using the `Reference` element in a purchase-order document. The uniqueness constraint `reference_is_unique` ensures that the value of node `/PurchaseOrder/Reference/text()` is unique across all documents that are stored in the table.

> **See Also:** "Partitioning or Constraining Binary XML Data using Virtual Columns" on page 3-3

Example 3–20 Constraining a Binary XML Table using a Virtual Column

```
CREATE TABLE po_binaryxml OF XMLType
  XMLTYPE STORE AS BINARY XML
  VIRTUAL COLUMNS
    (c_reference AS (XMLCast(XMLQuery('/PurchaseOrder/Reference'
                                      PASSING OBJECT_VALUE RETURNING CONTENT)
                             AS VARCHAR2(32))));

INSERT INTO po_binaryxml SELECT OBJECT_VALUE FROM OE.purchaseorder;

132 rows created.

ALTER TABLE po_binaryxml ADD CONSTRAINT reference_is_unique UNIQUE (c_reference);

INSERT INTO po_binaryxml
  VALUES (XMLType(bfilename('XMLDIR', 'DuplicateReference.xml'),
                  nls_charset_id('AL32UTF8')));
INSERT INTO po_binaryxml
*
ERROR at line 1:
ORA-00001: unique constraint (OE.REFERENCE_IS_UNIQUE) violated
```

Example 3–21 defines a uniqueness constraint similar to that of Example 3–20, but on `XMLType` table `purchaseorder` in standard database schema `OE`. In addition, it defines a foreign-key constraint that requires the `User` element of each purchase-order document to be the e-mail address of an employee that is in standard database table `HR.employees`. For XML data that is stored object-relationally, such as that in table `OE.purchaseorder`, constraints must be specified in terms of object attributes of the SQL data types that are used to manage the XML content.

Example 3–21 Integrity Constraints and Triggers for an XMLType Table Stored Object-Relationally

```
ALTER TABLE purchaseorder
  ADD CONSTRAINT reference_is_unique
  UNIQUE (XMLDATA."REFERENCE");

ALTER TABLE purchaseorder
  ADD CONSTRAINT user_is_valid
  FOREIGN KEY (XMLDATA."USERID") REFERENCES hr.employees(email);

INSERT INTO purchaseorder
  VALUES (XMLType(bfilename('XMLDIR', 'purchaseOrder.xml'),
                  nls_charset_id('AL32UTF8')));
```

```
INSERT INTO purchaseorder
  VALUES (XMLType(bfilename('XMLDIR', 'DuplicateReference.xml'),
                  nls_charset_id('AL32UTF8')));

INSERT INTO purchaseorder
*
ERROR at line 1:
ORA-00001: unique constraint (QUINE.REFERENCE_IS_UNIQUE) violated

INSERT INTO purchaseorder
  VALUES (XMLType(bfilename('XMLDIR', 'InvalidUser.xml'),
                  nls_charset_id('AL32UTF8')));

INSERT INTO purchaseorder
*
ERROR at line 1:
ORA-02291: integrity constraint (QUINE.USER_IS_VALID) violated - parent key not
 found
```

Just as for Example 3–20, the uniqueness constraint `reference_is_unique` of Example 3–21 ensures the uniqueness of the purchase-order `Reference` element across all documents stored in the table. The foreign key constraint `user_is_valid` here ensures that the value of element `User` corresponds to a value in column `email` of table `employees`.

The text node associated with the `Reference` element in the XML document `DuplicateRefernce.xml` contains the same value as the corresponding node in XML document `PurchaseOrder.xml`. Attempting to store both documents in Oracle XML DB thus violates the constraint `reference_is_unique`.

The text node associated with the `User` element in XML document `InvalidUser.xml` contains the value `HACKER`. There is no entry in the `employees` table where the value of column `email` is `HACKER`. Attempting to store this document in Oracle XML DB violates the constraint `user_is_valid`.

Integrity rules defined using constraints and triggers are also enforced when XML schema-based XML content is loaded into Oracle XML DB Repository. Example 3–22 illustrates this. It shows that database integrity is also enforced when a protocol, such as FTP, is used to upload XML schema-based XML content into Oracle XML DB Repository.

Example 3–22 Enforcing Database Integrity When Loading XML using FTP

```
$ ftp localhost 2100
Connected to localhost.
220 mdrake-sun FTP Server (Oracle XML DB/Oracle Database 10g Enterprise Edition
Release 10.1.0.0.0 - Beta) ready.
Name (localhost:oracle10): QUINE
331 Password required for QUINE
Password: password
230 QUINE logged in
ftp> cd /source/schemas
250 CWD Command successful
ftp> put InvalidReference.xml
200 PORT Command successful
150 ASCII Data Connection
550- Error Response
ORA-00604: error occurred at recursive SQL level 1
ORA-31154: invalid XML document
```

```
ORA-19202: Error occurred in XML processing
LSX-00221: "SBELL-20021009" is too short (minimum length is 18)
ORA-06512: at "SYS.XMLTYPE", line 333
ORA-06512: at "QUINE.VALIDATE_PURCHASEORDER", line 3
ORA-04088: error during execution of trigger 'QUINE.VALIDATE_PURCHASEORDER'
550 End Error Response
ftp> put InvalidElement.xml
200 PORT Command successful
150 ASCII Data Connection
550- Error Response
ORA-30937: No schema definition for 'UserName' (namespace '##local') in parent
'PurchaseOrder'
550 End Error Response
ftp> put DuplicateReference.xml
200 PORT Command successful
150 ASCII Data Connection
550- Error Response
ORA-00604: error occurred at recursive SQL level 1
ORA-00001: unique constraint (QUINE.REFERENCE_IS_UNIQUE) violated
550 End Error Response
ftp> put InvalidUser.xml
200 PORT Command successful
150 ASCII Data Connection
550- Error Response
ORA-00604: error occurred at recursive SQL level 1
ORA-02291: integrity constraint (QUINE.USER_IS_VALID) violated - parent key not
 found
550 End Error Response
```

When an error occurs while a document is being uploaded with a protocol, Oracle XML DB provides the client with the full SQL error trace. How the error is interpreted and reported to you is determined by the error-handling built into the client application. Some clients, such as the command line FTP tool, reports the error returned by Oracle XML DB, while others, such as Microsoft Windows Explorer, report a generic error message.

See also:

- "Specifying Relational Constraints on XMLType Tables and Columns" on page 7-34
- *Oracle Database Error Messages*

DML Operations on XML Content using Oracle XML DB

Another major advantage of using Oracle XML DB to manage XML content is that it leverages the power of Oracle Database to deliver powerful, flexible capabilities for querying and updating XML content, including the following:

- Retrieving nodes and fragments within an XML document
- Updating nodes and fragments within an XML document
- Creating indexes on specific nodes within an XML document
- Indexing the entire content of an XML document
- Determining whether an XML document contains a particular node

XPath and Oracle XML

Oracle XML DB includes `XMLType` methods and XML-specific SQL functions. With these, you can query and update XML content stored in Oracle Database. They use the W3C XPath Recommendation to identify the required node or nodes. Each node in an XML document can be uniquely identified by an XPath expression.

An XPath expression consists of a slash-separated list of element names, attributes names, and XPath functions. XPath expressions can contain positions and conditions that determine which branch of the tree is traversed in determining the target nodes.

By supporting XPath-based methods and functions, Oracle XML DB makes it possible for XML programmers to query and update XML documents in a familiar, standards-compliant manner.

Note: Oracle SQL functions and `XMLType` methods respect the W3C XPath recommendation, which states that if an XPath expression targets *no nodes* when applied to XML data, then an empty sequence must be returned. An error must *not* be raised in this case.

The specific semantics of an Oracle SQL function or `XMLType` method that applies an XPath expression to XML data determines what is returned. For example, SQL/XML function `XMLQuery` returns `NULL` if its XPath-expression argument targets no nodes, and the updating SQL functions, such as `deleteXML`, return the input XML data unchanged. An error is never raised if no nodes are targeted, but updating SQL functions can raise an error if an XPath-expression argument targets inappropriate nodes, such as attribute nodes or text nodes.

Querying XML Content Stored in Oracle XML DB

This section describes techniques for querying Oracle XML DB and retrieving XML content. This section contains these topics:

- PurchaseOrder XML Document
- Retrieving the Content of an XML Document using Pseudocolumn OBJECT_VALUE
- Accessing Fragments or Nodes of an XML Document using XMLQUERY
- Accessing Text Nodes and Attribute Values using XMLCAST and XMLQUERY
- Searching an XML Document using XMLEXISTS, XMLCast, and XMLQuery
- Performing SQL Operations on XMLType Fragments using XMLTABLE

PurchaseOrder XML Document

Examples in this section are based on the `PurchaseOrder` XML document shown in Example 3–23.

Example 3–23 PurchaseOrder XML Instance Document

```
<PurchaseOrder
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
    "http://localhost:8080/source/schemas/poSource/xsd/purchaseOrder.xsd">
  <Reference>SBELL-2002100912333601PDT</Reference>
```

```
  <Actions>
    <Action>
      <User>SVOLLMAN</User>
    </Action>
  </Actions>
  <Reject/>
  <Requestor>Sarah J. Bell</Requestor>
  <User>SBELL</User>
  <CostCenter>S30</CostCenter>
  <ShippingInstructions>
    <name>Sarah J. Bell</name>
    <address>400 Oracle Parkway
      Redwood Shores
      CA
      94065
      USA</address>
    <telephone>650 506 7400</telephone>
  </ShippingInstructions>
  <SpecialInstructions>Air Mail</SpecialInstructions>
  <LineItems>
    <LineItem ItemNumber="1">
      <Description>A Night to Remember</Description>
      <Part Id="715515009058" UnitPrice="39.95" Quantity="2"/>
    </LineItem>
    <LineItem ItemNumber="2">
      <Description>The Unbearable Lightness Of Being</Description>
      <Part Id="37429140222" UnitPrice="29.95" Quantity="2"/>
    </LineItem>
    <LineItem ItemNumber="3">
      <Description>Sisters</Description>
      <Part Id="715515011020" UnitPrice="29.95" Quantity="4"/>
    </LineItem>
  </LineItems>
</PurchaseOrder>
```

Retrieving the Content of an XML Document using Pseudocolumn OBJECT_VALUE

Pseudocolumn `OBJECT_VALUE` can be used as an alias for the value of an object table. For an `XMLType` table that consists of a single column of `XMLType`, the entire XML document is retrieved. (`OBJECT_VALUE` replaces the `value(x)` and `SYS_NC_ROWINFO$` aliases used in releases prior to Oracle Database10g Release 1.)

In Example 3–24, the SQL*Plus settings `PAGESIZE` and `LONG` are used to ensure that the entire document is printed correctly, without line breaks. (The output has been formatted for readability.)

Example 3–24 Retrieving an Entire XML Document using OBJECT_VALUE

```
SET LONG 10000
SET PAGESIZE 100

SELECT OBJECT_VALUE FROM purchaseorder;

OBJECT_VALUE
----------------------------------------------------------------------
<PurchaseOrder xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:noNamespaceSchemaLocation="http://localhost:8080/source/schemas
/poSource/xsd/purchaseOrder.xsd">
  <Reference>SBELL-20021009123336 01PDT</Reference>
  <Actions>
```

```
    <Action>
      <User>SVOLLMAN</User>
    </Action>
  </Actions>
  <Reject/>
  <Requestor>Sarah J. Bell</Requestor>
  <User>SBELL</User>
  <CostCenter>S30</CostCenter>
  <ShippingInstructions>
    <name>Sarah J. Bell</name>
    <address>400 Oracle Parkway
Redwood Shores
CA
94065
USA</address>
    <telephone>650 506 7400</telephone>
  </ShippingInstructions>
  <SpecialInstructions>Air Mail</SpecialInstructions>
  <LineItems>
    <LineItem ItemNumber="1">
      <Description>A Night to Remember</Description>
      <Part Id="715515009058" UnitPrice="39.95" Quantity="2"/>
    </LineItem>
    <LineItem ItemNumber="2">
      <Description>The Unbearable Lightness Of Being</Description>
      <Part Id="37429140222" UnitPrice="29.95" Quantity="2"/>
    </LineItem>
    <LineItem ItemNumber="3">
      <Description>Sisters</Description>
      <Part Id="715515011020" UnitPrice="29.95" Quantity="4"/>
    </LineItem>
  </LineItems>
</PurchaseOrder>

1 row selected.
```

Accessing Fragments or Nodes of an XML Document using XMLQUERY

You can use SQL/XML function `XMLQuery` to extract the nodes that match an XPath expression. The result is returned as an instance of `XMLType`. Example 3–25 illustrates this with several queries.

Example 3–25 Accessing XML Fragments using XMLQUERY

The following query returns an `XMLType` instance containing the `Reference` element that matches the XPath expression.

```
SELECT XMLQuery('/PurchaseOrder/Reference' PASSING OBJECT_VALUE RETURNING CONTENT)
  FROM purchaseorder;

XMLQUERY('/PURCHASEORDER/REFERENCE'PASSINGOBJECT_
-------------------------------------------------
<Reference>SBELL-2002100912333601PDT</Reference>

1 row selected.
```

The following query returns an `XMLType` instance containing the first `LineItem` element in the `LineItems` collection:

```
SELECT XMLQuery('/PurchaseOrder/LineItems/LineItem[1]'
```

```
                PASSING OBJECT_VALUE RETURNING CONTENT)
  FROM purchaseorder;

XMLQUERY('/PURCHASEORDER/LINEITEMS/LINEITEM[1]'PASSINGOBJECT_
------------------------------------------------------------
<LineItem ItemNumber="1">
  <Description>A Night to Remember</Description>
  <Part Id="715515009058" UnitPrice="39.95" Quantity="2"/>
</LineItem>

1 row selected.
```

The following query returns an `XMLType` instance that contains the three `Description` elements that match the XPath expression. These elements are returned as nodes in a single `XMLType` instance. The `XMLType` instance does not have a single root node; it is an XML *fragment*.

```
SELECT XMLQuery('/PurchaseOrder/LineItems/LineItem/Description'
                PASSING OBJECT_VALUE RETURNING CONTENT)
  FROM purchaseorder;

XMLQUERY('/PURCHASEORDER/LINEITEMS/LINEITEM/DESCRIPTION'PASSINGOBJECT_
----------------------------------------------------------------------
<Description>A Night to Remember</Description>
<Description>The Unbearable Lightness Of Being</Description>
<Description>Sisters</Description>

1 row selected.
```

See Also: "Performing SQL Operations on XMLType Fragments using XMLTABLE" on page 3-46

Accessing Text Nodes and Attribute Values using XMLCAST and XMLQUERY

You can access text node and attribute values using SQL/XML standard functions `XMLQuery` and `XMLCast`. To do this, the XPath expression passed to `XMLQuery` must uniquely identify a *single* text node or attribute value within the document – that is, a *leaf* node. Example 3–26 illustrates this using several queries.

Example 3–26 Accessing a Text Node Value using XMLCAST and XMLQuery

The following query returns the value of the text node associated with the `Reference` element that matches the target XPath expression. The value is returned as a `VARCHAR2` value.

```
SELECT  XMLCast(XMLQuery('$p/PurchaseOrder/Reference/text()'
                         PASSING OBJECT_VALUE AS "p" RETURNING CONTENT)
                AS VARCHAR2(30))
  FROM purchaseorder;

XMLCAST(XMLQUERY('$P/PURCHASEO
------------------------------
SBELL-2002100912333601PDT

1 row selected.
```

The following query returns the value of the text node associated with a `Description` element contained in a `LineItem` element. The particular `LineItem` element is specified by its `Id` attribute value. The predicate that identifies the `LineItem` element is `[Part/@Id="715515011020"]`. The at-sign character (`@`)

specifies that `Id` is an attribute rather than an element. The value is returned as a `VARCHAR2` value.

```
SELECT XMLCast(
         XMLQuery('$p/PurchaseOrder/LineItems/LineItem[Part/@Id="715515011020"]/Description/text()'
                  PASSING OBJECT_VALUE AS "p" RETURNING CONTENT)
               AS VARCHAR2(30))
  FROM purchaseorder;

XMLCAST(XMLQUERY('$P/PURCHASEO
------------------------------
Sisters

1 row selected.
```

The following query returns the value of the text node associated with the `Description` element contained in the first `LineItem` element. The first `LineItem` element is indicated by the position predicate `[1]`.

```
SELECT XMLCast(XMLQuery('$p/PurchaseOrder/LineItems/LineItem[1]/Description'
                        PASSING OBJECT_VALUE AS "p" RETURNING CONTENT)
               AS VARCHAR2(4000))
  FROM purchaseorder;

XMLCAST(XMLQUERY('$P/PURCHASEORDER/LINEITEMS/LINEITEM[1]/DESCRIPTION'PASSINGOBJECT_VALUEAS"P"
----------------------------------------------------------------------------------------------
A Night to Remember

1 row selected.
```

See Also:

- "Querying XMLType Data using SQL/XML Functions XMLExists and XMLCast" on page 4-2 for information on SQL/XML function `XMLCast`
- Chapter 5, "Using XQuery with Oracle XML DB" for information on SQL/XML function `XMLQuery`

Searching an XML Document using XMLEXISTS, XMLCast, and XMLQuery

SQL/XML standard function `XMLExists` evaluates whether or not a given document contains a node that matches a W3C XPath expression. Function `XMLExists` returns a Boolean value of `true` if the document contains the node specified by the XPath expression supplied to the function and a value of `false` if it does not. Since XPath expressions can contain predicates, `XMLExists` can determine whether or not a given node exists in the document, and whether or not a node with the specified value exists in the document.

Similarly, you can use SQL/XML functions `XMLCast` and `XMLQuery` in a SQL `WHERE` clause to limit the query results to documents that satisfy some property. Example 3–27 illustrates the use of `XMLExists`, `XMLCast`, and `XMLQuery` to search for documents.

Example 3–27 Searching XML Content using XMLExists, XMLCast, and XMLQuery

The following query uses `XMLExists` to check if the XML document contains an element named `Reference` that is a child of the root element `PurchaseOrder`:

```
SELECT count(*) FROM purchaseorder
  WHERE XMLExists('$p/PurchaseOrder/Reference' PASSING OBJECT_VALUE AS "p");
```

```
  COUNT(*)
----------
       132

1 row selected.
```

The following query checks if the value of the text node associated with the `Reference` element is `SBELL-2002100912333601PDT`:

```
SELECT count(*) FROM purchaseorder
  WHERE XMLExists('$p/PurchaseOrder[Reference="SBELL-2002100912333601PDT"]'
                  PASSING OBJECT_VALUE AS "p");

  COUNT(*)
----------
         1
1 row selected.
```

This query checks whether the value of the text node associated with the `Reference` element is `SBELL-XXXXXXXXXXXXXXXXXX`:

```
SELECT count(*) FROM purchaseorder
  WHERE XMLExists('$p/PurchaseOrder[Reference="SBELL-XXXXXXXXXXXXXXXXXX"]'
                  PASSING OBJECT_VALUE AS "p");

  COUNT(*)
----------
         0

1 row selected.
```

This query checks whether the XML document contains a root element `PurchaseOrder` that contains a `LineItems` element that contains a `LineItem` element that contains a `Part` element with an `Id` attribute.

```
SELECT count(*) FROM purchaseorder
  WHERE XMLExists('$p/PurchaseOrder/LineItems/LineItem/Part/@Id'
                  PASSING OBJECT_VALUE AS "p");

  COUNT(*)
----------
       132

1 row selected.
```

The following query checks whether the XML document contains a root element `PurchaseOrder` that contains a `LineItems` element that contains a `LineItem` element that contains a `Part` element with `Id` attribute value `715515009058`.

```
SELECT count(*) FROM purchaseorder
  WHERE XMLExists('$p/PurchaseOrder/LineItems/LineItem/Part[@Id="715515009058"]'
                  PASSING OBJECT_VALUE AS "p");

  COUNT(*)
----------
        21
```

The following query checks whether the XML document contains a root element `PurchaseOrder` that contains a `LineItems` element whose *third* `LineItem` element contains a `Part` element with `Id` attribute value `715515009058`.

```
SELECT count(*) FROM purchaseorder
  WHERE XMLExists(
          '$p/PurchaseOrder/LineItems/LineItem[3]/Part[@Id="715515009058"]'
          PASSING OBJECT_VALUE AS "p");

  COUNT(*)
----------
         1
1 row selected.
```

The following query limits the results of the `SELECT` statement to rows where the text node associated with element `User` starts with the letter `S`. XQuery does not include support for `LIKE`-based queries.

```
SELECT XMLCast(XMLQuery('$p/PurchaseOrder/Reference' PASSING OBJECT_VALUE AS "p"
                        RETURNING CONTENT)
               AS VARCHAR2(30))
  FROM purchaseorder
  WHERE XMLCast(XMLQuery('$p/PurchaseOrder/User' PASSING OBJECT_VALUE AS "p"
                         RETURNING CONTENT)
                AS VARCHAR2(30))
        LIKE 'S%';

XMLCAST(XMLQUERY('$P/PURCHASEORDER
----------------------------------
SBELL-20021009123336231PDT
SBELL-20021009123336331PDT
SKING-20021009123336321PDT
...
36 rows selected.
```

The following query performs a join based on the values of a node in an XML document and data in another table.

```
SELECT XMLCast(XMLQuery('$p/PurchaseOrder/Reference' PASSING OBJECT_VALUE AS "p"
                        RETURNING CONTENT)
               AS VARCHAR2(30))
  FROM purchaseorder p, hr.employees e
  WHERE XMLCast(XMLQuery('$p/PurchaseOrder/User' PASSING OBJECT_VALUE AS "p"
                         RETURNING CONTENT)
                AS VARCHAR2(30)) = e.email
    AND e.employee_id = 100;

XMLCAST(XMLQUERY('$P/PURCHASEOREDER
-----------------------------------
SKING-20021009123336321PDT
SKING-20021009123337153PDT
SKING-20021009123335560PDT
SKING-20021009123336952PDT
SKING-20021009123336622PDT
SKING-20021009123336822PDT
SKING-20021009123336131PDT
SKING-20021009123336392PDT
SKING-20021009123337974PDT
SKING-20021009123338294PDT
SKING-20021009123337703PDT
SKING-20021009123337383PDT
SKING-20021009123337503PDT

13 rows selected.
```

The following query uses `XMLExists` to limit the results of a `SELECT` statement to rows where the text node of element `User` contains the value `SBELL`.

```
SELECT XMLCast(XMLQuery('$p/PurchaseOrder/Reference' PASSING OBJECT_VALUE AS "p"
                        RETURNING CONTENT)
               AS VARCHAR2(30)) "Reference"
  FROM purchaseorder
  WHERE XMLExists('$p/PurchaseOrder[User="SBELL"]' PASSING OBJECT_VALUE AS "p");

Reference
------------------------------
SBELL-20021009123336231PDT
SBELL-20021009123336331PDT
SBELL-20021009123337353PDT
SBELL-20021009123338304PDT
SBELL-20021009123338505PDT
SBELL-20021009123335771PDT
SBELL-20021009123335280PDT
SBELL-2002100912333763PDT
SBELL-2002100912333601PDT
SBELL-20021009123336362PDT
SBELL-20021009123336532PDT
SBELL-20021009123338204PDT
SBELL-20021009123337673PDT

13 rows selected.
```

Example 3–28 uses SQL/XML functions `XMLQuery` and `XMLExists` to find the `Reference` element for any `PurchaseOrder` element whose first `LineItem` element contains an order for the item with `Id` `715515009058`. Function `XMLExists` is used in the `WHERE` clause to determine which rows are selected, and `XMLQuery` is used in the `SELECT` list to control which part of the selected documents appears in the result.

Example 3–28 Finding the Reference for a Purchase Order using XMLQuery and XMLExists

```
SELECT XMLCast(XMLQuery('$p/PurchaseOrder/Reference' PASSING OBJECT_VALUE AS "p"
                        RETURNING CONTENT)
               AS VARCHAR2(30)) "Reference"
  FROM purchaseorder
  WHERE XMLExists('$p/PurchaseOrder/LineItems/LineItem[1]/Part[@Id="715515009058"]'
                  PASSING OBJECT_VALUE AS "p");

Reference
-------------------------
SBELL-2002100912333601PDT

1 row selected.
```

See Also:

- "Querying XMLType Data using SQL/XML Functions XMLExists and XMLCast" on page 4-2 for information on SQL/XML functions `XMLCast` and `XMLExists`
- Chapter 5, "Using XQuery with Oracle XML DB" for information on SQL/XML function `XMLQuery`

Performing SQL Operations on XMLType Fragments using XMLTABLE

Example 3–25 demonstrates how to extract an `XMLType` instance that contains the node or nodes that match an XPath expression. When the document contains *multiple* nodes that match the supplied XPath expression, such a query returns an XML *fragment* that contains all of the matching nodes. Unlike an XML document, an XML **fragment** has no single element that is the *root* element.

This kind of result is common in these cases:

- When you retrieve the set of elements contained in a *collection*, in which case all nodes in the fragment are of the same type – see Example 3–29
- When the target XPath expression ends in a *wildcard*, in which case the nodes in the fragment can be of different types – see Example 3–31

You can use SQL/XML function `XMLTable` to break up an XML fragment contained in an `XMLType` instance, inserting the collection-element data into a new, virtual table, which you can then query using SQL—in a join expression, for example. In particular, converting an XML fragment into a virtual table makes it easier to process the result of evaluating an `XMLQuery` expression that returns multiple nodes.

Example 3–29 shows how to access the text nodes for each `Description` element in the `PurchaseOrder` document. It breaks up the single XML Fragment output from Example 3–25 into multiple text nodes.

Example 3–29 Accessing Description Nodes using XMLTABLE

```
SELECT des.COLUMN_VALUE
  FROM purchaseorder p,
       XMLTable('/PurchaseOrder/LineItems/LineItem/Description'
                PASSING p.OBJECT_VALUE) des
  WHERE XMLExists('$p/PurchaseOrder[Reference="SBELL-2002100912333601PDT"]'
                  PASSING OBJECT_VALUE AS "p");

COLUMN_VALUE
------------
<Description>A Night to Remember</Description>
<Description>The Unbearable Lightness Of Being</Description>
<Description>Sisters</Description>

3 rows selected.
```

To use SQL to process the contents of the text nodes, Example 3–29 converts the collection of `Description` nodes into a *virtual table*, using SQL/XML function `XMLTable`. The virtual table has three rows, each of which contains a single `XMLType` instance with a single `Description` element.

The XPath expression targets the `Description` elements. The `PASSING` clause says to use the contents (`OBJECT_VALUE`) of `XMLType` table `purchaseorder` as the context for evaluating the XPath expression.

The `XMLTable` expression thus *depends* on the `purchaseorder` table. This is a *left lateral join*. This correlated join ensures a one-to-many (1:N) relationship between the `purchaseorder` row accessed and the rows generated from it by `XMLTable`. Because of this correlated join, the `purchaseorder` table *must appear before* the `XMLTable` expression in the `FROM` list. This is a general requirement in any situation where the `PASSING` clause refers to a column of the table.

Each `XMLType` instance in the virtual table contains a single `Description` element. You can use the `COLUMNS` clause of `XMLTable` to break up the data targeted by the

XPath expression `'Description'` into a column named `description` of SQL data type `VARCHAR2(256)`. The `'Description'` expression that defines this column is *relative* to the *context* XPath expression, `'/PurchaseOrder/LineItems/LineItem'`.

```
SELECT des.description
  FROM purchaseorder p,
       XMLTable('/PurchaseOrder/LineItems/LineItem' PASSING p.OBJECT_VALUE
                COLUMNS description VARCHAR2(256) PATH 'Description') des
  WHERE XMLExists('$p/PurchaseOrder[Reference="SBELL-2002100912333601PDT"]'
                  PASSING OBJECT_VALUE AS "p");

DESCRIPTION
--------------------------------
A Night to Remember
The Unbearable Lightness Of Being
Sisters

3 rows selected.
```

The `COLUMNS` clause lets you specify precise SQL data types, which can make static type-checking more helpful. This example uses only a single column (`description`). To expose data that is contained at multiple levels in an `XMLType` table as individual rows in a relational view, apply `XMLTable` to each document level to be broken up and stored in relational columns. See Example 3–33 for an example.

Example 3–30 counts the number of elements in a collection. It also shows how SQL keywords such as `ORDER BY` and `GROUP BY` can be applied to the virtual table data created by SQL/XML function `XMLTable`.

Example 3–30 Counting the Number of Elements in a Collection using XMLTABLE

```
SELECT reference, count(*)
  FROM purchaseorder,
       XMLTable('/PurchaseOrder' PASSING OBJECT_VALUE
                COLUMNS reference VARCHAR2(32) PATH 'Reference',
                        lineitem  XMLType      PATH 'LineItems/LineItem'),
       XMLTable('LineItem' PASSING lineitem)
  WHERE XMLExists('$p/PurchaseOrder[User="SBELL"]'
                  PASSING OBJECT_VALUE AS "p")
  GROUP BY reference
  ORDER BY reference;

REFERENCE                    COUNT(*)
--------------------------   --------
SBELL-20021009123335280PDT         20
SBELL-20021009123335771PDT         21
SBELL-2002100912333601PDT           3
SBELL-20021009123336231PDT         25
SBELL-20021009123336331PDT         10
SBELL-20021009123336362PDT         15
SBELL-20021009123336532PDT         14
SBELL-20021009123337353PDT         10
SBELL-2002100912333763PDT          21
SBELL-20021009123337673PDT         10
SBELL-20021009123338204PDT         14
SBELL-20021009123338304PDT         24
SBELL-20021009123338505PDT         20

13 rows selected.
```

The query in Example 3–30 locates the set of XML documents that match the XPath expression to SQL/XML function `XMLExists`. It generates a virtual table with two columns:

- `reference`, containing the `Reference` node for each document selected
- `lineitem`, containing the set of `LineItem` nodes for each document selected

It counts the number of `LineItem` nodes for each document. A correlated join ensures that the `GROUP BY` correctly determines which `LineItem` elements belong to which `PurchaseOrder` element.

Example 3–31 shows how to use SQL/XML function `XMLTable` to count the number of *child* elements of a given element. The XPath expression passed to `XMLTable` contains a wildcard (`*`) that matches all elements that are direct descendants of a `PurchaseOrder` element. Each row of the virtual table created by `XMLTable` contains a node that matches the XPath expression. Counting the number of rows in the virtual table provides the number of element children of element `PurchaseOrder`.

Example 3–31 Counting the Number of Child Elements in an Element using XMLTABLE

```
SELECT count(*)
  FROM purchaseorder p, XMLTable('/PurchaseOrder/*' PASSING p.OBJECT_VALUE)
  WHERE XMLExists('$p/PurchaseOrder[Reference="SBELL-2002100912333601PDT"]'
                 PASSING OBJECT_VALUE AS "p");

  COUNT(*)
----------
         9

1 row selected.
```

Accessing XML Data in Oracle XML DB using Relational Views

You can use the XML-specific functions and methods provided by Oracle XML DB to create conventional *relational views* that provide relational access to XML content. This lets programmers, tools, and applications that understand Oracle Database, but not XML, to work with XML content stored in the database.

The relational views can use XPath expressions and SQL/XML query and access functions such as `XMLTable` to define a mapping between columns in the view and nodes in the XML document. For performance reasons, this approach is recommended only when XML documents are stored using structured (object-relational) or binary XML storage, not when stored as `CLOB` instances.

See Also:

- Chapter 4, "XMLType Operations" for a description of `XMLType` data type and functions
- `http://www.w3.org/TR/xpath` for information about XPath 1.0
- `http://www.w3.org/TR/xpath20/` for information about XPath 2.0
- `http://www.w3.org/TR/2002/NOTE-unicode-xml-20020218/` for information about using Unicode in XML

Breaking Up a Single Level of XML Data

When you need to expose each document in an XMLType table as a row in a relational view, you can use this technique:

1. Define the set of columns that make up the view, using CREATE OR REPLACE VIEW.
2. Map the nodes in the XML document to the columns defined by the view. You do this by extracting the nodes using SQL/XML function XMLTable with appropriate XPath expressions.

This technique can be used whenever there is a one-to-*one* (1:1) relationship between documents in the XMLType table and the rows in the view.

Example 3–32 shows how to create a simple relational view, purchaseorder_master_view, that exposes XML content. There is one row in the view for each row in XMLType table purchaseorder.

Example 3–32 Creating a Relational View of XML Content

```
CREATE OR REPLACE VIEW purchaseorder_master_view AS
  SELECT po.*
    FROM purchaseorder pur,
         XMLTable(
           '$p/PurchaseOrder' PASSING pur.OBJECT_VALUE as "p"
           COLUMNS
             reference       VARCHAR2(30)   PATH 'Reference',
             requestor       VARCHAR2(128)  PATH 'Requestor',
             userid          VARCHAR2(10)   PATH 'User',
             costcenter      VARCHAR2(4)    PATH 'CostCenter',
             ship_to_name    VARCHAR2(20)   PATH 'ShippingInstructions/name',
             ship_to_address VARCHAR2(256)  PATH 'ShippingInstructions/address',
             ship_to_phone   VARCHAR2(24)   PATH 'ShippingInstructions/telephone',
             instructions    VARCHAR2(2048) PATH 'SpecialInstructions') po;

View created.

DESCRIBE purchaseorder_master_view

Name            Null?    Type
-----------------------------------------
REFERENCE                VARCHAR2(30)
REQUESTOR                VARCHAR2(128)
USERID                   VARCHAR2(10)
COSTCENTER               VARCHAR2(4)
SHIP_TO_NAME             VARCHAR2(20)
SHIP_TO_ADDRESS          VARCHAR2(256)
SHIP_TO_PHONE            VARCHAR2(24)
INSTRUCTIONS             VARCHAR2(2048)
```

Breaking Up Multiple Levels of XML Data

When you need to expose data contained at multiple levels in an XMLType table as individual rows in a relational view, you use the same general approach as for breaking up a single level: 1) define the columns making up the view, and 2) map the XML nodes to the columns. However, in this case you apply XMLTable, to each document level that is to be broken up and stored in relational columns.

This technique can be used whenever there is a one-to-*many* (1:N) relationship between documents in the `XMLType` table and the rows in the view.

For example, each `PurchaseOrder` element contains a `LineItems` element, which in turn contains one or more `LineItem` elements. Each `LineItem` element has child elements, such as `Description`, and an `ItemNumber` attribute. To make such lower-level data accessible as a relational value, you must break up both the `PurchaseOrder` element and the `LineItem` collection. Each such decomposition is done with `XMLTable`. When element `PurchaseOrder` is broken up, the `LineItem` element is mapped to a relational column of type `XMLType`, which contains an XML fragment. That column is then passed to the second call to `XMLType`, to be broken into its various parts as multiple rows of relational values.

Example 3–33 illustrates this. It shows how to use SQL/XML function `XMLTable` for a one-to-*many* (1:N) relationship between the documents in `XMLType` table `purchaseorder` and the view rows. The view provides access to the individual members of a *collection*, and exposes the collection members as a set of rows.

Example 3–33 Accessing Individual Members of a Collection using a View

```
CREATE OR REPLACE VIEW purchaseorder_detail_view AS
  SELECT po.reference, li.*
    FROM purchaseorder p,
         XMLTable('/PurchaseOrder' PASSING p.OBJECT_VALUE
                  COLUMNS
                    reference VARCHAR2(30) PATH 'Reference',
                    lineitem  XMLType      PATH 'LineItems/LineItem') po,
         XMLTable('/LineItem' PASSING po.lineitem
                  COLUMNS
                    itemno      NUMBER(38)    PATH '@ItemNumber',
                    description VARCHAR2(256) PATH 'Description',
                    partno      VARCHAR2(14)  PATH 'Part/@Id',
                    quantity    NUMBER(12, 2) PATH 'Part/@Quantity',
                    unitprice   NUMBER(8, 4)  PATH 'Part/@UnitPrice') li;

View created.

DESCRIBE purchaseorder_detail_view
Name           Null?    Type
----------------------------
REFERENCE               VARCHAR2(30)
ITEMNO                  NUMBER(38)
DESCRIPTION             VARCHAR2(256)
PARTNO                  VARCHAR2(14)
QUANTITY                NUMBER(12,2)
UNITPRICE               NUMBER(8,4)
```

In Example 3–33, there is one row in view `purchaseorder_detail_view` for each `LineItem` element in the XML documents stored in `XMLType` table `purchaseorder`.

The `CREATE OR REPLACE VIEW` statement defines the set of columns that make up the view. The `SELECT` statement passes the `purchaseorder` table as context to function `XMLTable`, to create the virtual table `p`, which has columns `reference` and `lineitem`. These columns contain the `Reference` and `LineItem` elements of the purchase-order documents, respectively.

Column `lineitem` contains a collection of `LineItem` elements, as an `XMLType` instance—one row for each `LineItem` element. These rows are in turn passed to a second `XMLTable` expression, to serve as its context. This second `XMLTable` expression creates a virtual table of line-item rows, with columns corresponding to

various descendant nodes of element `LineItem`. Most of these descendants are attributes (`ItemNumber`, `Part/@Id`, and so on). One of the descendants is the `Description` child element.

The `Reference` element is included in view `purchaseorder_detail_view` as column `reference`. It provides a foreign key that can be used to joins rows in view `purchaseorder_detail_view` to the corresponding row in view `purchaseorder_master_view`. The correlated join in the `CREATE VIEW` statement ensures that the one-to-many (1:N) relationship between the `Reference` element and the associated `LineItem` elements is maintained whenever the view is accessed.

Querying XML Content As Relational Data

The examples in this section show relational queries of XML data. They point out some of the benefits provided by creating relational views over `XMLType` tables and columns.

Example 3–34 shows how to query master and detail relational views of XML data.

Example 3–34 Querying XML Data using Views

The following simple query against a master view uses a conventional `SELECT` statement to return the rows where the `userid` column starts with `S`.

```
SELECT reference, costcenter, ship_to_name
  FROM purchaseorder_master_view
  WHERE userid LIKE 'S%';

REFERENCE                      COST SHIP_TO_NAME
------------------------------ ---- --------------
SBELL-20021009123336231PDT     S30  Sarah J. Bell
SBELL-20021009123336331PDT     S30  Sarah J. Bell
SKING-20021009123336321PDT     A10  Steven A. King
...
36 rows selected.
```

The following query is based on a join between the master view and the detail view. A conventional `SELECT` statement finds the `purchaseorder_detail_view` rows where the value of column `itemno` is `1` and the corresponding `purchaseorder_master_view` row contains a `userid` column with the value `SBELL`.

```
SELECT d.reference, d.itemno, d.partno, d.description
  FROM purchaseorder_detail_view d, purchaseorder_master_view m
  WHERE m.reference = d.reference
    AND m.userid = 'SBELL'
    AND d.itemno = 1;

REFERENCE                          ITEMNO PARTNO         DESCRIPTION
------------------------------ ----------------------------------------------
SBELL-20021009123336231PDT              1 37429165829    Juliet of the Spirits
SBELL-20021009123336331PDT              1 715515009225   Salo
SBELL-20021009123337353PDT              1 37429141625    The Third Man
SBELL-20021009123338304PDT              1 715515009829   Nanook of the North
SBELL-20021009123338505PDT              1 37429122228    The 400 Blows
SBELL-20021009123335771PDT              1 37429139028    And the Ship Sails on
SBELL-20021009123335280PDT              1 715515011426   All That Heaven Allows
SBELL-2002100912333763PDT               1 715515010320   Life of Brian - Python
SBELL-2002100912333601PDT               1 715515009058   A Night to Remember
SBELL-20021009123336362PDT              1 715515012928   In the Mood for Love
SBELL-20021009123336532PDT              1 37429162422    Wild Strawberries
```

```
SBELL-20021009123338204PDT              1 37429168820    Red Beard
SBELL-20021009123337673PDT              1 37429156322    Cries and Whispers

13 rows selected.
```

The views in Example 3–34 look and act like standard relational views. They can be queried using standard relational syntax. No XML-specific syntax is required in either the query or the generated result set.

By exposing XML content as relational data, Oracle XML DB lets you apply advanced database features, such as business intelligence and analytic capabilities, to XML content, even if such features themselves are not XML-aware.

Example 3–35 shows how to use relational views over XML content to perform business-intelligence queries on XML documents. The example query selects `PurchaseOrder` documents that contain orders for titles identified by UPC codes `715515009058` and `715515009126`.

Example 3–35 Business-Intelligence Query of XML Data using a View

```
SELECT partno, count(*) "No of Orders", quantity "No of Copies"
  FROM purchaseorder_detail_view
  WHERE partno IN (715515009126, 715515009058)
  GROUP BY rollup(partno, quantity);

PARTNO         No of Orders No of Copies
-------------- ------------ ------------
715515009058              7            1
715515009058              9            2
715515009058              5            3
715515009058              2            4
715515009058             23
715515009126              4            1
715515009126              7            3
715515009126             11
                         34
9 rows selected.
```

The query in Example 3–35 determines the number of copies of each title that are ordered in each `PurchaseOrder` document. For part number `715515009126`, there are four `PurchaseOrder` documents where one copy of the item is ordered and seven `PurchaseOrder` documents where three copies of the item are ordered.

Updating XML Content Stored in Oracle XML DB

Oracle XML DB lets update operations take place on XML content. Update operations can either replace the entire contents of a document or parts of a document. The ability to perform partial updates on XML documents is very powerful, particularly when you make small changes to large documents, as it can significantly reduce the amount of network traffic and disk input-output required to perform the update.

SQL function `updateXML` enables partial update of an XML document stored as an `XMLType` instance. It lets multiple changes be made to the document in a single operation. Each change consists of an XPath expression that identifies a node to be updated, and the new value for the node.

Example 3–36 uses SQL function `updateXML` to update the text node associated with element `User`.

Example 3–36 Updating XML Content using UPDATEXML

```
SELECT XMLCast(XMLQuery('$p/PurchaseOrder/User' PASSING OBJECT_VALUE AS "p"
                                                RETURNING CONTENT)
               AS VARCHAR2(60))
  FROM purchaseorder
  WHERE XMLExists('$p/PurchaseOrder[Reference="SBELL-2002100912333601PDT"]'
                  PASSING OBJECT_VALUE AS "p");

XMLCAST(XMLQUERY('$P/PURCHAS
----------------------------
SBELL

1 row selected.

UPDATE purchaseorder
SET OBJECT_VALUE =
    updateXML(OBJECT_VALUE, '/PurchaseOrder/User/text()', 'SKING')
  WHERE XMLExists('$p/PurchaseOrder[Reference="SBELL-2002100912333601PDT"]'
                  PASSING OBJECT_VALUE AS "p");

1 row updated.

SELECT XMLCast(XMLQuery('$p/PurchaseOrder/User' PASSING OBJECT_VALUE AS "p"
                                                RETURNING CONTENT)
               AS VARCHAR2(60))
  FROM purchaseorder
  WHERE XMLExists('$p/PurchaseOrder[Reference="SBELL-2002100912333601PDT"]'
                  PASSING OBJECT_VALUE AS "p");

XMLCAST(XMLQUERY('$P/PURCHAS
----------------------------
SKING

1 row selected.
```

Example 3–37 uses SQL function `updateXML` to replace an entire element within an XML document. The XPath expression references the element, and the replacement value is passed as an `XMLType` object.

Example 3–37 Replacing an Entire Element using UPDATEXML

```
SELECT XMLQuery('$p/PurchaseOrder/LineItems/LineItem[1]'
                PASSING OBJECT_VALUE AS "p" RETURNING CONTENT)
  FROM purchaseorder
  WHERE XMLExists('$p/PurchaseOrder[Reference="SBELL-2002100912333601PDT"]'
                  PASSING OBJECT_VALUE AS "p");

XMLQUERY('$P/PURCHAS
--------------------
<LineItem ItemNumber="1">
  <Description>A Night to Remember</Description>
  <Part Id="715515009058" UnitPrice="39.95" Quantity="2"/>
</LineItem>

1 row selected.

UPDATE purchaseorder
  SET OBJECT_VALUE =
        updateXML(
          OBJECT_VALUE,
```

```
          '/PurchaseOrder/LineItems/LineItem[1]',
          XMLType('<LineItem ItemNumber="1">
                     <Description>The Lady Vanishes</Description>
                     <Part Id="37429122129" UnitPrice="39.95" Quantity="1"/>
                   </LineItem>'))
    WHERE XMLExists('$p/PurchaseOrder[Reference="SBELL-2002100912333601PDT"]'
                    PASSING OBJECT_VALUE AS "p");

1 row updated.

SELECT XMLQuery('$p/PurchaseOrder/LineItems/LineItem[1]'
                PASSING OBJECT_VALUE AS "p" RETURNING CONTENT)
  FROM purchaseorder
  WHERE XMLExists('$p/PurchaseOrder[Reference="SBELL-2002100912333601PDT"]'
                  PASSING OBJECT_VALUE AS "p");

XMLQUERY('$P/PURCHAS
--------------------
<LineItem ItemNumber="1">
  <Description>The Lady Vanishes</Description>
  <Part Id="37429122129" UnitPrice="39.95" Quantity="1"/>
</LineItem>

1 row selected.
```

Example 3–38 illustrates the common mistake of using SQL function `updateXML` to update a *node that occurs multiple times* in a collection. The `UPDATE` statement sets the value of the text node of a `Description` element to `The Wizard of Oz`, where the current value of the text node is `Sisters`. The statement includes an `XMLExists` expression in the `WHERE` clause that identifies the set of nodes to be updated.

Example 3–38 Incorrectly Updating a Node That Occurs Multiple Times in a Collection

```
SELECT XMLCast(des.COLUMN_VALUE AS VARCHAR2(256))
  FROM purchaseorder,
       XMLTable('$p/PurchaseOrder/LineItems/LineItem/Description'
                PASSING OBJECT_VALUE AS "p") des
  WHERE XMLExists('$p/PurchaseOrder[Reference="SBELL-2002100912333601PDT"]'
                  PASSING OBJECT_VALUE AS "p");

XMLCAST(DES.COLUMN_VALUEASVARCHAR2(256))
----------------------------------------
The Lady Vanishes
The Unbearable Lightness Of Being
Sisters

3 rows selected.

UPDATE purchaseorder p
  SET p.OBJECT_VALUE =
        updateXML(p.OBJECT_VALUE,
                  '/PurchaseOrder/LineItems/LineItem/Description/text()',
                  'The Wizard of Oz')
    WHERE XMLExists('$p/PurchaseOrder/LineItems/LineItem[Description="Sisters"]'
                    PASSING OBJECT_VALUE AS "p")
      AND XMLExists('$p/PurchaseOrder[Reference="SBELL-2002100912333601PDT"]'
                    PASSING OBJECT_VALUE AS "p");

1 row updated.
```

```
SELECT XMLCast(des.COLUMN_VALUE AS VARCHAR2(256))
  FROM purchaseorder,
       XMLTable('$p/PurchaseOrder/LineItems/LineItem/Description'
                PASSING OBJECT_VALUE AS "p") des
  WHERE XMLExists('$p/PurchaseOrder[Reference="SBELL-2002100912333601PDT"]'
                  PASSING OBJECT_VALUE AS "p");

XMLCAST(DES.COLUMN_VALUEASVARCHAR2(256))
----------------------------------------
The Wizard of Oz
The Wizard of Oz
The Wizard of Oz

3 rows selected.
```

In Example 3–38, instead of updating only the intended node, SQL function `updateXML` updates the values of *all* text nodes that belong to the `Description` element. This is the correct `updateXML` behavior, but it is not what was intended.

*A `WHERE` clause can be used only to identify which **documents** must be updated, not which **nodes** within a document must be updated.*

After the document has been selected, the *XPath expression* passed to `updateXML` determines which *nodes* within the document must be updated. In this case, the XPath expression identifies all three `Description` nodes, so all three of the associated text nodes were updated.

To correctly use SQL function `updateXML` to update a node that occurs multiple times within a collection, use the XPath expression passed to `updateXML` to identify which nodes in the XML document to update. By introducing the appropriate predicate into the XPath expression, you can limit which nodes in the document are updated. Example 3–39 illustrates the correct way to update one node within a collection.

Example 3–39 Correctly Updating a Node That Occurs Multiple Times in a Collection

```
SELECT XMLCast(des.COLUMN_VALUE AS VARCHAR2(256))
  FROM purchaseorder,
       XMLTable('$p/PurchaseOrder/LineItems/LineItem/Description'
                PASSING OBJECT_VALUE AS "p") des
  WHERE XMLExists('$p/PurchaseOrder[Reference="SBELL-2002100912333601PDT"]'
                  PASSING OBJECT_VALUE AS "p");

XMLCAST(DES.COLUMN_VALUEASVARCHAR2(256))
----------------------------------------

A Night to Remember
The Unbearable Lightness Of Being
Sisters

3 rows selected.

UPDATE purchaseorder p
 SET p.OBJECT_VALUE =
       updateXML(
         p.OBJECT_VALUE,
         '/PurchaseOrder/LineItems/LineItem/Description[text()="Sisters"]/text()',
         'The Wizard of Oz')
 WHERE XMLExists('$p/PurchaseOrder[Reference="SBELL-2002100912333601PDT"]'
                 PASSING OBJECT_VALUE AS "p");

1 row updated.
```

```
SELECT XMLCast(des.COLUMN_VALUE AS VARCHAR2(256))
  FROM purchaseorder,
       XMLTable('$p/PurchaseOrder/LineItems/LineItem/Description'
                PASSING OBJECT_VALUE AS "p") des
  WHERE XMLExists('$p/PurchaseOrder[Reference="SBELL-2002100912333601PDT"]'
                  PASSING OBJECT_VALUE AS "p");

XMLCAST(DES.COLUMN_VALUEASVARCHAR2(256))
----------------------------------------
A Night to Remember
The Unbearable Lightness Of Being
The Wizard of Oz

3 rows selected.
```

SQL function `updateXML` lets multiple changes be made to the document in one statement. Example 3–40 shows how to change the values of text nodes belonging to the `User` and `SpecialInstructions` elements in one statement.

Example 3–40 Changing Text Node Values using UPDATEXML

```
SELECT XMLCast(XMLQuery('$p/PurchaseOrder/CostCenter'
                        PASSING OBJECT_VALUE AS "p" RETURNING CONTENT)
               AS VARCHAR2(4)) "Cost Center",
       XMLCast(XMLQuery('$p/PurchaseOrder/SpecialInstructions'
                        PASSING OBJECT_VALUE AS "p" RETURNING CONTENT)
               AS VARCHAR2(2048)) "Instructions"
  FROM purchaseorder
  WHERE XMLExists('$p/PurchaseOrder[Reference="SBELL-2002100912333601PDT"]'
                  PASSING OBJECT_VALUE AS "p");

Cost Center  Instructions
------------ ------------
S30          Air Mail

1 row selected.
```

The following single `UPDATE` SQL statement changes the `User` and `SpecialInstructions` element text node values:

```
UPDATE purchaseorder
  SET OBJECT_VALUE =
        updateXML(OBJECT_VALUE,
                  '/PurchaseOrder/CostCenter/text()',
                  'B40',
                  '/PurchaseOrder/SpecialInstructions/text()',
                  'Priority Overnight Service')
  WHERE XMLExists('$p/PurchaseOrder[Reference="SBELL-2002100912333601PDT"]'
                  PASSING OBJECT_VALUE AS "p");

1 row updated.

SELECT XMLCast(XMLQuery('$p/PurchaseOrder/CostCenter'
                        PASSING OBJECT_VALUE AS "p" RETURNING CONTENT)
               AS VARCHAR2(4)) "Cost Center",
       XMLCast(XMLQuery('$p/PurchaseOrder/SpecialInstructions'
                        PASSING OBJECT_VALUE AS "p" RETURNING CONTENT)
               AS VARCHAR2(2048)) "Instructions"
  FROM purchaseorder
```

```
  WHERE XMLExists('$p/PurchaseOrder[Reference="SBELL-2002100912333601PDT"]'
                  PASSING OBJECT_VALUE AS "p");

Cost Center  Instructions
------------ -------------------------
B40          Priority Overnight Service

1 row selected.
```

Updating XML Schema-Based and Non-Schema-Based XML Documents

The way SQL functions such as `updateXML` modify an XML document depends on how the XML document is stored and whether it is based on an XML schema:

- XML documents stored in `CLOB` values – When a SQL function such as `updateXML` modifies an XML document stored as a `CLOB` (whether or not it is XML schema-based), Oracle XML DB performs the update by creating a Document Object Model (DOM) from the document and using DOM API methods to modify the appropriate XML data. After modification, the updated DOM is returned back to the underlying `CLOB` value.
- XML documents stored object-relationally – When a SQL function such as `updateXML` modifies an XML schema-based document that is stored object-relationally, Oracle XML DB can use XPath rewrite to modify the underlying objects in place. This is a partial update, which translates the XPath argument to the SQL function into an equivalent SQL operation. The SQL operation then directly modifies the attributes of underlying objects. Such a partial update can be much quicker than a DOM-based update. This can improve performance significantly when executing SQL code that applies a SQL function such as `updateXML` to a large number of documents.
- XML documents stored as binary XML – When a SQL function such as `updateXML` is used on a binary XML column, Oracle XML DB often need not build a DOM. The exact portion of the document that must be updated is calculated using query evaluation techniques such as streaming and `XMLIndex`. The updated data is written to disk starting only where the first change occurs—anything before that is unchanged. In addition, if SecureFile LOBs are used for storing the data (the default behavior), then the change is applied in a sliding manner, without causing the rest of the LOB to be rewritten. That is, with SecureFile LOB storage of binary XML data, only the data that is actually changed is updated. This can significantly improve performance relative to unstructured storage. These optimizations apply to both non-schema-based and XML schema-based data.

See Also: Chapter 8, "XPath Rewrite for Structured Storage"

Namespace Support in Oracle XML DB

Namespace support is a key feature of the W3C XML Recommendations. Oracle XML DB fully supports the W3C Namespace Recommendation. All `XMLType` methods and XML-specific SQL functions work with XPath expressions that include namespace prefixes. All methods and functions accept an optional `namespace` argument that provides the namespace declarations for correctly resolving namespace prefixes used in XPath expressions.

The `namespace` parameter is required whenever the provided XPath expression contains namespace prefixes. When parameter `namespace` is provided, it must provide an explicit declaration for the default namespace in addition to the prefixed

namespaces, unless the default namespace is the `noNamespace` namespace. When parameter `namespace` is *not* provided, Oracle XML DB makes the following assumptions about the XPath expression:

- If the content of the `XMLType` instance is *not* based on a registered XML schema, then any term in the XPath expression that does include a namespace prefix is assumed to be in the `noNamespace` namespace.
- If the content of the `XMLType` is based on a registered XML schema, then any term in the XPath expression that does not include a namespace prefix is assumed to be in the `targetNamespace` declared by the XML schema, if any. If the XML schema does not declare a `targetnamespace`, then names `noNamespace` is used.

Failing to correctly define the namespaces required to resolve XPath expressions results in XPath-based operations not working as expected. When the namespace declarations are incorrect or missing, the result of the operation is normally *null*, rather than an error. To avoid confusion, whenever any namespaces other than `noNamespace` are present in either the XPath expression or the target XML document, pass the *complete set of namespace declarations*, including the declaration for the *default* namespace.

How Oracle XML DB Processes XMLType Methods and SQL Functions

Oracle XML DB processes SQL/XML access and query functions such as `XMLQuery` and `XMLType` methods using DOM-based or SQL-based techniques:

- DOM-based `XMLType` processing – Oracle XML DB performs the required processing by constructing a DOM from the contents of the `XMLType` object. It uses methods provided by the DOM API to perform the required operation on the DOM. If the operation involves updating the DOM tree, then the entire XML document has to be written back to disk when the operation is completed. The process of using DOM-based operations on `XMLType` data is referred to as **functional evaluation**.

  The advantage of functional evaluation is that it can be used regardless of the storage model (structured, binary XML, or unstructured) used for the `XMLType` instance. The disadvantage of functional evaluation is that it much more *expensive* than XPath rewrite, and *does not scale* across large numbers of XML documents.

- SQL-based `XMLType` processing – Oracle XML DB constructs a SQL statement that performs the processing required to complete the function or method. The SQL statement works directly against the object-relational data structures that underlie a schema-based `XMLType`. This process is referred to as **XPath rewrite**. See Chapter 8, "XPath Rewrite for Structured Storage".

  The advantage of XPath rewrite is that it lets Oracle XML DB evaluate XPath-based SQL functions and methods at near *relational speeds*. This lets these operations scale across *large numbers of XML documents*. The disadvantage of XPath rewrite is that since it relies on direct access and updating the objects used to store the XML document, it can be used only when the `XMLType` instance is stored using XML *schema-based object-relational* storage techniques.

- Streaming evaluation of binary XML data – If you use binary XML as the `XMLType` storage model, then XPath expressions used in SQL/XML access and query functions such as `XMLQuery` are evaluated in a streaming fashion, without recourse to building a DOM.

Generating XML Data from Relational Data

This section presents examples of using Oracle XML DB to generate XML data from relational data.

See Also:

- Chapter 5, "Using XQuery with Oracle XML DB"
- Chapter 18, "Generating XML Data from the Database"

Generating XML Data from Relational Data using SQL/XML Functions

You can use standard SQL/XML functions to generate one or more XML documents. SQL/XML function `XMLQuery` is the most general way to do this. Other SQL/XML functions that you can use for this are the following:

- `XMLElement` creates a element
- `XMLAttributes` adds attributes to an element
- `XMLForest` creates forest of elements
- `XMLAgg` creates a single element from a collection of elements

The query in Example 3–41 uses these functions to generate an XML document that contains information from the tables `departments`, `locations`, `countries`, `employees`, and `jobs`.

Example 3–41 Generating XML Data using SQL/XML Functions

```
SELECT XMLElement(
         "Department",
         XMLAttributes(d.Department_id AS "DepartmentId"),
         XMLForest(d.department_name AS "Name"),
         XMLElement(
           "Location",
           XMLForest(street_address AS "Address",
                     city AS "City",
                     state_province AS "State",
                     postal_code AS "Zip",
                     country_name AS "Country")),
           XMLElement(
             "EmployeeList",
             (SELECT XMLAgg(
                       XMLElement(
                         "Employee",
                         XMLAttributes(e.employee_id AS "employeeNumber"),
                         XMLForest(
                           e.first_name AS "FirstName",
                           e.last_name AS "LastName",
                           e.email AS "EmailAddress",
                           e.phone_number AS "PHONE_NUMBER",
                           e.hire_date AS "StartDate",
                           j.job_title AS "JobTitle",
                           e.salary AS "Salary",
                           m.first_name || ' ' || m.last_name AS "Manager"),
                         XMLElement("Commission", e.commission_pct)))
                FROM hr.employees e, hr.employees m, hr.jobs j
                WHERE e.department_id = d.department_id
                  AND j.job_id = e.job_id
                  AND m.employee_id = e.manager_id)))
```

```
  AS XML
  FROM hr.departments d, hr.countries c, hr.locations l
  WHERE department_name = 'Executive'
    AND d.location_id = l.location_id
    AND l.country_id  = c.country_id;
```

The query returns the following XML:

```
XML
--------------------------------------------------------------------------------
<Department DepartmentId="90"><Name>Executive</Name><Location><Address>2004
 Charade Rd</Address><City>Seattle</City><State>Washingto
n</State><Zip>98199</Zip><Country>United States of
 America</Country></Location><EmployeeList><Employee
 employeeNumber="101"><FirstNa
me>Neena</FirstName><LastName>Kochhar</LastName><EmailAddress>NKOCHHAR</EmailAdd
ess><PHONE_NUMBER>515.123.4568</PHONE_NUMBER><Start
Date>2005-09-21</StartDate><JobTitle>Administration Vice
 President</JobTitle><Salary>17000</Salary><Manager>Steven King</Manager><Com
mission></Commission></Employee><Employee
 employeeNumber="102"><FirstName>Lex</FirstName><LastName>De
 Haan</LastName><EmailAddress>L
DEHAAN</EmailAddress><PHONE_NUMBER>515.123.4569</PHONE
NUMBER><StartDate>2001-01-13</StartDate><JobTitle>Administration Vice Presiden
t</JobTitle><Salary>17000</Salary><Manager>Steven
 King</Manager><Commission></Commission></Employee></EmployeeList></Department>
```

This query generates element `Department` for each row in the `departments` table.

- Each `Department` element contains attribute `DepartmentID`. The value of `DepartmentID` comes from the `department_id` column. The `Department` element contains sub-elements `Name`, `Location`, and `EmployeeList`.
- The text node associated with the `Name` element comes from the `name` column in the `departments` table.
- The `Location` element has child elements `Address`, `City`, `State`, `Zip`, and `Country`. These elements are constructed by creating a forest of named elements from columns in the `locations` and `countries` tables. The values in the columns become the text node for the named element.
- The `EmployeeList` element contains an aggregation of `Employee` Elements. The content of the `EmployeeList` element is created by a subquery that returns the set of rows in the `employees` table that correspond to the current department. Each `Employee` element contains information about the employee. The contents of the elements and attributes for each `Employee` element is taken from tables `employees` and `jobs`.

The output generated by SQL/XML functions is generally *not* pretty-printed. The only exception is function `XMLSerialize`—use `XMLSerialize` to pretty-print. This lets the other SQL/XML functions (1) avoid creating a full DOM when generating the required output, and (2) reduce the size of the generated document. This lack of pretty-printing by most SQL/XML functions does not matter to most applications. However, it makes verifying the generated output manually more difficult.

Example 3–42 Creating XMLType Views Over Conventional Relational Tables

```
CREATE OR REPLACE VIEW department_xml OF XMLType
  WITH OBJECT ID (substr(
                    XMLCast(
                      XMLQuery('$p/Department/Name'
```

```
                                PASSING OBJECT_VALUE AS "p" RETURNING CONTENT)
                        AS VARCHAR2(30)),
                    1,
                    128))
  AS
  SELECT XMLElement(
           "Department",
           XMLAttributes(d.department_id AS "DepartmentId"),
           XMLForest(d.department_name AS "Name"),
           XMLElement("Location", XMLForest(street_address AS "Address",
                                            city AS "City",
                                            state_province AS "State",
                                            postal_code AS "Zip",
                                            country_name AS "Country")),
           XMLElement(
             "EmployeeList",
             (SELECT XMLAgg(
                       XMLElement(
                          "Employee",
                          XMLAttributes(e.employee_id AS "employeeNumber"),
                          XMLForest(e.first_name AS "FirstName",
                                    e.last_name AS "LastName",
                                    e.email AS "EmailAddress",
                                    e.phone_number AS "PHONE_NUMBER",
                                    e.hire_date AS "StartDate",
                                    j.job_title AS "JobTitle",
                                    e.salary AS "Salary",
                                    m.first_name || ' ' ||
                                    m.last_name AS "Manager"),
                          XMLElement("Commission", e.commission_pct)))
                FROM hr.employees e, hr.employees m, hr.jobs j
                WHERE e.department_id = d.department_id
                  AND j.job_id = e.job_id
                  AND m.employee_id = e.manager_id))).extract('/*')
    AS XML
    FROM hr.departments d, hr.countries c, hr.locations l
    WHERE d.location_id = l.location_id
      AND l.country_id  = c.country_id;

View created.
```

The `XMLType` view lets relational data be persisted as XML content. Rows in `XMLType` views can be persisted as documents in Oracle XML DB Repository. The contents of an `XMLType` view can be queried, as shown in Example 3–43.

Example 3–43 shows a simple query against an `XMLType` view. The XPath expression passed to SQL/XML function `XMLExists` restricts the result set to the node that contains the `Executive` department information. The result is shown pretty-printed here for clarity.

Example 3–43 Querying XMLType Views

```
SELECT OBJECT_VALUE FROM department_xml
  WHERE XMLExists('$p/Department[Name="Executive"]' PASSING OBJECT_VALUE AS "p");

OBJECT_VALUE
-----------------------------------------------
<Department DepartmentId="90">
  <Name>Executive</Name>
  <Location>
```

```
    <Address>2004 Charade Rd</Address>
    <City>Seattle</City>
    <State>Washington</State>
    <Zip>98199</Zip>
    <Country>United States of America</Country>
  </Location>
  <EmployeeList>
    <Employee employeeNumber="101">
      <FirstName>Neena</FirstName>
      <LastName>Kochhar</LastName>
      <EmailAddress>NKOCHHAR</EmailAddress>
      <PHONE_NUMBER>515.123.4568</PHONE_NUMBER>
      <StartDate>2005-09-21</StartDate>
      <JobTitle>Administration Vice President</JobTitle>
      <Salary>17000</Salary>
      <Manager>Steven King</Manager>
      <Commission/>
    </Employee>
    <Employee employeeNumber="102">
      <FirstName>Lex</FirstName>
      <LastName>De Haan</LastName>
      <EmailAddress>LDEHAAN</EmailAddress>
      <PHONE_NUMBER>515.123.4569</PHONE_NUMBER>
      <StartDate>2001-01-13</StartDate>
      <JobTitle>Administration Vice President</JobTitle>
      <Salary>17000</Salary>
      <Manager>Steven King</Manager>
      <Commission/>
    </Employee>
  </EmployeeList>
</Department>

1 row selected.
```

As can be seen from the following execution plan output, Oracle XML DB is able to correctly rewrite the XPath-expression argument in the `XMLExists` expression into a `SELECT` statement on the underlying relational tables.

```
SELECT OBJECT_VALUE FROM department_xml
  WHERE XMLExists('$p/Department[Name="Executive"]' PASSING OBJECT_VALUE AS "p");

PLAN_TABLE_OUTPUT
-----------------------------------------------------------------------------------------------------
Plan hash value: 2414180351

-----------------------------------------------------------------------------------------------------
| Id  | Operation                      | Name               | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT               |                    |     1 |    80 |     3   (0)| 00:00:01 |
|   1 |  SORT AGGREGATE                |                    |     1 |   115 |            |          |
|*  2 |   HASH JOIN                    |                    |    10 |  1150 |     7  (15)| 00:00:01 |
|*  3 |    HASH JOIN                   |                    |    10 |   960 |     5  (20)| 00:00:01 |
|   4 |     TABLE ACCESS BY INDEX ROWID| EMPLOYEES          |    10 |   690 |     2   (0)| 00:00:01 |
|*  5 |      INDEX RANGE SCAN          | EMP_DEPARTMENT_IX  |    10 |       |     1   (0)| 00:00:01 |
|   6 |     TABLE ACCESS FULL          | JOBS               |    19 |   513 |     2   (0)| 00:00:01 |
|   7 |    TABLE ACCESS FULL           | EMPLOYEES          |   107 |  2033 |     2   (0)| 00:00:01 |
|   8 |  NESTED LOOPS                  |                    |     1 |    80 |     3   (0)| 00:00:01 |
|   9 |   NESTED LOOPS                 |                    |     1 |    68 |     3   (0)| 00:00:01 |
|* 10 |    TABLE ACCESS FULL           | DEPARTMENTS        |     1 |    19 |     2   (0)| 00:00:01 |
|  11 |    TABLE ACCESS BY INDEX ROWID | LOCATIONS          |     1 |    49 |     1   (0)| 00:00:01 |
|* 12 |     INDEX UNIQUE SCAN          | LOC_ID_PK          |     1 |       |     0   (0)| 00:00:01 |
|* 13 |   INDEX UNIQUE SCAN            | COUNTRY_C_ID_PK    |     1 |    12 |     0   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------------
```

```
Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("M"."EMPLOYEE_ID"="E"."MANAGER_ID")
   3 - access("J"."JOB_ID"="E"."JOB_ID")
   5 - access("E"."DEPARTMENT_ID"=:B1)
  10 - filter("D"."DEPARTMENT_NAME"='Executive')
  12 - access("D"."LOCATION_ID"="L"."LOCATION_ID")
  13 - access("L"."COUNTRY_ID"="C"."COUNTRY_ID")

30 rows selected.
```

Note: XPath rewrite on XML expressions that operate on `XMLType` views is only supported when nodes referenced in the XPath expression are *not* descendants of an element created using SQL function `XMLAgg`.

Generating XML Data from Relational Data using DBURITYPE

You can also generate XML from relational data using SQL function `DBURIType`. Function `DBURIType` exposes one or more rows in a given table or view as a single XML document. The name of the root element is derived from the name of the table or view. The root element contains a set of `ROW` elements. There is one `ROW` element for each row in the table or view. The children of each `ROW` element are derived from the columns in the table or view. Each child element contains a text node with the value of the column for the given row.

Example 3–44 shows how to use SQL function `DBURIType` to access the contents of table `departments` in database schema `HR`. It uses method `getXML()` to return the resulting document as an `XMLType` instance.

Example 3–44 Generating XML Data from a Relational Table using DBURIType and getXML()

```
SELECT DBURIType('/HR/DEPARTMENTS').getXML() FROM DUAL;

DBURITYPE('/HR/DEPARTMENTS').GETXML()
-------------------------------------------------------
<?xml version="1.0"?>
<DEPARTMENTS>
 <ROW>
  <DEPARTMENT_ID>10</DEPARTMENT_ID>
  <DEPARTMENT_NAME>Administration</DEPARTMENT_NAME>
  <MANAGER_ID>200</MANAGER_ID>
  <LOCATION_ID>1700</LOCATION_ID>
 </ROW>
...
 <ROW>
  <DEPARTMENT_ID>20</DEPARTMENT_ID>
  <DEPARTMENT_NAME>Marketing</DEPARTMENT_NAME>
  <MANAGER_ID>201</MANAGER_ID>
  <LOCATION_ID>1800</LOCATION_ID>
 </ROW>
</DEPARTMENTS>
```

Example 3–45 shows how to use an XPath predicate to restrict the rows that are included in an XML document generated using `DBURIType`. The XPath expression in the example restricts the XML document to `DEPARTMENT_ID` columns with value `10`.

Example 3–45 Restricting Rows using an XPath Predicate

```
SELECT DBURIType('/HR/DEPARTMENTS/ROW[DEPARTMENT_ID="10"]').getXML()
  FROM DUAL;

DBURITYPE('/HR/DEPARTMENTS/ROW[DEPARTMENT_ID="10"]').GETXML()
-------------------------------------------------------------------
<?xml version="1.0"?>
 <ROW>
  <DEPARTMENT_ID>10</DEPARTMENT_ID>
  <DEPARTMENT_NAME>Administration</DEPARTMENT_NAME>
  <MANAGER_ID>200</MANAGER_ID>
  <LOCATION_ID>1700</LOCATION_ID>
 </ROW>

1 row selected.
```

SQL function `DBURIType` provides a simple way to expose some or all rows in a relational table as one or more XML documents. The URL passed to function `DBURIType` can be extended to return a single column from the view or table, but in that case the URL must also include predicates that identify a single row in the target table or view.

Example 3–46 illustrates this. The predicate `[DEPARTMENT_ID="10"]` causes the query to return the value of column `department_name` for the `departments` row where column `department_id` has the value `10`.

Example 3–46 Restricting Rows and Columns using an XPath Predicate

```
SELECT DBURIType(
         '/HR/DEPARTMENTS/ROW[DEPARTMENT_ID="10"]/DEPARTMENT_NAME').getXML()
  FROM DUAL;

DBURITYPE('/HR/DEPARTMENTS/ROW[DEPARTMENT_ID="10"]/DEPARTMENT_NAME').GETXML()
-------------------------------------------------------------------------------
<?xml version="1.0"?>
 <DEPARTMENT_NAME>Administration</DEPARTMENT_NAME>

1 row selected.
```

SQL function `DBURIType` is less flexible than the SQL/XML functions:

- It provides no way to control the shape of the generated document.
- The data can come only from a single table or view.
- The generated document consists of one or more `ROW` elements. Each `ROW` element contains a child for each column in the target table.
- The names of the child elements are derived from the column names.

To control the names of the XML elements, to include columns from more than one table, or to control which columns from a table appear in the generated document, create a relational view that exposes the desired set of columns as a single row, and then use function `DBURIType` to generate an XML document from the contents of that view.

XSL Transformation and Oracle XML DB

The W3C XSLT Recommendation defines an XML language for specifying how to transform XML documents from one form to another. Transformation can include

mapping from one XML schema to another or mapping from XML to some other format such as HTML or WML.

> **See Also:** http://www.w3.org/XML/Schema for information about the XSLT standard

XSL transformation is typically expensive in terms of the amount of memory and processing required. Both the source document and the style sheet must be parsed and loaded into memory structures that allow random access to different parts of the documents. Most XSL processors use DOM to provide the dynamic memory representation of both documents. The XSL processor then applies the style sheet to the source document, generating a third document.

Oracle XML DB includes an XSLT processor that lets XSL transformations be performed *inside the database*. In this way, Oracle XML DB can provide XML-specific memory optimizations that significantly reduce the memory required to perform the transformation. It can also eliminate overhead associated with parsing the documents. These optimizations are only available when the source for the transformation is a *schema-based* XML document, however.

Oracle XML provides three ways to invoke the XSL processor:

- SQL function `XMLtransform`
- `XMLType` method `transform()`
- PL/SQL package `DBMS_XSLPROCESSOR`

Of these different ways to transform XML data, `DBMS_XSLPROCESSOR` has the best performance, because the style sheet is parsed only once.

Each of these XML transformation methods takes as input a source XML document and an XSL style sheet in the form of `XMLType` instances. For SQL function `XMLtransform` and `XMLType` method `transform()`, the result of the transformation can be an XML document or a non-XML document, such as HTML. However, for PL/SQL package `DBMS_XSLPROCESSOR`, the result of the transformation is expected to be a valid XML document. Any HTML generated by a transformation using package `DBMS_XSLPROCESSOR` is XHTML, which is both valid XML and valid HTML.

Example 3–47 shows *part* of an XSLT style sheet, `PurchaseOrder.xsl`. The complete style sheet is given in "XSL Style Sheet Example, PurchaseOrder.xsl" on page A-38.

Example 3–47 XSLT Style Sheet Example: PurchaseOrder.xsl

```
<?xml version="1.0" encoding="WINDOWS-1252"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xdb="http://xmlns.oracle.com/xdb"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <xsl:template match="/">
    <html>
      <head/>
      <body bgcolor="#003333" text="#FFFFCC" link="#FFCC00" vlink="#66CC99" alink="#669999">
        <FONT FACE="Arial, Helvetica, sans-serif">
          <xsl:for-each select="PurchaseOrder"/>
          <xsl:for-each select="PurchaseOrder">
            <center>
              <span style="font-family:Arial; font-weight:bold">
                <FONT COLOR="#FF0000">
                  <B>PurchaseOrder </B>
                </FONT>
              </span>
            </center>
```

```
  <br/>
  <center>
    <xsl:for-each select="Reference">
      <span style="font-family:Arial; font-weight:bold">
        <xsl:apply-templates/>
      </span>
    </xsl:for-each>
  </center>
</xsl:for-each>
<P>
  <xsl:for-each select="PurchaseOrder">
    <br/>
  </xsl:for-each>
  <P/>
  <P>
    <xsl:for-each select="PurchaseOrder">
      <br/>
    </xsl:for-each>
  </P>
</P>
<xsl:for-each select="PurchaseOrder"/>
<xsl:for-each select="PurchaseOrder">
  <table border="0" width="100%" BGCOLOR="#000000">
    <tbody>
      <tr>
        <td WIDTH="296">
          <P>
            <B>
              <FONT SIZE="+1" COLOR="#FF0000" FACE="Arial, Helvetica, sans-serif">Internal</FONT>
            </B>
          </P>

          ...

        </td>
        <td width="93"/>
        <td valign="top" WIDTH="340">
          <B>
            <FONT COLOR="#FF0000">
              <FONT SIZE="+1">Ship To</FONT>
            </FONT>
          </B>
          <xsl:for-each select="ShippingInstructions">
            <xsl:if test="position()=1"/>
          </xsl:for-each>
          <xsl:for-each select="ShippingInstructions">
          </xsl:for-each>

            ...
```

These is nothing Oracle XML DB-specific about the style sheet of Example 3–47. A style sheet can be stored in an `XMLType` table or column or stored as non-schema-based XML data inside Oracle XML DB Repository.

Performing transformations inside the database lets Oracle XML DB optimize features such as memory usage, I/O operations, and network traffic. These optimizations are particularly effective when the transformation operates on a *small subset* of the nodes in the source document.

In traditional XSL processors, the entire source document must be parsed and loaded into memory before XSL processing can begin. This process requires significant amounts of memory and processor. When only a small part of the document is processed this is inefficient.

When Oracle XML DB performs XSL transformations on a *schema-based* XML document there is no need to parse the document before processing can begin. The lazily loaded virtual DOM eliminates the need to parse the document, by loading content directly from disk as the nodes are accessed. The lazy load also reduces the amount of memory required to perform the transformation, because only the parts of the document that are processed are loaded into memory.

Example 3–48 shows how to use SQL function `XMLtransform` to apply an XSL style sheet to a document stored in an `XMLType` table, producing HTML code. SQL function `XDBURIType` reads the XSL style sheet from Oracle XML DB Repository.

In the interest of brevity, only part of the result of the transformation is shown in Example 3–48. Omitted parts are indicated with an ellipsis (`. . .`). Figure 3–7 shows what the transformed result looks like in a Web browser.

Example 3–48 Applying a Style Sheet using TRANSFORM

```
SELECT
  XMLtransform(
    OBJECT_VALUE,
    XDBURIType('/source/schemas/poSource/xsl/purchaseOrder.xsl').getXML())
  FROM purchaseorder
  WHERE XMLExists('$p/PurchaseOrder[Reference="SBELL-2002100912333601PDT"]'
                  PASSING OBJECT_VALUE AS "p");

XMLTRANSFORM(OBJECT_VALUE, XDBURITYPE('/SOURCE/SCHEMAS/POSOURCE/XSL/PURCHASEORDER.XSL').GET
--------------------------------------------------------------------------------------------
<html xmlns:xdb="http://xmlns.oracle.com/xdb"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <head/>
  <body bgcolor="#003333" text="#FFFFCC" link="#FFCC00" vlink="#66CC99" alink="#669999">
    <FONT FACE="Arial, Helvetica, sans-serif">
      <center>
        <span style="font-family:Arial; font-weight:bold">
          <FONT COLOR="#FF0000">
            <B>PurchaseOrder </B>
          </FONT>
        </span>
      </center>
      <br/>
      <center>
        <span style="font-family:Arial; font-weight:bold">SBELL-2002100912333601PDT</span>
      </center>
      <P>
        <br/>
        <P/>
        <P>
          <br/>
        </P>
      </P>
      <table border="0" width="100%" BGCOLOR="#000000">
        <tbody>
          <tr>
            <td WIDTH="296">
              <P>
                <B>
                  <FONT SIZE="+1" COLOR="#FF0000" FACE="Arial, Helvetica,
                        sans-serif">Internal</FONT>
                </B>
              </P>
              <table border="0" width="98%" BGCOLOR="#000099">
                                                          . . .
              </table>
            </td>
            <td width="93">
```

```
            </td>
            <td valign="top" WIDTH="340">
              <B>
                <FONT COLOR="#FF0000">
                  <FONT SIZE="+1">Ship To</FONT>
                </FONT>
              </B>
              <table border="0" BGCOLOR="#999900">
                . . .
              </table>
            </td>
          </tr>
        </tbody>
      </table>
      <br/>
      <B>
        <FONT COLOR="#FF0000" SIZE="+1">Items:</FONT>
      </B>
      <br/>
      <br/>
      <table border="0">
        . . .
      </table>
    </FONT>
  </body>
</html>

1 row selected.
```

See Also: Chapter 11, "Transforming and Validating XMLType Data"

Using Oracle XML DB Repository

Oracle XML DB Repository makes it possible to organize XML content using a file/folder metaphor. This lets you use a URL to uniquely identify XML documents stored in the database. This approach appeals to XML developers used to using constructs such as URLs and XPath expressions to identify content.

Oracle XML DB Repository is modelled on the DAV standard. The DAV standard uses the term **resource** to describe any file or folder managed by a WebDAV server. A resource consists of a combination of metadata and content. The DAV specification defines the set of (system-defined) metadata properties that a WebDAV server is expected to maintain for each resource and the set of XML documents that a DAV server and DAV-enabled client uses to exchange metadata.

Although Oracle XML DB Repository can manage any kind of content, it provides specialized capabilities and optimizations related to managing resources where the content is XML.

Installing and Uninstalling Oracle XML DB Repository

All of the metadata and content managed by Oracle XML DB Repository is stored using a set of tables in the database schema owned by database schema (user account) `XDB`. User `XDB` is a locked account that is installed using DBCA or by running script `catqm.sql`. Script `catqm.sql` is located in the directory `ORACLE_HOME/rdbms/admin`. The repository can be uninstalled using DBCA or by running the script `catnoqm.sql`. Take great care when running `catnoqm.sql` as it *drops all content* stored in Oracle XML DB Repository and *invalidates* any `XMLType` tables or columns associated with registered XML schemas.

See Also: *Oracle Database 2 Day + Security Guide* for information about database schema `XDB`

Oracle XML DB Provides Name-Level Locking

When using a relational database to maintain hierarchical folder structures, ensuring a high degree of concurrency when adding and removing items in a folder is a challenge. In conventional file system there is no concept of a transaction. Each operation (add a file, create a subfolder, rename a file, delete a file, and so on) is treated as an atomic transaction. Once the operation has completed the change is immediately available to all other users of the file system.

Note: As a consequence of transactional semantics enforced by the database, folders created using SQL statements are *not* visible to other database users until the transaction is committed. *Concurrent* access to Oracle XML DB Repository is controlled by the same mechanism used to control concurrency in Oracle Database. The integration of the repository with Oracle Database provides *strong management options for XML content*.

One key advantage of Oracle XML DB Repository is the ability to use SQL for repository operations in the context of a logical transaction. Applications can create long-running transactions that include updates to one or more folders. In this situation, a conventional locking strategy that takes an exclusive lock on each updated folder or directory tree would quickly result in significant concurrency problems.

Oracle XML DB solves this by providing for name-level locking rather than folder-level locking. Repository operations such as creating, renaming, moving, or deleting a sub-folder or file do not require that your operation be granted an exclusive write lock on the target folder. The repository manages concurrent folder operations by locking the name within the folder rather than the folder itself. The name and the modification type are put on a queue.

Only when the transaction is committed is the folder locked and its contents modified. Hence Oracle XML DB lets multiple applications perform concurrent updates on the contents of a folder. The queue is also used to manage folder concurrency by preventing two applications from creating objects with the same name.

Queuing folder modifications until commit time also minimizes I/O when a number of changes are made to a single folder in the same transaction.

This is useful when several applications generate files quickly in the same directory, for example when generating trace or log files, or when maintaining a spool directory for printing or e-mail delivery.

Use Protocols or SQL to Access and Process Repository Content

You can work with content stored in Oracle XML DB Repository in these ways:

- Using industry standard protocols such as HTTP(S), WebDAV, and FTP to perform document-level operations such as insert, update, and delete.
- By directly accessing Oracle XML DB Repository content at the table or row level, using SQL.
- Using Oracle XML DB Content Connector—see Chapter 31, "Using Oracle XML DB Content Connector".

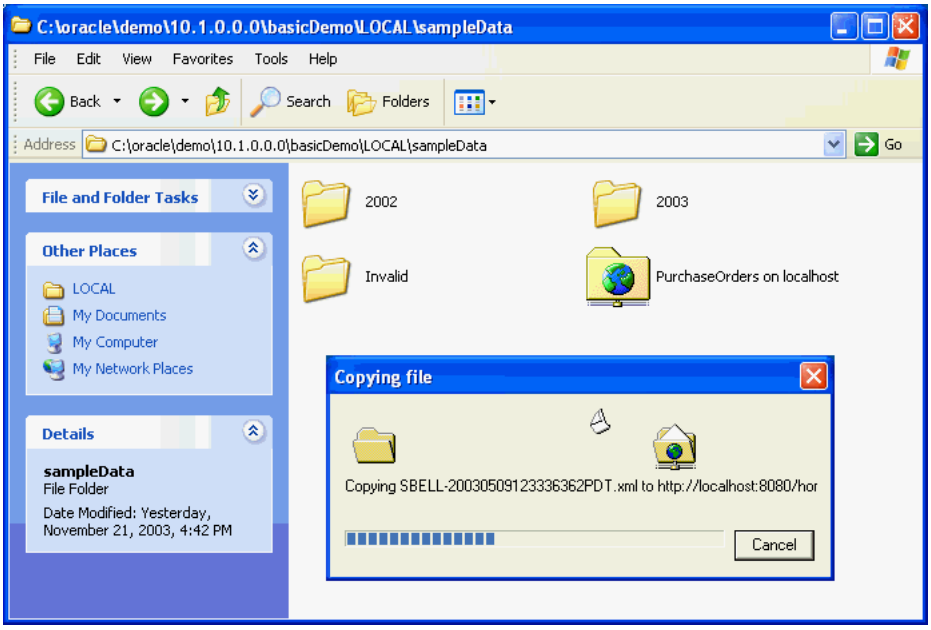
Storing and Retrieving Database Content using Standard Protocols

Oracle XML DB supports industry-standard internet protocols such as HTTP(S), WebDav, and FTP. The combination of protocol support and URL-based access makes it possible to insert, retrieve, update, and delete content stored in Oracle Database from standard desktop applications such as Windows Explorer, Microsoft Word, and XMLSpy.

Figure 3–4 shows Windows Explorer used to insert a folder from the local hard drive into Oracle Database. Windows Explorer includes support for the WebDAV protocol. WebDAV extends the HTTP standard, adding additional verbs that allow an HTTP server to act as a file server.

When a Windows Explorer copy operation or FTP input command is used to transfer a number of documents into Oracle XML DB Repository, each `put` or `post` command is treated as a separate atomic operation. This ensures that the client does not get confused if one of the file transfers fails. It also means that changes made to a document through a protocol are visible to other users as soon as the request has been processed.

Figure 3–4 Copying Files into Oracle XML DB Repository



Uploading Content to Oracle XML DB using FTP

Example 3–49 shows commands issued and output generated when a standard command line FTP tool loads documents into Oracle XML DB Repository:

Example 3–49 Uploading Content to the Repository using FTP

```
$ ftp mdrake-sun 2100
Connected to mdrake-sun.
220 mdrake-sun FTP Server (Oracle XML DB/Oracle Database 10g Enterprise Edition
Release 10.1.0.1.0 - Beta) ready.
Name (mdrake-sun:oracle10): QUINE
331 Password required for QUINE
Password: password
230 QUINE logged in
```

```
ftp> cd /source/schemas
250 CWD Command successful
ftp> mkdir PurchaseOrders
257 MKD Command successful
ftp> cd PurchaseOrders
250 CWD Command successful
ftp> mkdir 2002
257 MKD Command successful
ftp> cd 2002
250 CWD Command successful
ftp> mkdir "Apr"
257 MKD Command successful
ftp> put "Apr/AMCEWEN-20021009123336171PDT.xml"
"Apr/AMCEWEN-20021009123336171PDT.xml"
200 PORT Command successful
150 ASCII Data Connection
226 ASCII Transfer Complete
local: Apr/AMCEWEN-20021009123336171PDT.xml remote:
Apr/AMCEWEN-20021009123336171PDT.xml
4718 bytes sent in 0.0017 seconds (2683.41 Kbytes/s)
ftp> put "Apr/AMCEWEN-20021009123336271PDT.xml"
"Apr/AMCEWEN-20021009123336271PDT.xml"
200 PORT Command successful
150 ASCII Data Connection
226 ASCII Transfer Complete
local: Apr/AMCEWEN-20021009123336271PDT.xml remote:
Apr/AMCEWEN-20021009123336271PDT.xml
4800 bytes sent in 0.0014 seconds (3357.81 Kbytes/s)
.....
ftp> cd "Apr"
250 CWD Command successful
ftp> ls -l
200 PORT Command successful
150 ASCII Data Connection
-rw-r--r1 QUINE oracle 0 JUN 24 15:41 AMCEWEN-20021009123336171PDT.xml
-rw-r--r1 QUINE oracle 0 JUN 24 15:41 AMCEWEN-20021009123336271PDT.xml
-rw-r--r1 QUINE oracle 0 JUN 24 15:41 EABEL-20021009123336251PDT.xml
-rw-r--r1 QUINE oracle 0 JUN 24 15:41 PTUCKER-20021009123336191PDT.xml
-rw-r--r1 QUINE oracle 0 JUN 24 15:41 PTUCKER-20021009123336291PDT.xml
-rw-r--r1 QUINE oracle 0 JUN 24 15:41 SBELL-20021009123336231PDT.xml
-rw-r--r1 QUINE oracle 0 JUN 24 15:41 SBELL-20021009123336331PDT.xml
-rw-r--r1 QUINE oracle 0 JUN 24 15:41 SKING-20021009123336321PDT.xml
-rw-r--r1 QUINE oracle 0 JUN 24 15:41 SMCCAIN-20021009123336151PDT.xml
-rw-r--r1 QUINE oracle 0 JUN 24 15:41 SMCCAIN-20021009123336341PDT.xml
-rw-r--r1 QUINE oracle 0 JUN 24 15:41 VJONES-20021009123336301PDT.xml
226 ASCII Transfer Complete
remote: -l
959 bytes received in 0.0027 seconds (349.45 Kbytes/s)
ftp> cd ".."
250 CWD Command successful
....
ftp> quit
221 QUIT Goodbye.
$
```

The key point demonstrated by Figure 3–4 and Example 3–49 is that neither Windows Explorer nor an FTP tool is aware that it is working with Oracle XML DB. Since the tools and Oracle XML DB both support open Internet protocols they work with each other out of the box.

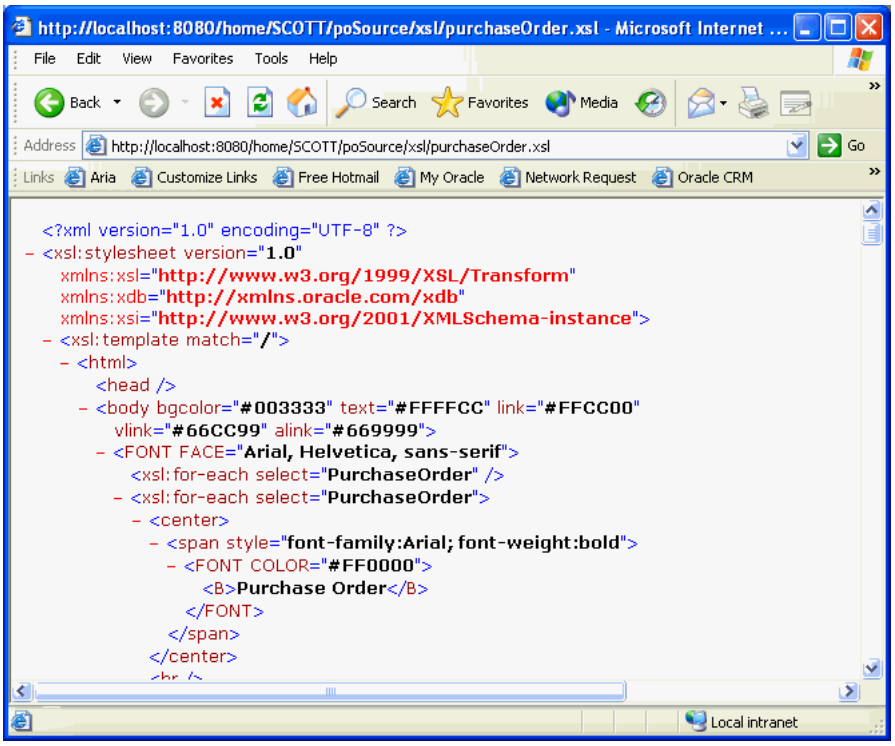
Any tool that understands the WebDAV or FTP protocol can be used to create content managed by Oracle XML DB Repository. No additional software has to installed on the client or the mid-tier.

When the contents of the folders are viewed using a tool such as Windows Explorer or FTP, the length of any schema-based XML documents contained in the folder is shown as zero (0) bytes. This was designed as such for two reasons:

- It is not clear what the size of the document should be. Is it the size of the `CLOB` instance generated by printing the document, or the number of bytes required to store the objects used to persist the document inside the database?
- Regardless of which definition is chosen, calculating and maintaining this information is costly.

Figure 3–5 shows Internet Explorer using a URL and the HTTP protocol to view an XML document stored in the database.

Figure 3–5 Path-Based Access using HTTP and a URL



Accessing Oracle XML DB Repository Programmatically

Oracle XML DB Repository can be accessed and updated directly from SQL. Thus, any application or programming language that can use SQL to interact with Oracle Database can also access and update content stored in the repository. Oracle XML DB includes PL/SQL package `DBMS_XDB`, which provides methods that allow resources to be created, modified, and deleted programmatically.

Example 3–50 shows how to create a simple text document resource using PL/SQL function `DBMS_XDB.createResource`.

Example 3–50 Creating a Text Document Resource using CREATERESOURCE

```
DECLARE
```

```
  res BOOLEAN;
BEGIN
  res := DBMS_XDB.createResource('/home/QUINE/NurseryRhyme.txt',
                                 bfilename('XMLDIR', 'tdadxdb-03-01.txt'),
                                 nls_charset_id('AL32UTF8'));
END;
/
```

Accessing and Updating XML Content in the Repository

This section describes features for accessing and updating Oracle XML DB Repository content.

Accessing XML Documents using SQL

Content stored in the repository can be accessed and updated from SQL and PL/SQL. You can interrogate the structure of the repository in complex ways. For example, you can query to determine how many files with extension `.xsl` are under a location other than `/home/mystylesheetdir`.

You can also mix path-based repository access with content-based access. You can, for example, ask "How many documents not under `/home/purchaseOrders` have a node identified by the XPath `/PurchaseOrder/User/text()` with a value of `KING`?"

All of the *metadata* for managing the repository is stored in a database schema owned by database schema (user account) `XDB`. User `XDB` is created during Oracle XML DB installation. The primary table in this schema is an `XMLType` table called `XDB$RESOURCE`. This contains one row for each resource (file or folder) in the repository. Documents in this table are referred to as **resource documents**. The XML schema that defines the structure of an Oracle XML DB resource document is registered under URL, `"http://xmlns.oracle.com/xdb/XDBResource.xsd`.

> **See Also:** *Oracle Database 2 Day + Security Guide* for information about database schema `XDB`

Repository Content is Exposed Through RESOURCE_VIEW and PATH_VIEW

Table `XDB$RESOURCE` is not directly exposed to SQL programmers. Instead, the contents of the repository are exposed through two public views, `RESOURCE_VIEW` and `PATH_VIEW`. Through these views, you can access and update both the metadata and the content of documents stored in the repository.

Both views contain a virtual column, `RES`. Use `RES` to access and update resource documents with SQL statements using a path notation. Operations on the views use underlying tables in the repository.

Use EXISTS_PATH and UNDER_PATH for Path-Based Predicates in a WHERE Clause

Oracle XML DB includes two repository-specific SQL functions: `exists_path` and `under_path`. Use these functions to include path-based predicates in the `WHERE` clause of a SQL statement. SQL operations can select repository content based on the location of the content in the repository folder hierarchy. The hierarchical repository index ensures that path-based queries are executed efficiently.

When *XML schema-based* XML documents are stored in the repository, the document content is stored as an object in the default table identified by the XML schema. The

repository contains only *metadata* about the document and a *pointer* (`REF` of `XMLType`) that identifies the row in the default table that contains the content.

You Can Also Store Non-XML Documents in the Repository

It is also possible to store other kinds of documents in the repository. When a document that is not XML or is not schema-based XML is stored in the repository, the document *content* is stored in a LOB along with the metadata about the document.

PL/SQL Packages to Create, Delete, Rename, Move,... Folders and Documents

Because you can access and update Oracle XML DB Repository using SQL, any application capable of calling a PL/SQL procedure can use the repository. All SQL and PL/SQL repository operations are transactional. Access to the repository and its contents is subject to both standard database security controls and repository access control lists (ACLs).

With supplied PL/SQL packages `DBMS_XDB`, `DBMS_XDBZ`, and `DBMS_XDB_VERSION`, you can create, delete, and rename documents and folders, move a file or folder within the folder hierarchy, set and change the access permissions on a file or folder, and initiate and manage versioning.

Example 3–51 uses PL/SQL package `DBMS_XDB` to create a set of subfolders beneath folder `/public`.

Example 3–51 Creating Folders using PL/SQL Package DBMS_XDB

```
DECLARE
  RESULT BOOLEAN;
BEGIN
  IF (NOT DBMS_XDB.existsResource('/public/mysource')) THEN
     result := DBMS_XDB.createFolder('/public/mysource');
  END IF;
  IF (NOT DBMS_XDB.existsResource('/public/mysource/schemas')) THEN
     result := DBMS_XDB.createFolder('/public/mysource/schemas');
  END IF;
  IF (NOT DBMS_XDB.existsResource('/public/mysource/schemas/poSource')) THEN
     result := DBMS_XDB.createFolder('/public/mysource/schemas/poSource');
  END IF;
  IF (NOT DBMS_XDB.existsResource('/public/mysource/schemas/poSource/xsd')) THEN
     result := DBMS_XDB.createFolder('/public/mysource/schemas/poSource/xsd');
  END IF;
  IF (NOT DBMS_XDB.existsResource('/public/mysource/schemas/poSource/xsl')) THEN
     result := DBMS_XDB.createFolder('/public/mysource/schemas/poSource/xsl');
  END IF;
END;
/
```

Accessing the Content of Documents using SQL

You can access the content of documents stored in Oracle XML DB Repository in several ways. The easiest way is to use `XDBURIType`. `XDBURIType` uses a URL to specify which resource to access. The URL passed to the `XDBURIType` is assumed to start at the root of the repository. Data type `XDBURIType` provides methods `getBLOB()`, `getCLOB()`, and `getXML()` to access the different kinds of content that can be associated with a resource.

Example 3–52 shows how to use `XDBURIType` to access the content of the text document:

Example 3–52 Accessing a Text Document in the Repository using XDBURITYPE

```
SELECT XDBURIType('/home/QUINE/NurseryRhyme.txt').getCLOB() FROM DUAL;

XDBURITYPE('/HOME/QUINE/NURSERYRHYME.TXT').GETCLOB()
----------------------------------------------------
Mary had a little lamb
Its fleece was white as snow
and everywhere that Mary went
that lamb was sure to go

1 row selected.
```

The contents of a document can also be accessed using the resource document. Example 3–53 shows how to access the content of a text document:

Example 3–53 Accessing Resource Content using RESOURCE_VIEW

```
SELECT CONTENT
  FROM RESOURCE_VIEW,
       XMLTable(XMLNamespaces(default 'http://xmlns.oracle.com/xdb/XDBResource.xsd'),
                '/Resource/Contents' PASSING RES
                COLUMNS content CLOB PATH 'text')
  WHERE equals_path(RES, '/home/QUINE/NurseryRhyme.txt') = 1;

CONTENT
-------
Mary had a little lamb
Its fleece was white as snow
and everywhere that Mary went
that lamb was sure to go

1 row selected.
```

The content of non-schema-based and schema-based XML documents can also be accessed through a resource. Example 3–54 shows how to use an XPath expression that includes nodes from a resource document and nodes from an XML document to access the contents of a `PurchaseOrder` document using the resource.

Example 3–54 Accessing XML Documents using Resource and Namespace Prefixes

```
SELECT des.description
  FROM RESOURCE_VIEW rv,
       XMLTable(XMLNAMESPACES ('http://xmlns.oracle.com/xdb/XDBResource.xsd' AS "r"),
                '/r:Resource/r:Contents/PurchaseOrder/LineItems/LineItem'
                PASSING rv.RES
                COLUMNS description VARCHAR2(256) PATH 'Description') des
  WHERE
    equals_path(rv.RES, '/home/QUINE/PurchaseOrders/2002/Mar/SBELL-2002100912333601PDT.xml') = 1;

DES.DESCRIPTION
---------------------------------
A Night to Remember
The Unbearable Lightness Of Being
The Wizard of Oz

3 rows selected.
```

In Example 3–54, the namespace prefix, `r` identifies which nodes in the XPath expression are members of the resource namespace. Namespace prefix `r` is defined

using the XMLNAMESPACES clause of SQL/XML function XMLTable. The namespace declaration is needed here because the purchase-order XML schema does not define a namespace, and it is not possible to apply a namespace prefix to nodes in the PurchaseOrder document.

> **See Also:** Chapter 5, "Using XQuery with Oracle XML DB" for more information about the XMLNAMESPACES clause of XMLTable

Accessing the Content of XML Schema-Based Documents

The content of a schema-based XML document can be accessed in two ways.

- In the same manner as for non-schema-based XML documents, by using the resource document. This lets RESOURCE_VIEW be used to query different types of schema-based XML documents with a single SQL statement.
- As a row in the default table that was defined when the XML schema was registered with Oracle XML DB.

Accessing Resource Content using Element XMLRef in Joins

The XMLRef element in the resource document provides the join key required when a SQL statement needs to access or update metadata and content as part of a single operation.

The following queries use joins based on the value of element XMLRef to access resource content.

Example 3–55 locates a row in the defaultTable based on a path in Oracle XML DB Repository. SQL function ref locates the target row in the default table, based on the value of the XMLRef element in the resource document, RES.

Example 3–55 Querying Repository Resource Data using SQL Function REF and Element XMLRef

```
SELECT des.description
  FROM RESOURCE_VIEW rv,
       purchaseorder p,
       XMLTable('$p/PurchaseOrder/LineItems/LineItem' PASSING p.OBJECT_VALUE AS "p"
                COLUMNS description VARCHAR2(256) PATH 'Description') des
  WHERE equals_path(rv.RES, '/home/QUINE/PurchaseOrders/2002/Mar/SBELL-2002100912333601PDT.xml')
        = 1
    AND ref(p) = XMLCast(XMLQuery('declare default element namespace
                                   "http://xmlns.oracle.com/xdb/XDBResource.xsd"; (: :)
                                   fn:data(/Resource/XMLRef)' PASSING rv.RES RETURNING CONTENT)
                         AS REF XMLType);

DES.DESCRIPTION
--------------------------------
A Night to Remember
The Unbearable Lightness Of Being
The Wizard of Oz

3 rows selected.
```

Example 3–56 shows how to select fragments from XML documents based on metadata, path, and content. The query returns the value of element Reference for documents under /home/QUINE/PurchaseOrders/2002/Mar that contain orders for part number 715515009058.

Example 3–56 Selecting XML Document Fragments Based on Metadata, Path, and Content

```
SELECT XMLCast(XMLQuery('$p/PurchaseOrder/Reference'
                        PASSING po.OBJECT_VALUE AS "p" RETURNING CONTENT)
               AS VARCHAR2(30))
  FROM RESOURCE_VIEW rv, purchaseorder po
  WHERE under_path(rv.RES, '/home/QUINE/PurchaseOrders/2002/Mar') = 1
    AND ref(po) = XMLCast(
                    XMLQuery('declare default element namespace
                              "http://xmlns.oracle.com/xdb/XDBResource.xsd"; (: :)
                              fn:data(/Resource/XMLRef)'
                             PASSING rv.RES RETURNING CONTENT)
                          AS REF XMLType)
    AND XMLExists('$p/PurchaseOrder/LineItems/LineItem/Part[@Id="715515009058"]'
                  PASSING po.OBJECT_VALUE AS "p");

XMLCAST(XMLQUERY('$P/PURCHASEO
------------------------------
CJOHNSON-20021009123335851PDT
LSMITH-2002100912333661PDT
SBELL-2002100912333601PDT

3 rows selected.
```

In general, when accessing the content of schema-based XML documents, joining `RESOURCE_VIEW` or `PATH_VIEW` with the default table is more efficient than using `RESOURCE_VIEW` or `PATH_VIEW` on its own. An explicit join between the resource document and the default table tells Oracle XML DB that the SQL statement works on only one type of XML document. XPath rewrite can thus be used to optimize operations on the default table and the resource.

Updating the Content of Documents Stored in the Repository

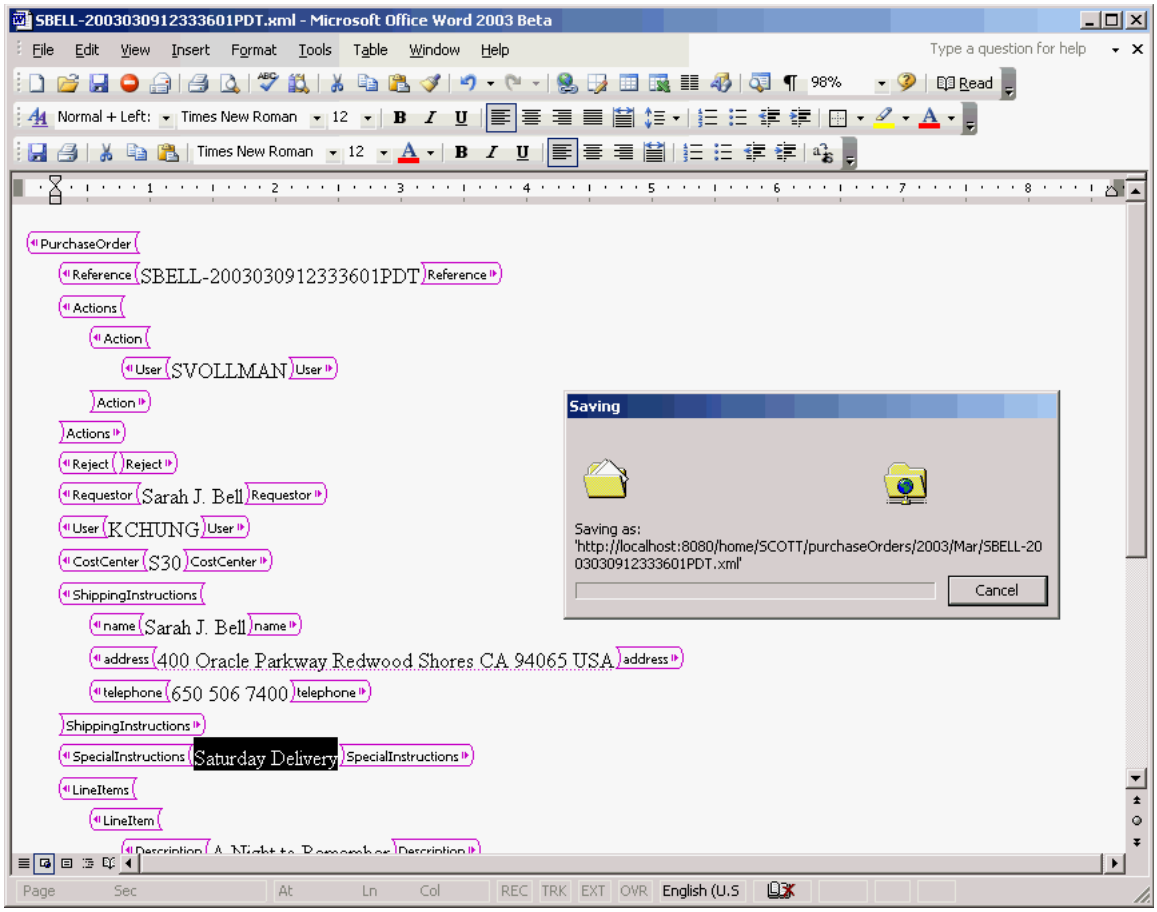
You can update the content of documents stored in Oracle XML DB Repository using protocols or SQL.

Updating Repository Content using Protocols

The most popular content authoring tools support HTTP, FTP, and WebDAV protocols. These tools can use a URL and the HTTP verb `get` to access the content of a document, and the HTTP verb `put` to save the contents of a document. Hence, given the appropriate access permissions, a simple URL is all you need to access and edit content stored in Oracle XML DB Repository.

Figure 3–6 shows how, with the WebDAV support included in Microsoft Word, you can use Microsoft Word to update and edit a document stored in Oracle XML DB Repository.

Figure 3–6 Updating and Editing Content Stored in Oracle XML DB using Microsoft Word



When an editing application such as Microsoft Word updates an XML document that is stored in Oracle XML DB, the database receives an input stream containing the new content of the document. Unfortunately, applications such as Word do not provide Oracle XML DB with any way of identifying which changes have taken place in the document.Partial updates are thus impossible. It is necessary to parse the entire document again, replacing all of the objects derived from the original document with objects derived from the new content.

Updating Repository Content using SQL

SQL functions such as `updateXML` can be used to update the content of any document stored in Oracle XML DB Repository. The content of the document can be modified by updating the resource document or by updating the default table that holds the content of the document.

Example 3–57 shows how to update the contents of a simple text document using a SQL `UPDATE` statement and SQL function `updateXML` on the resource document. An XPath expression is passed to `updateXML` as the target of the update operation, identifying the text node belonging to element `/Resource/Contents/text`.

Example 3–57 Updating a Document using UPDATE and UPDATEXML on the Resource

```
DECLARE
  file        BFILE;
  contents    CLOB;
  dest_offset NUMBER := 1;
```

```
  src_offset   NUMBER := 1;
  lang_context NUMBER := 0;
  conv_warning NUMBER := 0;
BEGIN
  file := bfilename('XMLDIR', 'tdadxdb-03-02.txt');
  DBMS_LOB.createTemporary(contents, true, DBMS_LOB.SESSION);
  DBMS_LOB.fileopen(file, DBMS_LOB.file_readonly);
  DBMS_LOB.loadClobfromFile(contents,
                            file,
                            DBMS_LOB.getLength(file),
                            dest_offset,
                            src_offset,
                            nls_charset_id('AL32UTF8'),
                            lang_context,
                            conv_warning);
  DBMS_LOB.fileclose(file);
  UPDATE RESOURCE_VIEW
    SET res = updateXML(res,
                        '/Resource/Contents/text/text()',
                        contents,
                        'xmlns="http://xmlns.oracle.com/xdb/XDBResource.xsd"')
      WHERE equals_path(res, '/home/QUINE/NurseryRhyme.txt') = 1;
  DBMS_LOB.freeTemporary(contents);
END;
/
```

This technique for updating the content of a document by updating the associated resource has the advantage that it can be used to update any kind of document stored in Oracle XML DB Repository.

Example 3–58 shows how to update a node in an XML document by performing an update on the resource document. Here, SQL function `updateXML` changes the value of the text node associated with element `User`.

Example 3–58 Updating a Node using UPDATE and UPDATEXML

```
UPDATE RESOURCE_VIEW
  SET res = updateXML(res,
                      '/r:Resource/r:Contents/PurchaseOrder/User/text()',
                      'SKING',
                      'xmlns:r="http://xmlns.oracle.com/xdb/XDBResource.xsd"')
  WHERE equals_path(
          res,
          '/home/QUINE/PurchaseOrders/2002/Mar/SBELL-2002100912333601PDT.xml')
        = 1;

1 row updated.

SELECT XMLCast(XMLQuery(
                 'declare namespace ns="http://xmlns.oracle.com/xdb/XDBResource.xsd"; (: :)
                  $r/ns:Resource/ns:Contents/PurchaseOrder/User/text()'
                 PASSING RES AS "r" RETURNING CONTENT)
               AS VARCHAR2(32))
  FROM RESOURCE_VIEW
  WHERE equals_path(RES,
                    '/home/QUINE/PurchaseOrders/2002/Mar/SBELL-2002100912333601PDT.xml')
        = 1;

XMLCAST(XMLQUERY('DECLARENAMESPA
--------------------------------
```

```
SKING

1 row selected.
```

Updating XML Schema-Based Documents in the Repository

You can update XML schema-based XML documents by performing the update operation directly on the default table that is used to manage the content of the document. If the document must be located by a `WHERE` clause that includes a path or conditions based on metadata, then the `UPDATE` statement must use a join between the resource and the default table.

In general, when updating the contents of XML schema-based XML documents, joining the `RESOURCE_VIEW` or `PATH_VIEW` with the default table is more efficient than using the `RESOURCE_VIEW` or `PATH_VIEW` on its own. The explicit join between the resource document and the default table tells Oracle XML DB that the SQL statement works on only one type of XML document. This lets a partial update be used on the default table and resource.

In Example 3–59, SQL function `updateXML` operates on the default table, with the target row identified by a path. The row to be updated is identified by a `REF`. The `REF` is identified by a repository path using SQL function `equals_path`. This limits the update to the row corresponding to the resource identified by the specified path.

Example 3–59 Updating XML Schema-Based Documents in the Repository

```
UPDATE purchaseorder p
  SET p.OBJECT_VALUE = updateXML(p.OBJECT_VALUE, '/PurchaseOrder/User/text()', 'SBELL')
    WHERE ref(p) =
      (SELECT XMLCast(XMLQuery('declare default element namespace
                               "http://xmlns.oracle.com/xdb/XDBResource.xsd"; (: :)
                               fn:data(/Resource/XMLRef)' PASSING rv.RES RETURNING CONTENT)
                       AS REF XMLType)
         FROM RESOURCE_VIEW rv
         WHERE equals_path(rv.RES,
                           '/home/QUINE/PurchaseOrders/2002/Mar/SBELL-2002100912333601PDT.xml')
               = 1);

SELECT XMLCast(XMLQuery('$p/PurchaseOrder/User/text()'
                        PASSING p.OBJECT_VALUE AS "p" RETURNING CONTENT)
               AS VARCHAR2(32))
  FROM purchaseorder p, RESOURCE_VIEW rv
  WHERE ref(p) = XMLCast(XMLQuery('declare default element namespace
                               "http://xmlns.oracle.com/xdb/XDBResource.xsd"; (: :)
                               fn:data(/Resource/XMLRef)' PASSING rv.RES RETURNING CONTENT)
                       AS REF XMLType)
    AND equals_path(rv.RES, '/home/QUINE/PurchaseOrders/2002/Mar/SBELL-2002100912333601PDT.xml')
        = 1;

XMLCAST(XMLQUERY('$P/PURCHASEO
------------------------------
SBELL
```

Controlling Access to Repository Data

You can control access to the resources in Oracle XML DB Repository by using access control lists (ACLs). An ACL is a list of access control entries (ACEs), each of which grants or denies a set of privileges to a specific principal. The principal can be a database user, a database role, an LDAP user, an LDAP group or the special principal `DAV::owner`, which refers to the owner of the resource. Each resource in the

repository is protected by an ACL. The ACL determines what privileges, such as `read-properties` and `update`, a user has on the resource. Each repository operation includes a check of the ACL to determine if the current user is allowed to perform the operation.

By default, a new resource inherits the ACL of its parent folder. But you can set the ACL of a resource using PL/SQL procedure `DBMS_XDB.setACL`. For more details on Oracle XML DB resource security, see Chapter 27, "Repository Access Control".

In the following example, the current user is `QUINE`. The query gives the number of resources in the folder `/public`. Assume that there are only two resources in this folder: `f1` and `f2`. Also assume that the ACL on `f1` grants the `read-properties` privilege to `QUINE` while the ACL on `f2` does not grant `QUINE` any privileges. A user needs the `read-properties` privilege on a resource for it to be visible to the user. The result of the query is `1`, because only `f1` is visible to `QUINE`.

```
SELECT count(*) FROM RESOURCE_VIEW r WHERE under_path(r.res, '/public') = 1;

COUNT(*)
--------
       1
```

Oracle XML DB Transactional Semantics

When working from SQL, normal transactional behavior is enforced. Multiple calls to SQL functions such as `updateXML` can be used within a single logical unit of work. Changes made through functions like `updateXML` are not visible to other database users until the transaction is committed. At any point, `ROLLBACK` can be used to back out the set of changes made since the last commit.

Querying Metadata and the Folder Hierarchy

In Oracle XML DB, the system-defined metadata for each resource is preserved as an XML document. The structure of these resource documents is defined by XML schema `XDBResource.xsd`. This schema is registered as a global XML schema at URL `http://xmlns.oracle.com/xdb/XDBResource.xsd`.

Oracle XML DB gives you access to metadata and information about the folder hierarchy using two public views, `RESOURCE_VIEW` and `PATH_VIEW`.

RESOURCE_VIEW and PATH_VIEW

`RESOURCE_VIEW` contains one entry for each file or folder stored in Oracle XML DB Repository. Column `RES` of `RESOURCE_VIEW` contains the resource, an XML document that manages the metadata properties associated with the resource content. Column `ANY_PATH` contains a valid URL that the current user can pass to `XDBURIType` to access the resource content. If this content is not binary data, then the resource itself also contains the content.

Oracle XML DB supports the concept of **linking**. Linking makes it possible to define multiple paths to a given document. A separate XML document, called the **link-properties document**, maintains metadata properties that are specific to the path, rather than to the resource. Whenever a resource is created, an initial link is also created.

`PATH_VIEW` exposes the link-properties documents. There is one entry in `PATH_VIEW` for each possible path to a document. Column `RES` of `PATH_VIEW` contains the resource document pointed to by this link. Column `PATH` contains the path that the

link lets you use to access the resource. Column LINK contains the link-properties document (metadata) for this PATH.

Example 3–60 shows the description of public views RESOURCE_VIEW and PATH_VIEW:

Example 3–60 Viewing RESOURCE_VIEW and PATH_VIEW Structures

```
DESCRIBE RESOURCE_VIEW

Name      Null?    Type
-----------------------------------------------------------
RES                SYS.XMLTYPE(XMLSchema
                              "http://xmlns.oracle.com/xdb/XDBResource.xsd"
                              Element
                              "Resource")
ANY_PATH           VARCHAR2(4000)
RESID              RAW(16)


DESCRIBE PATH_VIEW

Name      Null?    Type
-----------------------------------------------------------
PATH               VARCHAR2(1024)
RES                SYS.XMLTYPE(XMLSchema
                              "http://xmlns.oracle.com/xdb/XDBResource.xsd"
                              Element
                              "Resource")
LINK               SYS.XMLTYPE
RESID              RAW(16)
```

See Also:

- Chapter 25, "Accessing the Repository using RESOURCE_VIEW and PATH_VIEW"
- *Oracle Database Reference* for more information about view PATH_VIEW
- *Oracle Database Reference* for more information about view RESOURCE_VIEW

Querying Resources in RESOURCE_VIEW and PATH_VIEW

Oracle XML DB provides two SQL functions, equals_path and under_path, that can be used to perform **folder-restricted queries**. Such queries limit SQL statements that operate on the RESOURCE_VIEW or PATH_VIEW to documents that are at a particular location in Oracle XML DB folder hierarchy. Function equals_path restricts the statement to a single document identified by the specified path. Function under_path restricts the statement to those documents that exist beneath a certain point in the hierarchy.

The following examples demonstrate simple folder-restricted queries against resource documents stored in RESOURCE_VIEW and PATH_VIEW.

The query in Example 3–61 uses SQL function equals_path and RESOURCE_VIEW to access the resource created in Example 3–60.

Example 3–61 Accessing Resources using EQUALS_PATH and RESOURCE_VIEW

```
SELECT XMLSerialize(DOCUMENT r.res AS CLOB)
  FROM RESOURCE_VIEW r
  WHERE equals_path(res, '/home/QUINE/NurseryRhyme.txt') = 1;

XMLSERIALIZE(DOCUMENTR.RESASCLOB)
--------------------------------------------------------------------------------
<Resource xmlns="http://xmlns.oracle.com/xdb/XDBResource.xsd"
          Hidden="false"
          Invalid="false"
          Container="false"
          CustomRslv="false"
          VersionHistory="false"
          StickyRef="true">
  <CreationDate>2005-06-13T13:19:20.566623</CreationDate>
  <ModificationDate>2005-06-13T13:19:22.997831</ModificationDate>
  <DisplayName>NurseryRhyme.txt</DisplayName>
  <Language>en-US</Language>
  <CharacterSet>UTF-8</CharacterSet>
  <ContentType>text/plain</ContentType>
  <RefCount>1</RefCount>
  <ACL>
    <acl description=
         "Private:All privileges to OWNER only and not accessible to others"
         xmlns="http://xmlns.oracle.com/xdb/acl.xsd" xmlns:dav="DAV:"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.oracle.com/xdb/acl.xsd
         http://xmlns.oracle.com/xdb/acl.xsd"
         shared="true">
      <ace>
        <grant>true</grant>
        <principal>dav:owner</principal>
        <privilege>
          <all/>
        </privilege>
      </ace>
    </acl>
  </ACL>
  <Owner>QUINE</Owner>
  <Creator>QUINE</Creator>
  <LastModifier>QUINE</LastModifier>
  <SchemaElement>http://xmlns.oracle.com/xdb/XDBSchema.xsd#text</SchemaElement>
  <Contents>
    <text>Hickory Dickory Dock
The Mouse ran up the clock
The clock struck one
The Mouse ran down
Hickory Dickory Dock
    </text>
  </Contents>
</Resource>

1 row selected.
```

As Example 3–61 shows, a resource document is an XML document that captures the set of metadata defined by the DAV standard. The metadata includes information such as `CreationDate`, `Creator`, `Owner`, `ModificationDate`, and `DisplayName`. The content of the resource document can be queried and updated just like any other XML document, using SQL/XML access and query functions.

The query in Example 3–62 finds a path to each of the XSL style sheets stored in Oracle XML DB Repository. It performs a search based on the `DisplayName` ending in `.xsl`.

Example 3–62 Determining the Path to XSL Style Sheets Stored in the Repository

```
SELECT ANY_PATH FROM RESOURCE_VIEW
  WHERE XMLCast(XMLQuery(
                  'declare namespace ns="http://xmlns.oracle.com/xdb/XDBResource.xsd"; (: :)
                   $r/ns:Resource/ns:DisplayName'
                  PASSING RES AS "r" RETURNING CONTENT)
                AS VARCHAR2(100))
        LIKE '%.xsl';

ANY_PATH
-------------------------------------------
/source/schemas/poSource/xsl/empdept.xsl
/source/schemas/poSource/xsl/purchaseOrder.xsl

2 rows selected.
```

The query in Example 3–63 counts the number of resources (files and folders) under the path `/home/QUINE/PurchaseOrders`. Using `RESOURCE_VIEW` rather than `PATH_VIEW` ensures that any resources that are the target of multiple links are only counted once. SQL function `under_path` restricts the result set to documents that can be accessed using a path that starts from `/home/QUINE/PurchaseOrders`.

Example 3–63 Counting Resources Under a Path

```
SELECT count(*)
   FROM RESOURCE_VIEW
   WHERE under_path(RES, '/home/QUINE/PurchaseOrders') = 1;

  COUNT(*)
----------
       145

1 row selected.
```

The query in Example 3–64 lists the contents of the folder identified by path `/home/QUINE/PurchaseOrders/2002/Apr`. This is effectively a directory listing of the folder.

Example 3–64 Listing the Folder Contents in a Path

```
SELECT PATH
  FROM PATH_VIEW
  WHERE under_path(RES, '/home/QUINE/PurchaseOrders/2002/Apr') = 1;

PATH
-----------------------------------------------------------------------
/home/QUINE/PurchaseOrders/2002/Apr/AMCEWEN-20021009123336171PDT.xml
/home/QUINE/PurchaseOrders/2002/Apr/AMCEWEN-20021009123336271PDT.xml
/home/QUINE/PurchaseOrders/2002/Apr/EABEL-20021009123336251PDT.xml
/home/QUINE/PurchaseOrders/2002/Apr/PTUCKER-20021009123336191PDT.xml
/home/QUINE/PurchaseOrders/2002/Apr/PTUCKER-20021009123336291PDT.xml
/home/QUINE/PurchaseOrders/2002/Apr/SBELL-20021009123336231PDT.xml
/home/QUINE/PurchaseOrders/2002/Apr/SBELL-20021009123336331PDT.xml
/home/QUINE/PurchaseOrders/2002/Apr/SKING-20021009123336321PDT.xml
/home/QUINE/PurchaseOrders/2002/Apr/SMCCAIN-20021009123336151PDT.xml
/home/QUINE/PurchaseOrders/2002/Apr/SMCCAIN-20021009123336341PDT.xml
```

```
/home/QUINE/PurchaseOrders/2002/Apr/VJONES-20021009123336301PDT.xml

11 rows selected.
```

The query in Example 3–65 lists the set of links contained in the folder identified by the path `/home/QUINE/PurchaseOrders/2002/Apr` where the `DisplayName` element in the associated resource starts with `S`.

Example 3–65 Listing the Links Contained in a Folder

```
SELECT PATH
  FROM PATH_VIEW
  WHERE XMLCast(XMLQuery(
                  'declare namespace ns="http://xmlns.oracle.com/xdb/XDBResource.xsd"; (: :)
                   $r/ns:Resource/ns:DisplayName'
                  PASSING RES AS "r" RETURNING CONTENT)
                AS VARCHAR2(100))
        LIKE 'S%'
    AND under_path(RES, '/home/QUINE/PurchaseOrders/2002/Apr') = 1;

PATH
--------------------------------------------------------------------
/home/QUINE/PurchaseOrders/2002/Apr/SBELL-20021009123336231PDT.xml
/home/QUINE/PurchaseOrders/2002/Apr/SBELL-20021009123336331PDT.xml
/home/QUINE/PurchaseOrders/2002/Apr/SKING-20021009123336321PDT.xml
/home/QUINE/PurchaseOrders/2002/Apr/SMCCAIN-20021009123336151PDT.xml
/home/QUINE/PurchaseOrders/2002/Apr/SMCCAIN-20021009123336341PDT.xml

5 rows selected.
```

The query in Example 3–66 finds a path to each resource in Oracle XML DB Repository that contains a `PurchaseOrder` document. The documents are identified based on the metadata property `SchemaElement` that identifies the XML schema URL and global element for schema-based XML data stored in the repository.

Example 3–66 Finding Paths to Resources that Contain Purchase-Order XML Documents

```
SELECT ANY_PATH
  FROM RESOURCE_VIEW
 WHERE XMLExists(
         'declare namespace ns="http://xmlns.oracle.com/xdb/XDBResource.xsd"; (: :)
          $r/ns:Resource[ns:SchemaElement=
          "http://localhost:8080/source/schemas/poSource/xsd/purchaseOrder.xsd#PurchaseOrder"]'
         PASSING RES AS "r");
```

The query returns the following paths, each of which contains a `PurchaseOrder` document:

```
ANY_PATH
--------------------------------------------------------------------
/home/QUINE/PurchaseOrders/2002/Apr/AMCEWEN-20021009123336171PDT.xml
/home/QUINE/PurchaseOrders/2002/Apr/AMCEWEN-20021009123336271PDT.xml
/home/QUINE/PurchaseOrders/2002/Apr/EABEL-20021009123336251PDT.xml
/home/QUINE/PurchaseOrders/2002/Apr/PTUCKER-20021009123336191PDT.xml

...

132 rows selected.
```

Oracle XML DB Hierarchical Repository Index

In a conventional relational database, path-based access and folder-restricted queries are implemented using `CONNECT BY` operations. Such queries are expensive, so path-based access and folder-restricted queries would become inefficient as the number of documents and depth of the folder hierarchy increase.

To address this issue, Oracle XML DB introduces a new index type, the **hierarchical repository index**. This lets the database resolve folder-restricted queries without relying on a `CONNECT BY` operation. Because of this, Oracle XML DB can execute path-based and folder-restricted queries efficiently. The hierarchical repository index is implemented as an Oracle domain index. This is the same technique used to add Oracle Text indexing support and many other advanced index types to the database.

Example 3–67 shows the execution plan output generated for a folder-restricted query. As shown, the hierarchical repository index `XDBHI_IDX` is used to resolve the query.

Example 3–67 Execution Plan Output for a Folder-Restricted Query

```
SELECT PATH
  FROM PATH_VIEW
  WHERE XMLCast(
          XMLQuery(
            'declare namespace ns="http://xmlns.oracle.com/xdb/XDBResource.xsd"; (: :)
             $r/ns:Resource/ns:DisplayName'
            PASSING RES AS "r" RETURNING CONTENT)
          AS VARCHAR2(100))
        LIKE 'S%'
    AND under_path(RES, '/home/QUINE/PurchaseOrders/2002/Apr') = 1;

PLAN_TABLE_OUTPUT
-----------------------------------------------------------------------------------------------------
Plan hash value: 2568289845

-----------------------------------------------------------------------------------------------------
| Id  | Operation                            | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                     |               |    17 |  3111 |    34   (6)| 00:00:01 |
|   1 |  NESTED LOOPS                        |               |    17 |  3111 |    34   (6)| 00:00:01 |
|   2 |   NESTED LOOPS                       |               |    17 |  2822 |    34   (6)| 00:00:01 |
|   3 |    NESTED LOOPS                      |               |   466 | 63842 |    34   (6)| 00:00:01 |
|*  4 |     TABLE ACCESS BY INDEX ROWID      | XDB$RESOURCE  |     1 |   135 |     3   (0)| 00:00:01 |
|*  5 |      DOMAIN INDEX                    | XDBHI_IDX     |       |       |            |          |
|   6 |     COLLECTION ITERATOR PICKLER FETCH|               |       |       |            |          |
|*  7 |    INDEX UNIQUE SCAN                 | XDB_PK_H_LINK |     1 |    28 |     0   (0)| 00:00:01 |
|*  8 |   INDEX UNIQUE SCAN                  | SYS_C003900   |     1 |    17 |     0   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   4 - filter(CAST("P"."SYS_NC00011$" AS VARCHAR2(100)) LIKE 'S%')
   5 - access("XDB"."UNDER_PATH"(SYS_MAKEXML('8758D485E6004793E034080020B242C6',734,"XMLEXTRA"
              ,"XMLDATA"),'/home/QUINE/PurchaseOrders/2002/Apr',9999)=1)
   7 - access("H"."PARENT_OID"=SYS_OP_ATG(VALUE(KOKBF$),3,4,2) AND
              "H"."NAME"=SYS_OP_ATG(VALUE(KOKBF$),2,3,2))
   8 - access("R2"."SYS_NC_OID$"=SYS_OP_ATG(VALUE(KOKBF$),3,4,2))

25 rows selected.
```

How Documents are Stored in the Repository

Oracle XML DB provides special handling for XML documents. The rules for storing the contents of schema-based XML document are defined by the XML schema. The content of the document is stored in the default table associated with the global element definition.

Oracle XML DB Repository also stores files that do not contain XML data, such as JPEG images or Word documents. The XML schema for each resource defines which elements are allowed, and specifies whether the content of these files is to be stored as `BLOB` or `CLOB` instances. The content of a non-schema-based XML document is stored as a `CLOB` instance in the repository.

There is one resource and one link-properties document for each file or folder in the repository. If there are multiple access paths to a given document, there is a link-properties document for each possible link. Both the resource document and the link-properties are stored as XML documents. All these documents are stored in tables in the repository.

When an XML file is loaded into the repository, the following sequence of events takes place:

1. Oracle XML DB examines the root element of the XML document to see if it is associated with a known (registered) XML schema. This involves looking to see if the document includes a namespace declaration for the `XMLSchema-instance` namespace, and then looking for a `schemaLocation` or `noNamespaceSchemaLocation` attribute that identifies which XML schema the document is associated with.
2. If the document is based on a known XML schema, then the metadata for the XML schema is loaded from the XML schema cache.
3. The XML document is parsed and decomposed into a set of SQL objects derived from the XML schema.
4. The SQL objects created from the XML file are stored in the default table defined when the XML schema was registered with the database.
5. A resource document is created for each document processed. This lets the content of the document be accessed using the repository. The resource document for an XML schema-based `XMLType` instance includes an `XMLRef` element. This element contains a `REF` of `XMLType` that can be used to locate the row in the default table containing the content associated with the resource.

Viewing Relational Data as XML From a Browser

The HTTP server built into Oracle XML DB makes it possible to use a browser to access any document stored in Oracle XML DB Repository. Since a resource can include a `REF` to a row in an `XMLType` table or view, it is possible to use a path to access this type of content.

Accessing a Table or View from a Browser using DBUri SERVLET

Oracle XML DB includes the DBUri servlet, which makes it possible to access the content of any table or view directly from a browser. DBUri servlet uses the facilities of the `DBURIType` to generate a simple XML document from the contents of the table. The servlet is C language-based and installed in the Oracle XML DB HTTP server. By default, the servlet is installed under the virtual directory `/oradb`.

The URL passed to the DBUri Servlet is an extension of the URL passed to the DBURIType. The URL is extended with the address and port number of the Oracle XML DB HTTP server and the virtual root that directs HTTP(S) requests to the DBUri servlet. The default configuration for this is /oradb.

The URL http://localhost:8080/oradb/HR/DEPARTMENTS would thus return an XML document containing the contents of the DEPARTMENTS table in the HR database schema. This assumes that the Oracle XML DB HTTP server is running on port 8080, the virtual root for the DBUri servlet is /oradb, and that the user making the request has access to the HR database schema.

DBUri servlet accepts parameters that allow you to specify the name of the ROW tag and MIME-type of the document that is returned to the client.

Content in XMLType table or view can also be accessed through the DBUri servlet. When the URL passed to the DBUri servlet references an XMLType table or XMLType view the URL can be extended with an XPath expression that can determine which documents in the table or row are returned. The XPath expression appended to the URL can reference any node in the document.

XML generated by DBUri servlet can be transformed using the XSLT processor built into Oracle XML DB. This lets XML that is generated by DBUri servlet be presented in a more legible format such as HTML.

> **See Also:** "DBUriServlet" on page 20-26

Style sheet processing is initiated by specifying a transform parameter as part of the URL passed to DBUri servlet. The style sheet is specified using a URI that references the location of the style sheet within database. The URI can either be a DBURIType value that identifies a XMLType column in a table or view, or a path to a document stored in Oracle XML DB Repository. The style sheet is applied directly to the generated XML before it is returned to the client. When using DBUri servlet for XSLT processing, it is good practice to use the contenttype parameter to explicitly specify the MIME type of the generated output.

If the XML document being transformed is stored as an XML schema-based XMLType instance, then Oracle XML DB can reduce the overhead associated with XSL transformation by leveraging the capabilities of the lazily loaded virtual DOM.

The root of the URL is /oradb, so the URL is passed to the DBUri servlet that accesses the purchaseorder table in the SCOTT database schema, rather than as a resource in Oracle XML DB Repository. The URL includes an XPath expression that restricts the result set to those documents where node /PurchaseOrder/Reference/text() contains the value specified in the predicate. The contenttype parameter sets the MIME type of the generated document to text/xml.

XSL Transformation using DBUri Servlet

Figure 3–7 shows how an XSL transformation can be applied to XML content generated by the DBUri servlet. In this example the URL passed to the DBUri includes the transform parameter. This causes the DBUri servlet to use Oracle SQL function XMLtransform to apply the style sheet /home/SCOTT/xsl/purchaseOrder.xsl to the PurchaseOrder document identified by the main URL, before returning the document to the browser. This style sheet transforms the XML document to a more user-friendly HTML page. The URL also uses contentType parameter to specify that the MIME-type of the final document is text/html.

Figure 3–7 Database XSL Transformation of a PurchaseOrder using DBUri Servlet

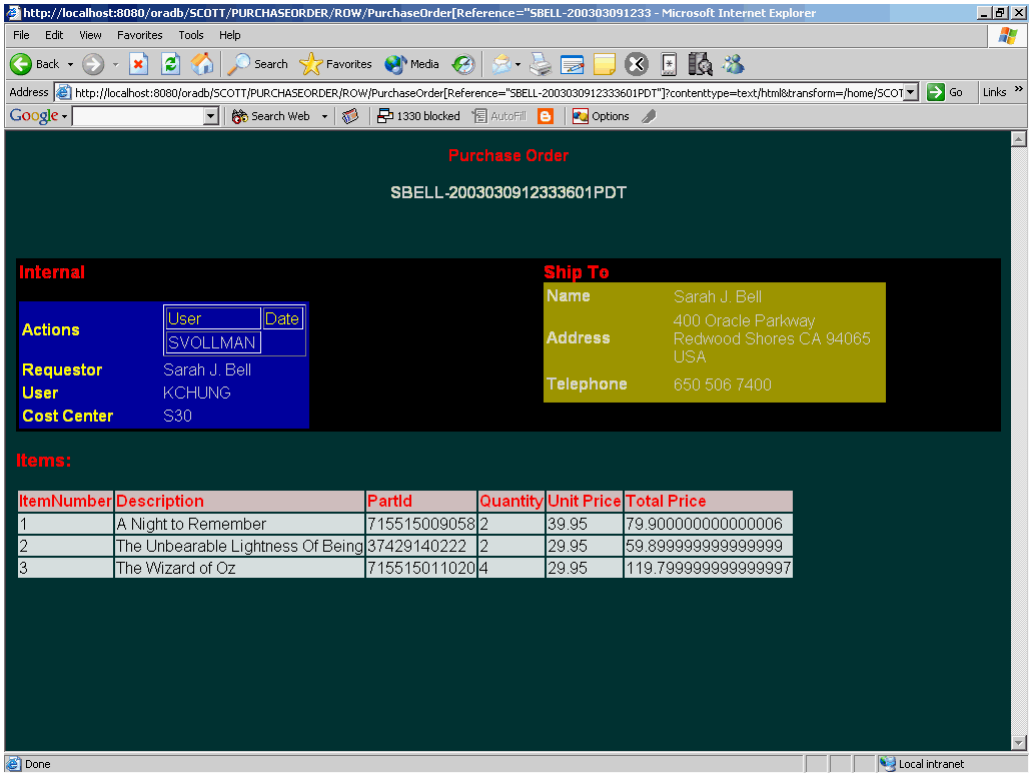
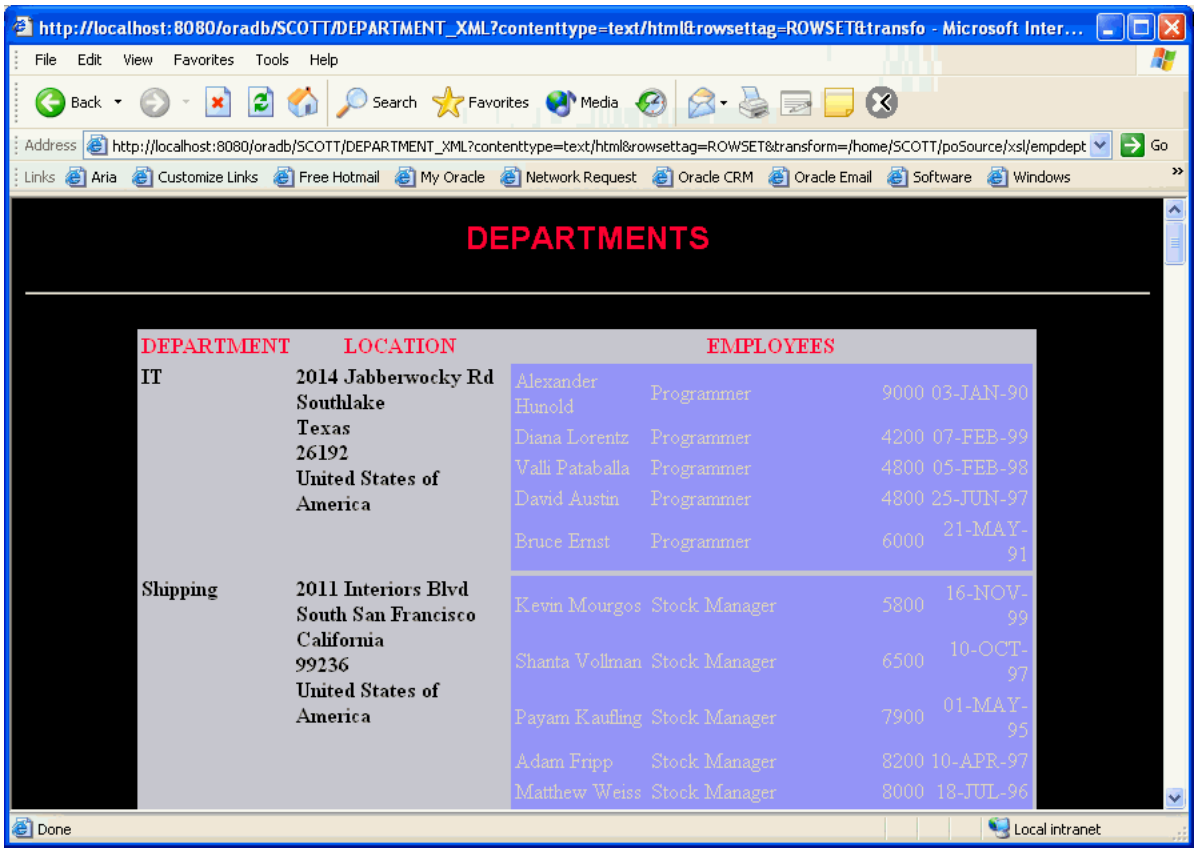


Figure 3–8 shows the `departments` table displayed as an HTML document. You need no code to achieve this, you only need an `XMLType` view, based on SQL/XML functions, an industry-standard XSL style sheet, and `DBUri` servlet.

Figure 3–8 Database XSL Transformation of Departments Table using DBUri Servlet



Part II

Storing and Retrieving XML Data in Oracle XML DB

Part II of this manual introduces you to ways you can store, retrieve, validate, and transform XML data using Oracle XML DB. It contains the following chapters:

- Chapter 4, "XMLType Operations"
- Chapter 5, "Using XQuery with Oracle XML DB"
- Chapter 6, "Indexing XMLType Data"
- Chapter 7, "XML Schema Storage and Query: Basic"
- Chapter 8, "XPath Rewrite for Structured Storage"
- Chapter 9, "XML Schema Storage and Query: Advanced"
- Chapter 10, "XML Schema Evolution"
- Chapter 11, "Transforming and Validating XMLType Data"
- Chapter 12, "Full-Text Search Over XML Data"

4

XMLType Operations

This chapter describes `XMLType` operations for XML applications (XML schema-based and non-schema-based). It includes guidelines for creating, manipulating, updating, and querying `XMLType` columns and tables.

This chapter contains these topics:

- Selecting and Querying XML Data
- Updating XML Data

See Also:

- Chapter 3, "Using Oracle XML DB" for `XMLType` storage recommendations
- Chapter 7, "XML Schema Storage and Query: Basic" for how to work with XML schema-based `XMLType` tables and columns

Selecting and Querying XML Data

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

- Select `XMLType` columns using SQL, PL/SQL, or Java.
- Use the XQuery language. See "Using XQuery with XMLType Data" on page 5-23.
- Query `XMLType` columns directly or using SQL/XML functions such as `XMLQuery`.
- Use Oracle Text operators for full-text search. See Chapter 6, "Indexing XMLType Data" and Chapter 12, "Full-Text Search Over XML Data".

Searching XML Documents using XPath Expressions

The XPath language is a W3C Recommendation for navigating XML documents. It is a subset of the XQuery language, in the sense that an XPath expression is also an XQuery expression.

XPath models an XML document as a tree of nodes. It provides a rich set of operations that walk this tree and apply predicates and node-test functions. Applying an XPath expression to an XML document can result in a set of nodes. For example, the expression `/PO/PONO` selects all `PONO` child elements under the `PO` root element of the document.

> **Note:** Oracle SQL functions and `XMLType` methods respect the W3C XPath recommendation, which states that if an XPath expression targets *no nodes* when applied to XML data, then an empty sequence must be returned. An error must *not* be raised in this case.
>
> The specific semantics of an Oracle SQL function or `XMLType` method that applies an XPath expression to XML data determines what is returned. For example, SQL/XML function `XMLQuery` returns `NULL` if its XPath-expression argument targets no nodes, and the updating SQL functions, such as `deleteXML`, return the input XML data unchanged. An error is never raised if no nodes are targeted, but updating SQL functions can raise an error if an XPath-expression argument targets inappropriate nodes, such as attribute nodes or text nodes.

Table 4–1 lists some common constructs used in XPath.

Table 4–1 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, `/PurchaseOrder/Reference` identifies the purchase-order name element `Reference`, a child of the root element.
`//`	Used to identify all descendants of the current node. For example, `PurchaseOrder//ShippingInstructions` matches any `ShippingInstructions` element under the `PurchaseOrder` 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` selects the shipping address element of all purchase orders whose purchase-order number is `20` and whose purchase-order name is `PO_2`. Brackets are also used to denote a position (index). For example, `/PO/PONO[2]` identifies the second purchase-order number element under the `PO` root element.
Functions	XPath and XQuery support a set of built-in functions such as `substring`, `round`, and `not`. In addition, these languages provide for extension functions through the use of namespaces. Oracle XQuery extension functions use the namespace prefix `ora`, for namespace `http://xmlns.oracle.com/xdb`. See "Oracle XQuery Extension Functions" on page 5-11.

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

You can select `XMLType` data using PL/SQL, C, or Java. You can also use `XMLType` method `getNumberVal()` to retrieve XML data as a `NUMBER`.

Querying XMLType Data using SQL/XML Functions XMLExists and XMLCast

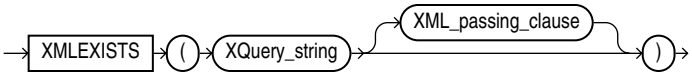
You can query `XMLType` data and extract portions of it using SQL/XML standard functions `XMLQuery`, `XMLTable`, `XMLExists`, and `XMLCast`.

See Chapter 5, "Using XQuery with Oracle XML DB" for more information about functions `XMLQuery` and `XMLTable`. Functions `XMLExists` and `XMLCast` are described in this section.

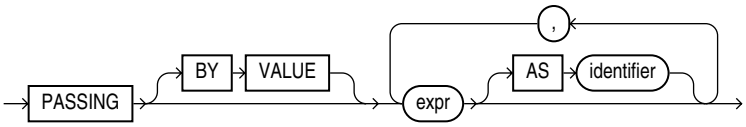
XMLEXISTS SQL/XML Function

Figure 4–1 describes the syntax for SQL/XML standard function `XMLExists`. This function checks whether a given XQuery expression returns a non-empty XQuery sequence. If so, the function returns `TRUE`. Otherwise, it returns `FALSE`.

Figure 4–1 XMLExists Syntax



XML_passing_clause ::=



- *`XQuery_string`* is a complete XQuery expression, possibly including a prolog, as a literal string. It can contain XQuery variables that you bind using the XQuery `PASSING` clause (*`XML_passing_clause`* in the syntax diagram). The predefined namespace prefixes recognized for SQL/XML function `XMLQuery` are also recognized in *`XQuery_string`*—see "Predefined Namespaces and Prefixes" on page 5-10.
- The *`XML_passing_clause`* is the keyword `PASSING` followed by one or more SQL expressions (*`expr`*) that each return an `XMLType` instance or an instance of a SQL scalar data type. All but possibly one of the expressions must each be followed by the keyword `AS` and an XQuery *`identifier`*. The result of evaluating each *`expr`* is bound to the corresponding *`identifier`* for the evaluation of *`XQuery_string`*. If there is an *`expr`* that is not followed by an `AS` clause, then the result of evaluating that *`expr`* is used as the *context* item for evaluating *`XQuery_string`*. Oracle XML DB supports only passing `BY VALUE`, not passing `BY REFERENCE`, so the clause `BY VALUE` is implicit and can be omitted.

If an XQuery expression such as `/PurchaseOrder/Reference` or `/PurchaseOrder/Reference/text()` targets a single node, then `XMLExists` returns `true` for that expression. If `XMLExists` is called with an XQuery expression that locates no nodes, then `XMLExists` returns `false`.

Function `XMLExists` can be used in queries, and it can be used to create function-based indexes to speed up evaluation of queries.

Note: Oracle XML DB limits the use of `XMLExists` to a SQL `WHERE` clause or `CASE` expression. If you need to use `XMLExists` in a `SELECT` list, then wrap it in a `CASE` expression:

```
CASE WHEN XMLExists(...) THEN 'TRUE' ELSE 'FALSE' END
```

Example 4–1 uses SQL/XML standard function `XMLExists` to select rows with `SpecialInstructions` set to `Expedite`.

Example 4–1 Finding a Node using SQL/XML Function XMLExists

```
SELECT OBJECT_VALUE
  FROM purchaseorder
  WHERE XMLExists('/PurchaseOrder[SpecialInstructions="Expedite"]'
                  PASSING OBJECT_VALUE);

OBJECT_VALUE
--------------------------------------------------------------------
<PurchaseOrder xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
<PurchaseOrder xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
<PurchaseOrder xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
<PurchaseOrder xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
<PurchaseOrder xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
<PurchaseOrder xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
<PurchaseOrder xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
<PurchaseOrder xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
<PurchaseOrder xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
<PurchaseOrder xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
<PurchaseOrder xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
<PurchaseOrder xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
<PurchaseOrder xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

13 rows selected.
```

You can create function-based indexes using SQL/XML function `XMLExists` to speed up the execution. You can also create an `XMLIndex` index to help speed up arbitrary XQuery searching.

Note: Prior to Oracle Database 11*g* Release 2, some users employed Oracle SQL function `existsNode` to do some of what can be done better using SQL/XML function `XMLExists`. Function `existsNode` is *deprecated* in Oracle Database 11*g* Release 2. The two functions differ in these important ways:

- Function `existsNode` returns `0` or `1`. Function `XMLExists` returns a Boolean value, `TRUE` or `FALSE`.
- You can use `existsNode` in a query `SELECT` list. You cannot use `XMLExists` directly in a `SELECT` list, but you can use `XMLExists` within a `CASE` expression in a `SELECT` list.

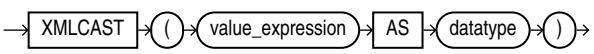
See Also:

- "Indexing XMLType Data Stored Object-Relationally" on page 6-6
- "XMLIndex" on page 6-7

XMLCAST SQL/XML Function

Figure 4–2 describes the syntax for SQL/XML standard function `XMLCast`.

Figure 4–2 XMLCast Syntax



SQL/XML standard function `XMLCast` casts its first argument to the scalar SQL data type specified by its second argument. The first argument is a SQL expression that is evaluated. Any of the following SQL data types can be used as the second argument:

- `NUMBER`
- `VARCHAR2`
- `CHAR`
- `CLOB`
- `BLOB`
- `REF XMLTYPE`
- any SQL date or time data type

> **Note:** Unlike the SQL/XML standard, Oracle XML DB limits the use of `XMLCast` to cast XML to a SQL scalar data type. Oracle XML DB does not support casting XML to XML or from a scalar SQL type to XML.

The result of evaluating the first `XMLCast` argument is an XML value. It is converted to the target SQL data type by using the XQuery atomization process and then casting the XQuery atomic values to the target data type. If this conversion fails, then an error is raised. If conversion succeeds, the result returned is an instance of the target data type.

The query in Example 4–2 extracts the scalar value of node `Reference`.

Example 4–2 Extracting the Scalar Value of an XML Fragment using XMLCAST

```
SELECT XMLCast(XMLQuery('/PurchaseOrder/Reference' PASSING OBJECT_VALUE
                                                   RETURNING CONTENT)
               AS VARCHAR2(100)) "REFERENCE"
  FROM purchaseorder
  WHERE XMLExists('/PurchaseOrder[SpecialInstructions="Expedite"]'
                  PASSING OBJECT_VALUE);

REFERENCE
----------------------------
AMCEWEN-20021009123336271PDT
SKING-20021009123336321PDT
AWALSH-20021009123337303PDT
JCHEN-20021009123337123PDT
AWALSH-20021009123336642PDT
SKING-20021009123336622PDT
SKING-20021009123336822PDT
AWALSH-20021009123336101PDT
WSMITH-20021009123336412PDT
AWALSH-20021009123337954PDT
SKING-20021009123338294PDT
WSMITH-20021009123338154PDT
TFOX-20021009123337463PDT

13 rows selected.
```

Note:

- Prior to Oracle Database 11*g* Release 2, some users employed Oracle SQL function `extractValue` to do some of what can be done better using SQL/XML functions `XMLQuery` and `XMLCast`. Function `extractValue` is *deprecated* in Oracle Database 11*g* Release 2.
- Function `extractValue` raises an error when its XPath expression argument matches multiple text nodes. `XMLCast` applied to an `XMLQuery` result returns the concatenation of the text nodes—it does *not* raise an error.

See Also:

- "Indexing XMLType Data Stored Object-Relationally" on page 6-6
- "XMLIndex" on page 6-7

Examples of Querying XML Data using SQL/XML Functions

The examples in this section illustrate ways you can use SQL to query XML data. Example 4–3 inserts two rows into table `purchaseorder`, then queries data in those rows using SQL/XML functions `XMLCast`, `XMLQuery`, and `XMLExists`.

Example 4–3 Querying XMLTYPE Data

```
INSERT INTO purchaseorder
  VALUES (XMLType(bfilename('XMLDIR', 'SMCCAIN-2002091213000000PDT.xml'),
                  nls_charset_id('AL32UTF8')));

INSERT INTO purchaseorder
  VALUES (XMLType(bfilename('XMLDIR', 'VJONES-20020916140000000PDT.xml'),
                  nls_charset_id('AL32UTF8')));

SELECT XMLCast(XMLQuery('$p/PurchaseOrder/Reference'
                        PASSING po.OBJECT_VALUE AS "p" RETURNING CONTENT)
               AS VARCHAR2(30)) reference,
       XMLCast(XMLQuery('$p/PurchaseOrder/*//User'
                        PASSING po.OBJECT_VALUE AS "p" RETURNING CONTENT)
               AS VARCHAR2(30)) userid,
       CASE
         WHEN XMLExists('$p/PurchaseOrder/Reject/Date'
                        PASSING po.OBJECT_VALUE AS "p")
           THEN 'Rejected'
           ELSE 'Accepted'
       END "STATUS",
       XMLCast(XMLQuery('$p//Date'
                        PASSING po.OBJECT_VALUE AS "p" RETURNING CONTENT)
               AS VARCHAR2(12)) status_date
  FROM purchaseorder po
  WHERE XMLExists('$p//Date' PASSING po.OBJECT_VALUE AS "p")
  ORDER BY XMLCast(XMLQuery('$p//Date' PASSING po.OBJECT_VALUE AS "p"
                                       RETURNING CONTENT)
                   AS VARCHAR2(12));

REFERENCE                        USERID   STATUS   STATUS_DATE
```

```
------------------------------- -------- -------- ------------
VJONES-20020916140000000PDT      SVOLLMAN Accepted 2002-10-11
SMCCAIN-2002091213000000PDT      SKING    Rejected 2002-10-12

2 rows selected.
```

Example 4–4 uses a PL/SQL cursor to query XML data. It uses a local `XMLType` instance to store transient data.

Example 4–4 Querying Transient XMLTYPE Data using a PL/SQL Cursor

```
DECLARE
  xNode      XMLType;
  vText      VARCHAR2(256);
  vReference VARCHAR2(32);
  CURSOR getPurchaseOrder(reference IN VARCHAR2) IS
           SELECT OBJECT_VALUE XML
             FROM purchaseorder
             WHERE XMLExists('$p/PurchaseOrder[Reference=$r]'
                             PASSING OBJECT_VALUE AS "p",
                                     reference    AS "r");
BEGIN
  vReference := 'EABEL-20021009123335791PDT';
  FOR c IN getPurchaseOrder(vReference) LOOP
    xNode := c.XML.extract('//Requestor');
    SELECT XMLSerialize(CONTENT
                        XMLQuery('//text()' PASSING xNode RETURNING CONTENT))
           INTO vText FROM DUAL;
    DBMS_OUTPUT.put_line('The Requestor for Reference '
                         || vReference || ' is '|| vText);
  END LOOP;
  vReference := 'PTUCKER-20021009123335430PDT';
  FOR c IN getPurchaseOrder(vReference) LOOP
    xNode := c.XML.extract('//LineItem[@ItemNumber="1"]/Description');
    SELECT XMLSerialize(CONTENT
                        XMLQuery('//text()' PASSING xNode RETURNING CONTENT))
           INTO vText FROM DUAL;
    DBMS_OUTPUT.put_line('The Description of LineItem[1] for Reference '
                         || vReference || ' is '|| vText);
  END LOOP;
END;
/
The Requestor for Reference EABEL-20021009123335791PDT is Ellen S. Abel
The Description of LineItem[1] for Reference PTUCKER-20021009123335430PDT is
 Picnic at
Hanging Rock

PL/SQL procedure successfully completed.
```

Example 4–5 uses SQL/XML function `XMLTable` to extract data from an XML purchase-order document, and then inserts that data into a relational table.

Example 4–5 Extracting XML Data using XMLTABLE, and Inserting It into a Database Table

```
CREATE TABLE purchaseorder_table (reference          VARCHAR2(28) PRIMARY KEY,
                                  requestor          VARCHAR2(48),
                                  actions            XMLType,
                                  userid             VARCHAR2(32),
                                  costcenter         VARCHAR2(3),
                                  shiptoname         VARCHAR2(48),
                                  address            VARCHAR2(512),
```

```
                                phone               VARCHAR2(32),
                                rejectedby          VARCHAR2(32),
                                daterejected        DATE,
                                comments            VARCHAR2(2048),
                                specialinstructions VARCHAR2(2048));

CREATE TABLE purchaseorder_lineitem (reference,
                                   FOREIGN KEY ("REFERENCE")
                                     REFERENCES "PURCHASEORDER_TABLE" ("REFERENCE") ON DELETE CASCADE,
                                   lineno      NUMBER(10), PRIMARY KEY ("REFERENCE", "LINENO"),
                                   upc         VARCHAR2(14),
                                   description VARCHAR2(128),
                                   quantity    NUMBER(10),
                                   unitprice   NUMBER(12,2));

INSERT INTO purchaseorder_table (reference, requestor, actions, userid, costcenter, shiptoname, address,
                                phone, rejectedby, daterejected, comments, specialinstructions)
  SELECT t.reference, t.requestor, t.actions, t.userid, t.costcenter, t.shiptoname, t.address,
          t.phone, t.rejectedby, t.daterejected, t.comments, t.specialinstructions
    FROM purchaseorder p,
         XMLTable('/PurchaseOrder' PASSING p.OBJECT_VALUE
                  COLUMNS reference           VARCHAR2(28)   PATH 'Reference',
                          requestor           VARCHAR2(48)   PATH 'Requestor',
                          actions             XMLType        PATH 'Actions',
                          userid              VARCHAR2(32)   PATH 'User',
                          costcenter          VARCHAR2(3)    PATH 'CostCenter',
                          shiptoname          VARCHAR2(48)   PATH 'ShippingInstructions/name',
                          address             VARCHAR2(512)  PATH 'ShippingInstructions/address',
                          phone               VARCHAR2(32)   PATH 'ShippingInstructions/telephone',
                          rejectedby          VARCHAR2(32)   PATH 'Reject/User',
                          daterejected        DATE           PATH 'Reject/Date',
                          comments            VARCHAR2(2048) PATH 'Reject/Comments',
                          specialinstructions VARCHAR2(2048) PATH 'SpecialInstructions') t
    WHERE t.reference = 'EABEL-20021009123336251PDT';

INSERT INTO purchaseorder_lineitem (reference, lineno, upc, description, quantity, unitprice)
  SELECT t.reference, li.lineno, li.upc, li.description, li.quantity, li.unitprice
    FROM purchaseorder p,
         XMLTable('/PurchaseOrder' PASSING p.OBJECT_VALUE
                  COLUMNS reference VARCHAR2(28) PATH 'Reference',
                          lineitem XMLType PATH 'LineItems/LineItem') t,
         XMLTable('LineItem' PASSING t.lineitem
                  COLUMNS lineno      NUMBER(10)    PATH '@ItemNumber',
                          upc         VARCHAR2(14)  PATH 'Part/@Id',
                          description VARCHAR2(128) PATH 'Description',
                          quantity    NUMBER(10)    PATH 'Part/@Quantity',
                          unitprice   NUMBER(12,2)  PATH 'Part/@UnitPrice') li
    WHERE t.reference = 'EABEL-20021009123336251PDT';

SELECT reference, userid, shiptoname, specialinstructions FROM purchaseorder_table;

REFERENCE                       USERID   SHIPTONAME                                      SPECIALINSTRUCTIONS
------------------------------- -------- ----------------------------------------------- -------------------
EABEL-20021009123336251PDT      EABEL    Ellen S. Abel                                   Counter to Counter

SELECT reference, lineno, upc, description, quantity FROM purchaseorder_lineitem;

REFERENCE                           LINENO UPC            DESCRIPTION                          QUANTITY
------------------------------- ---------- -------------- ---------------------------------- ----------
EABEL-20021009123336251PDT               1 37429125526    Samurai 2: Duel at Ichijoji Temple          3
EABEL-20021009123336251PDT               2 37429128220    The Red Shoes                               4
EABEL-20021009123336251PDT               3 715515009058   A Night to Remember                         1
```

Example 4–6 defines and uses a PL/SQL procedure to extract data from an XML purchase-order document and insert it into a relational table.

Example 4–6 Extracting XML Data and Inserting It into a Table using a PL/SQL Procedure

```
CREATE OR REPLACE PROCEDURE insertPurchaseOrder(purchaseorder XMLType) AS reference VARCHAR2(28);
BEGIN
  INSERT INTO purchaseorder_table (reference, requestor, actions, userid, costcenter, shiptoname, address,
                                   phone, rejectedby, daterejected, comments, specialinstructions)
    SELECT * FROM XMLTable('$p/PurchaseOrder' PASSING purchaseorder AS "p"
                          COLUMNS reference           VARCHAR2(28)   PATH 'Reference',
                                  requestor           VARCHAR2(48)   PATH 'Requestor',
                                  actions             XMLType        PATH 'Actions',
                                  userid              VARCHAR2(32)   PATH 'User',
                                  costcenter          VARCHAR2(3)    PATH 'CostCenter',
                                  shiptoname          VARCHAR2(48)   PATH 'ShippingInstructions/name',
                                  address             VARCHAR2(512)  PATH 'ShippingInstructions/address',
                                  phone               VARCHAR2(32)   PATH 'ShippingInstructions/telephone',
                                  rejectedby          VARCHAR2(32)   PATH 'Reject/User',
                                  daterejected        DATE           PATH 'Reject/Date',
                                  comments            VARCHAR2(2048) PATH 'Reject/Comments',
                                  specialinstructions VARCHAR2(2048) PATH 'SpecialInstructions');

  INSERT INTO purchaseorder_lineitem (reference, lineno, upc, description, quantity, unitprice)
    SELECT t.reference, li.lineno, li.upc, li.description, li.quantity, li.unitprice
    FROM XMLTable('$p/PurchaseOrder' PASSING purchaseorder AS "p"
                  COLUMNS reference VARCHAR2(28) PATH 'Reference',
                          lineitem XMLType PATH 'LineItems/LineItem') t,
         XMLTable('LineItem' PASSING t.lineitem
                  COLUMNS lineno NUMBER(10)    PATH '@ItemNumber',
                          upc VARCHAR2(14)  PATH 'Part/@Id',
                          description VARCHAR2(128) PATH 'Description',
                          quantity NUMBER(10)    PATH 'Part/@Quantity',
                          unitprice NUMBER(12,2)  PATH 'Part/@UnitPrice') li;
END;
CALL insertPurchaseOrder(XMLType(bfilename('XMLDIR', 'purchaseOrder.xml'), nls_charset_id('AL32UTF8')));

SELECT reference, userid, shiptoname, specialinstructions FROM purchaseorder_table;

REFERENCE                      USERID   SHIPTONAME                                      SPECIALINSTRUCTIONS
------------------------------ -------- ----------------------------------------------- -------------------
SBELL-2002100912333601PDT      SBELL    Sarah J. Bell                                   Air Mail


SELECT reference, lineno, upc, description, quantity FROM purchaseorder_lineitem;

REFERENCE                 LINENO UPC          DESCRIPTION                       QUANTITY
------------------------- ------ ------------ --------------------------------- --------
SBELL-2002100912333601PDT      1 715515009058 A Night to Remember                       2
SBELL-2002100912333601PDT      2 37429140222  The Unbearable Lightness Of Being         2
SBELL-2002100912333601PDT      3 715515011020 Sisters                                   4
```

Example 4–7 extracts the purchase-order name from XML element `PurchaseOrder` for customers with "`ll`" (double L) in their names and the word "`Shores`" in the shipping instructions. It uses Oracle XQuery extension function `ora:contains` to perform full-text search.

Example 4–7 Searching XML Data using SQL/XML Functions

```
SELECT XMLCast(XMLQuery('$p/PurchaseOrder/Requestor'
                        PASSING po.OBJECT_VALUE AS "p" RETURNING CONTENT)
                  AS VARCHAR2(128)) name,
       count(*)
  FROM purchaseorder po
  WHERE
    XMLExists(
      'declare namespace ora="http://xmlns.oracle.com/xdb"; (: :)
       $p/PurchaseOrder/ShippingInstructions[ora:contains(address/text(), "Shores") > 0]'
     PASSING po.OBJECT_VALUE AS "p")
```

```
    AND XMLCast(XMLQuery('$p/PurchaseOrder/Requestor/text()'
                         PASSING po.OBJECT_VALUE AS "p" RETURNING CONTENT)
                  AS VARCHAR2(128))
        LIKE '%ll%'
  GROUP BY XMLCast(XMLQuery('$p/PurchaseOrder/Requestor'
                            PASSING po.OBJECT_VALUE AS "p" RETURNING CONTENT)
                     AS VARCHAR2(128));

NAME                   COUNT(*)
-------------------- ----------
Allan D. McEwen               9
Ellen S. Abel                 4
Sarah J. Bell                13
William M. Smith              7
```

Example 4–8 uses SQL/XML function `XMLQuery` to extract nodes identified by an XPath expression. The `XMLType` instance returned by `XMLQuery` can be a set of nodes, a singleton node, or a text value. Example 4–8 uses `XMLType` method `isFragment()` to determine whether the result is a fragment.

Example 4–8 Extracting Fragments from an XMLTYPE Instance using XMLQUERY

```
SELECT XMLCast(XMLQuery('$p/PurchaseOrder/Reference' PASSING po.OBJECT_VALUE AS "p"
                                                    RETURNING CONTENT)
               AS VARCHAR2(30)) reference,
       count(*)
  FROM purchaseorder po, XMLTable('$p//LineItem[Part/@Id="37429148327"]' PASSING OBJECT_VALUE AS "p")
  WHERE XMLQuery('$p/PurchaseOrder/LineItems/LineItem[Part/@Id="37429148327"]'
                 PASSING po.OBJECT_VALUE AS "p" RETURNING CONTENT).isFragment() = 1
  GROUP BY XMLCast(XMLQuery('$p/PurchaseOrder/Reference' PASSING po.OBJECT_VALUE AS "p" RETURNING CONTENT)
                   AS VARCHAR2(30))
  ORDER BY XMLCast(XMLQuery('$p/PurchaseOrder/Reference' PASSING po.OBJECT_VALUE AS "p" RETURNING CONTENT)
                   AS VARCHAR2(30));

REFERENCE                         COUNT(*)
-------------------------------- ----------
TFOX-20021009123337784PDT                 3
```

Note: You cannot insert fragments into `XMLType` columns. You can use SQL function `sys_XMLGen` to convert a fragment into a well-formed document by adding an enclosing tag. See "Generating XML using SQL Function SYS_XMLGEN" on page 18-46.

Updating XML Data

This section covers updating XML data, both transient data and data stored in tables. It describes the use of the following SQL functions:

- `updateXML`
- `insertChildXML`
- `insertChildXMLbefore`
- `insertChildXMLafter`
- `insertXMLbefore`
- `insertXMLafter`
- `appendChildXML`
- `deleteXML`

Updating an Entire XML Document

For unstructured storage (CLOB), an update effectively replaces the entire document. To update an entire XML document, use a SQL UPDATE statement. The right side of the UPDATE statement SET clause must be an XMLType instance. This can be created in any of the following ways:

- Use SQL functions or XML constructors that return an XML instance.
- Use the PL/SQL DOM APIs for XMLType that change and bind an existing XML instance.
- Use the Java DOM API that changes and binds an existing XML instance.

Updates for non-schema-based XML documents stored as CLOB values (unstructured storage) always update the entire XML document. Updates for non-schema-based documents stored as binary XML can be made in a piecewise manner. See "Updating XML Schema-Based and Non-Schema-Based XML Documents" on page 3-57.

Example 4–9 updates an XMLType instance using a SQL UPDATE statement.

Example 4–9 Updating XMLType Data using a SQL UPDATE Statement

```
SELECT t.reference, li.lineno, li.description
  FROM purchaseorder po,
       XMLTable('$p/PurchaseOrder' PASSING po.OBJECT_VALUE AS "p"
                COLUMNS reference VARCHAR2(28) PATH 'Reference',
                        lineitem  XMLType      PATH 'LineItems/LineItem') t,
       XMLTable('$l/LineItem' PASSING t.lineitem AS "l"
                COLUMNS lineno      NUMBER(10)    PATH '@ItemNumber',
                        description VARCHAR2(128) PATH 'Description') li
  WHERE t.reference = 'DAUSTIN-20021009123335811PDT' AND ROWNUM < 6;

REFERENCE                        LINENO DESCRIPTION
-------------------------------- ------- -----------------
DAUSTIN-20021009123335811PDT           1 Nights of Cabiria
DAUSTIN-20021009123335811PDT           2 For All Mankind
DAUSTIN-20021009123335811PDT           3 Dead Ringers
DAUSTIN-20021009123335811PDT           4 Hearts and Minds
DAUSTIN-20021009123335811PDT           5 Rushmore

UPDATE purchaseorder po
  SET po.OBJECT_VALUE = XMLType(bfilename('XMLDIR','NEW-DAUSTIN-20021009123335811PDT.xml'),
                                nls_charset_id('AL32UTF8'))
  WHERE XMLExists('$p/PurchaseOrder[Reference="DAUSTIN-20021009123335811PDT"]'
                  PASSING po.OBJECT_VALUE AS "p");

SELECT t.reference, li.lineno, li.description
  FROM purchaseorder po,
       XMLTable('$p/PurchaseOrder' PASSING po.OBJECT_VALUE AS "p"
                COLUMNS reference VARCHAR2(28) PATH 'Reference',
                        lineitem  XMLType      PATH 'LineItems/LineItem') t,
       XMLTable('$l/LineItem' PASSING t.lineitem AS "l"
                COLUMNS lineno      NUMBER(10)    PATH '@ItemNumber',
                        description VARCHAR2(128) PATH 'Description') li
  WHERE t.reference = 'DAUSTIN-20021009123335811PDT';

REFERENCE                        LINENO DESCRIPTION
-------------------------------- ------- --------------------------------
DAUSTIN-20021009123335811PDT           1 Dead Ringers
DAUSTIN-20021009123335811PDT           2 Getrud
DAUSTIN-20021009123335811PDT           3 Branded to Kill
```

SQL Functions that Update XML Data

There are several Oracle SQL functions that you can use to update XML data incrementally—that is, to replace, insert, or delete XML data without replacing the entire surrounding XML document. This is also called **partial updating**. These Oracle SQL functions are described in the following sections:

- `updateXML` – Replace XML nodes of any kind. See "UPDATEXML SQL Function" on page 4-14.
- `insertChildXML` – Insert XML element or attribute nodes as children of a given element node. See "INSERTCHILDXML SQL Function" on page 4-23.
- `insertChildXMLbefore` – Insert new collection elements immediately before a given collection element of the same type. See "INSERTCHILDXMLBEFORE SQL Function" on page 4-25.
- `insertChildXMLafter` – Insert new collection elements immediately after a given collection element of the same type. See "INSERTCHILDXMLAFTER SQL Function" on page 4-26.
- `insertXMLbefore` – Insert XML nodes of any kind immediately before a given node (other than an attribute node). See "INSERTXMLBEFORE SQL Function" on page 4-27.
- `insertXMLafter` – Insert XML nodes of any kind immediately after a given node (other than an attribute node). See "INSERTXMLAFTER SQL Function" on page 4-29.
- `appendChildXML` – Insert XML nodes of any kind as the last child nodes of a given element node. See "APPENDCHILDXML SQL Function" on page 4-30.
- `deleteXML` – Delete XML nodes of any kind. See "DELETEXML SQL Function" on page 4-32.

Use functions `insertChildXML`, `insertChildXMLbefore`, `insertChildXMLafter`, `insertXMLbefore`, `insertXMLafter`, and `appendChildXML` to insert XML data. Use function `deleteXML` to delete XML data. Use function `updateXML` to replace XML data.

In particular, do *not* use function `updateXML` to insert or delete XML data by replacing a parent node in its entirety. That works, but it is less efficient than using one of the other functions, which perform more localized updates.

These Oracle SQL functions do *not*, by themselves, change database data – they are all pure functions, without side effect. Each applies an XPath-expression argument to input XML data and returns a modified *copy* of the input XML data. You can then use that result with SQL DML operator `UPDATE` to modify database data. This is no different from the way you use SQL function `upper` to convert database data to uppercase: you must use a SQL DML operator such as `UPDATE` to change the stored data.

Each of these functions can be used on XML documents that are either schema-based or non-schema-based. For XML schema-based data, these Oracle SQL functions perform partial validation on the result, and, where appropriate, argument values are also checked for compatibility with the XML schema.

Note: Oracle SQL functions and `XMLType` methods respect the W3C XPath recommendation, which states that if an XPath expression targets *no nodes* when applied to XML data, then an empty sequence must be returned. An error must *not* be raised in this case.

The specific semantics of an Oracle SQL function or `XMLType` method that applies an XPath expression to XML data determines what is returned. For example, SQL/XML function `XMLQuery` returns `NULL` if its XPath-expression argument targets no nodes, and the updating Oracle SQL functions, such as `deleteXML`, return the input XML data unchanged. An error is never raised if no nodes are targeted, but updating SQL functions can raise an error if an XPath-expression argument targets inappropriate nodes, such as attribute nodes or text nodes.

See Also: "Partial Validation" on page 3-32 for more information about partial validation against an XML schema

Inserting XML Elements using SQL Functions

There are several Oracle SQL functions for inserting XML nodes into (a copy of) existing XML data. Each can insert nodes at multiple locations that are referenced by an XPath expression. They differ in the placement of the new nodes and how the target XML data is referenced.

- Function `appendChildXML` appends nodes to the target elements. That is, for each target element, it inserts one or more nodes of any kind as the element's last children.
- Function `insertChildXML` inserts new children (one or more elements of the same type or a single attribute) under target elements. The position of a new child element under its parent is not specified. If the target data is XML schema-based, then the schema can sometimes be used to determine the position. Otherwise, the position is arbitrary.
- Function `insertXMLbefore` inserts one or more nodes of *any kind* immediately before a target node (which is not an attribute node).

  Function `insertXMLafter` inserts a node similarly, but after the target, not before.
- Function `insertChildXMLbefore` is similar to `insertChildXML`, except that the inserted node must be an element (not an attribute), and you specify the position of the new element among its siblings. It is similar to `insertXMLbefore`, except that it inserts only *collection* elements, not arbitrary elements. The insertion position specifies a successor collection member. The actual element to be inserted must correspond to the element type for the collection.

  Function `insertChildXMLafter` inserts a node similarly, but after the target, not before.

Though the effect of `insertChildXMLbefore` (`-after`) is similar to that of `insertXMLbefore` (`-after`), the target location is expressed differently. For the former, the target is the parent of the new child. For the latter, the target is the succeeding (or preceding) sibling. This difference is reflected in the function names (`Child`).

For example, to insert a new `LineItem` element before the third `LineItem` element under element `/PurchaseOrder/LineItems`, you can use `insertChildXMLbefore`, specifying the target parent as `/PurchaseOrder/LineItems` and the succeeding sibling as `LineItem[3]`. Or you can use `insertXMLbefore`, specifying the target succeeding sibling as `/PurchaseOrder/LineItems/LineItem[3]`. If you use `insertChildXML` for the insertion, then you cannot specify the position of the new element in the collection—the resulting position is indeterminate.

Another difference among these functions is that all of them *except* `insertXMLbefore`, `insertXMLafter`, and `appendChildXML`—are optimized for SQL `UPDATE` operations on `XMLType` tables and columns that are stored object-relationally or as binary XML.

> **See Also:** "Optimization of Oracle SQL Functions that Modify XML Data" on page 4-20

UPDATEXML SQL Function

Oracle SQL function `updateXML` replaces XML nodes of any kind. The XML document that is the target of the update can be schema-based or non-schema-based.

A *copy* of the input `XMLType` instance is modified and returned. The original data is unaffected. You can then use the returned data with SQL operation `UPDATE` to modify database data.

Function `updateXML` has the following parameters (in order):

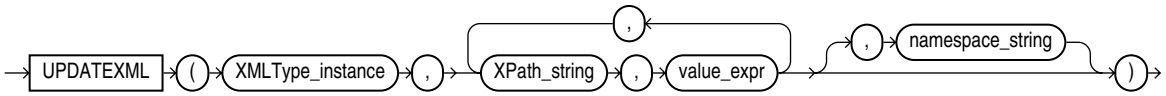
- ***target-data*** (`XMLType`) – The XML data containing the target node to replace.
- One or more *pairs* of *`xpath`* and *`replacement`* parameters:
  - ***xpath*** (`VARCHAR2`) – An XPath 1.0 expression that locates the nodes within *`target-data`* to replace. *Each* targeted node is replaced by *`replacement`*. These can be nodes of any kind. If *`xpath`* matches an empty sequence of nodes then no replacement is done, and *`target-data`* is returned unchanged (and no error is raised).
  - ***replacement*** (`XMLType` or `VARCHAR2`) – The XML data that replaces the data targeted by *`xpath`*. The data type of *`replacement`* must correspond to the data to be replaced. If *`xpath`* targets an element node for replacement, then the data type must be `XMLType`. If *`xpath`* targets an attribute node or a text node, then it must be `VARCHAR2`. For an attribute node, *`replacement`* is only the replacement *value* of the attribute (for example, 23), not the complete attribute node including the name (for example, `my_attribute="23"`).
- ***namespace*** (`VARCHAR2`, *optional*) – The XML namespace for parameter *`xpath`*.

Oracle SQL function `updateXML` can be used to *replace* existing elements, attributes, and other nodes with new values. It is *not* an efficient way to insert new nodes or delete existing ones. You can perform insertions and deletions with `updateXML` only by using it to replace the entire node that is the parent of the node to be inserted or deleted.

Function `updateXML` updates only the transient XML instance in memory. Use a SQL `UPDATE` statement to update data stored in tables.

Figure 4–3 illustrates the syntax.

Figure 4–3 UPDATEXML Syntax



****

Example 4–10 uses `updateXML` on the right side of an `UPDATE` statement to update the XML document in a table instead of creating a new document. The entire document is updated, not just the part that is selected.

Example 4–10 Updating XMLTYPE using UPDATE and UPDATEXML

```
SELECT XMLQuery('$p/PurchaseOrder/Actions/Action[1]' PASSING po.OBJECT_VALUE AS "p"
                                                     RETURNING CONTENT) action
  FROM purchaseorder po
  WHERE XMLExists('$p/PurchaseOrder[Reference="SBELL-2002100912333601PDT"]'
                  PASSING po.OBJECT_VALUE AS "p");

ACTION
--------------------------------
<Action>
  <User>SVOLLMAN</User>
</Action>

UPDATE purchaseorder po
  SET po.OBJECT_VALUE = updateXML(po.OBJECT_VALUE,
                                  '/PurchaseOrder/Actions/Action[1]/User/text()',
                                  'SKING')
  WHERE XMLExists('$p/PurchaseOrder[Reference="SBELL-2002100912333601PDT"]'
                  PASSING po.OBJECT_VALUE AS "p");

SELECT XMLQuery('$p/PurchaseOrder/Actions/Action[1]' PASSING po.OBJECT_VALUE AS "p"
                                                     RETURNING CONTENT) action
  FROM purchaseorder po
  WHERE XMLExists('$p/PurchaseOrder[Reference="SBELL-2002100912333601PDT"]'
                  PASSING po.OBJECT_VALUE AS "p");

ACTION
---------------------------------
<Action>
  <User>SKING</User>
</Action>
```

Example 4–11 updates multiple nodes using Oracle SQL function `updateXML`.

Example 4–11 Updating Multiple Text Nodes and Attribute Values using UPDATEXML

```
SELECT XMLCast(XMLQuery('$p/PurchaseOrder/Requestor'
                        PASSING po.OBJECT_VALUE AS "p" RETURNING CONTENT)
               AS VARCHAR2(30)) name,
       XMLQuery('$p/PurchaseOrder/LineItems'
                PASSING po.OBJECT_VALUE AS "p" RETURNING CONTENT) lineitems
  FROM purchaseorder po
  WHERE XMLExists('$p/PurchaseOrder[Reference="SBELL-2002100912333601PDT"]'
                  PASSING po.OBJECT_VALUE AS "p");

NAME            LINEITEMS
--------------- ------------------------------------------------------------------
```

```
Sarah J. Bell    <LineItems>
                   <LineItem ItemNumber="1">
                     <Description>A Night to Remember</Description>
                     <Part Id="715515009058" UnitPrice="39.95" Quantity="2"/>
                   </LineItem>
                   <LineItem ItemNumber="2">
                     <Description>The Unbearable Lightness Of Being</Description>
                     <Part Id="37429140222" UnitPrice="29.95" Quantity="2"/>
                   </LineItem>
                   <LineItem ItemNumber="3">
                     <Description>Sisters</Description>
                     <Part Id="715515011020" UnitPrice="29.95" Quantity="4"/>
                   </LineItem>
                 </LineItems>

UPDATE purchaseorder
  SET OBJECT_VALUE = updateXML(OBJECT_VALUE,
                               '/PurchaseOrder/Requestor/text()','Stephen G. King',
                               '/PurchaseOrder/LineItems/LineItem[1]/Part/@Id','786936150421',
                               '/PurchaseOrder/LineItems/LineItem[1]/Description/text()','The Rock',
                               '/PurchaseOrder/LineItems/LineItem[3]',
                               XMLType('<LineItem ItemNumber="99">
                                          <Description>Dead Ringers</Description>
                                          <Part Id="715515009249" UnitPrice="39.95" Quantity="2"/>
                                        </LineItem>'))
  WHERE XMLExists('$p/PurchaseOrder[Reference="SBELL-2002100912333601PDT"]'
                  PASSING OBJECT_VALUE AS "p");

SELECT XMLCast(XMLQuery('$p/PurchaseOrder/Requestor'
                        PASSING po.OBJECT_VALUE AS "p" RETURNING CONTENT)
               AS VARCHAR2(30)) name,
       XMLQuery('$p/PurchaseOrder/LineItems'
                PASSING po.OBJECT_VALUE AS "p" RETURNING CONTENT) lineitems
  FROM purchaseorder po
  WHERE XMLExists('$p/PurchaseOrder[Reference="SBELL-2002100912333601PDT"]'
                  PASSING po.OBJECT_VALUE AS "p");

NAME             LINEITEMS
---------------- ------------------------------------------------------------------
Stephen G. King  <LineItems>
                   <LineItem ItemNumber="1">
                     <Description>The Rock</Description>
                     <Part Id="786936150421" UnitPrice="39.95" Quantity="2"/>
                   </LineItem>
                   <LineItem ItemNumber="2">
                     <Description>The Unbearable Lightness Of Being</Description>
                     <Part Id="37429140222" UnitPrice="29.95" Quantity="2"/>
                   </LineItem>
                   <LineItem ItemNumber="99">
                     <Description>Dead Ringers</Description>
                     <Part Id="715515009249" UnitPrice="39.95" Quantity="2"/>
                   </LineItem>
                 </LineItems>
```

Example 4–12 uses SQL function `updateXML` to update selected nodes within a collection.

Example 4–12 Updating Selected Nodes within a Collection using UPDATEXML

```
SELECT XMLCast(XMLQuery('$p/PurchaseOrder/Requestor'
                        PASSING po.OBJECT_VALUE AS "p" RETURNING CONTENT)
               AS VARCHAR2(30)) name,
       XMLQuery('$p/PurchaseOrder/LineItems'
                PASSING po.OBJECT_VALUE AS "p" RETURNING CONTENT) lineitems
  FROM purchaseorder po
```

```
  WHERE XMLExists('$p/PurchaseOrder[Reference="SBELL-2002100912333601PDT"]'
                  PASSING po.OBJECT_VALUE AS "p");

NAME             LINEITEMS
---------------- ------------------------------------------------------------
Sarah J. Bell    <LineItems>
                   <LineItem ItemNumber="1">
                     <Description>A Night to Remember</Description>
                     <Part Id="715515009058" UnitPrice="39.95" Quantity="2"/>
                   </LineItem>
                   <LineItem ItemNumber="2">
                     <Description>The Unbearable Lightness Of Being</Description>
                     <Part Id="37429140222" UnitPrice="29.95" Quantity="2"/>
                   </LineItem>
                   <LineItem ItemNumber="3">
                     <Description>Sisters</Description>
                     <Part Id="715515011020" UnitPrice="29.95" Quantity="4"/>
                   </LineItem>
                 </LineItems>

UPDATE purchaseorder
  SET OBJECT_VALUE =
      updateXML(OBJECT_VALUE,
                '/PurchaseOrder/Requestor/text()','Stephen G. King',
                '/PurchaseOrder/LineItems/LineItem/Part[@Id="715515009058"]/@Quantity',
                25,
                '/PurchaseOrder/LineItems/LineItem[Description/text() =
                                                   "The Unbearable Lightness Of Being"]',
                XMLType('<LineItem ItemNumber="99">
                          <Part Id="786936150421" Quantity="5" UnitPrice="29.95"/>
                          <Description>The Rock</Description>
                        </LineItem>'))
  WHERE XMLExists('$p/PurchaseOrder[Reference="SBELL-2002100912333601PDT"]'
                  PASSING OBJECT_VALUE AS "p");

SELECT XMLCast(XMLQuery('$p/PurchaseOrder/Requestor'
                        PASSING po.OBJECT_VALUE AS "p" RETURNING CONTENT)
               AS VARCHAR2(30)) name,
       XMLQuery('$p/PurchaseOrder/LineItems'
                PASSING po.OBJECT_VALUE AS "p" RETURNING CONTENT) lineitems
  FROM purchaseorder po
  WHERE XMLExists('$p/PurchaseOrder[Reference="SBELL-2002100912333601PDT"]'
                  PASSING po.OBJECT_VALUE AS "p");

NAME             LINEITEMS
---------------- ------------------------------------------------------------
Stephen G. King  <LineItems>
                   <LineItem ItemNumber="1">
                     <Description>A Night to Remember</Description>
                     <Part Id="715515009058" UnitPrice="39.95" Quantity="25"/>
                   </LineItem>
                   <LineItem ItemNumber="99">
                     <Part Id="786936150421" Quantity="5" UnitPrice="29.95"/>
                     <Description>The Rock</Description>
                   </LineItem>
                   <LineItem ItemNumber="3">
                     <Description>Sisters</Description>
                     <Part Id="715515011020" UnitPrice="29.95" Quantity="4"/>
                   </LineItem>
                 </LineItems>
```

UPDATEXML and NULL Values

- If you update an XML *element* to NULL, the attributes and children of the element are removed, and the element becomes empty. The type and namespace properties of the element are retained. See Example 4–13.
- If you update an *attribute* value to NULL, the value appears as the empty string. See Example 4–13.
- If you update the *text* node of an element to NULL, the content (text) of the element is removed. The element itself remains, but it is empty. See Example 4–14.

Example 4–13 updates all of the following to NULL:

- The Description element and the Quantity attribute of the LineItem element whose Part element has attribute Id value 715515009058.
- The LineItem element whose Description element has the content (text) "The Unbearable Lightness Of Being".

Example 4–13 NULL Updates with UPDATEXML – Element and Attribute

```
SELECT XMLCast(XMLQuery('$p/PurchaseOrder/Requestor'
                        PASSING po.OBJECT_VALUE AS "p" RETURNING CONTENT)
               AS VARCHAR2(30)) name,
       XMLQuery('$p/PurchaseOrder/LineItems'
                PASSING po.OBJECT_VALUE AS "p" RETURNING CONTENT) lineitems
  FROM purchaseorder po
  WHERE XMLExists('$p/PurchaseOrder[Reference="SBELL-2002100912333601PDT"]'
                  PASSING po.OBJECT_VALUE AS "p");

NAME             LINEITEMS
---------------- ----------------------------------------------------------------
Sarah J. Bell    <LineItems>
                   <LineItem ItemNumber="1">
                     <Description>A Night to Remember</Description>
                     <Part Id="715515009058" UnitPrice="39.95" Quantity="2"/>
                   </LineItem>
                   <LineItem ItemNumber="2">
                     <Description>The Unbearable Lightness Of Being</Description>
                     <Part Id="37429140222" UnitPrice="29.95" Quantity="2"/>
                   </LineItem>
                   <LineItem ItemNumber="3">
                     <Description>Sisters</Description>
                     <Part Id="715515011020" UnitPrice="29.95" Quantity="4"/>
                   </LineItem>
                 </LineItems>

UPDATE purchaseorder
  SET OBJECT_VALUE =
      updateXML(
        OBJECT_VALUE,
        '/PurchaseOrder/LineItems/LineItem[Part/@Id="715515009058"]/Description', NULL,
             '/PurchaseOrder/LineItems/LineItem/Part[@Id="715515009058"]/@Quantity', NULL,
        '/PurchaseOrder/LineItems/LineItem[Description/text()=
                                           "The Unbearable Lightness Of Being"]', NULL)
  WHERE XMLExists('$p/PurchaseOrder[Reference="SBELL-2002100912333601PDT"]'
                  PASSING OBJECT_VALUE AS "p");

SELECT XMLCast(XMLQuery('$p/PurchaseOrder/Requestor'
                        PASSING po.OBJECT_VALUE AS "p" RETURNING CONTENT)
               AS VARCHAR2(30)) name,
```

```
       XMLQuery('$p/PurchaseOrder/LineItems'
                PASSING po.OBJECT_VALUE AS "p" RETURNING CONTENT) lineitems
  FROM purchaseorder po
  WHERE XMLExists('$p/PurchaseOrder[Reference="SBELL-2002100912333601PDT"]'
                  PASSING po.OBJECT_VALUE AS "p");

NAME             LINEITEMS
---------------- ------------------------------------------------------------
Sarah J. Bell    <LineItems>
                   <LineItem ItemNumber="1">
                     <Description/>
                     <Part Id="715515009058" UnitPrice="39.95" Quantity=""/>
                   </LineItem>
                   <LineItem/>
                   <LineItem ItemNumber="3">
                     <Description>Sisters</Description>
                     <Part Id="715515011020" UnitPrice="29.95" Quantity="4"/>
                   </LineItem>
                 </LineItems>
```

Example 4–14 updates the text node of a `Part` element whose `Description` attribute has value "`A Night to Remember`" to `NULL`. The XML data for this example corresponds to a different, revised purchase-order XML schema – see "Scenario for Copy-Based Evolution" on page 10-2. In that XML schema, `Description` is an attribute of the `Part` element, not a sibling element.

Example 4–14 NULL Updates with UPDATEXML – Text Node

```
SELECT XMLCast(XMLQuery('$p/PurchaseOrder/LineItems/LineItem/Part[@Description="A Night to Remember"]'
                        PASSING po.OBJECT_VALUE AS "p" RETURNING CONTENT)
               AS VARCHAR2(128)) part
  FROM purchaseorder po
  WHERE XMLExists('$p/PurchaseOrder[@Reference="SBELL-2003030912333601PDT"]'
                  PASSING po.OBJECT_VALUE AS "p");

PART
----
<Part Description="A Night to Remember" UnitCost="39.95">715515009058</Part>

UPDATE purchaseorder
  SET OBJECT_VALUE =
        updateXML(OBJECT_VALUE,
                  '/PurchaseOrder/LineItems/LineItem/Part[@Description="A Night to Remember"]/text()', NULL)
  WHERE XMLExists('$p/PurchaseOrder[@Reference="SBELL-2003030912333601PDT"]'
                  PASSING OBJECT_VALUE AS "p");

SELECT XMLCast(XMLQuery('$p/PurchaseOrder/LineItems/LineItem/Part[@Description="A Night to Remember"]'
                        PASSING po.OBJECT_VALUE AS "p" RETURNING CONTENT)
               AS VARCHAR2(128)) part
  FROM purchaseorder po
  WHERE XMLExists('$p/PurchaseOrder[@Reference="SBELL-2003030912333601PDT"]'
                  PASSING po.OBJECT_VALUE AS "p");

PART
----
<Part Description="A Night to Remember" UnitCost="39.95"/>
```

See Also: Example 3–36, "Updating XML Content using UPDATEXML"

Updating the Same XML Node More Than Once

You can update the same XML node more than once in an `updateXML` expression. For example, you can update both `/EMP[EMPNO=217]` and `/EMP[EMPNAME="Jane"]/EMPNO`, where the first XPath identifies the `EMPNO` node 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.

Preserving DOM Fidelity When using UPDATEXML

Here are some guidelines for preserving DOM fidelity when using Oracle SQL function `updateXML`:

When DOM Fidelity is Preserved When you update an element to `NULL`, you make that element appear *empty* in its parent, such as in `<myElem/>`.

When you update a text node inside an element to `NULL`, you *remove* that text node from the element.

When you update an attribute node to `NULL`, you make the value of the attribute become the *empty* string, for example, `myAttr=""`.

When DOM Fidelity is Not Preserved When you update a `complexType` element to `NULL`, you make the element appear *empty* in its parent, for example, `<myElem/>`.

When you update a SQL-inlined `simpleType` element to `NULL`, you make the element *disappear* from its parent.

When you update a text node to `NULL`, you are doing the same thing as setting the parent `simpleType` element to `NULL`. Furthermore, text nodes can appear only inside `simpleType` elements when DOM fidelity is not preserved, since there is no positional descriptor with which to store mixed content.

When you update an attribute node to `NULL`, you *remove* the attribute from the element.

Determining Whether DOM Fidelity is Preserved You can determine whether or not DOM fidelity is preserved for particular parts of a given `XMLType` in a given XML schema by querying the schema metadata for attribute `maintainDOM`.

See Also:

- "Querying a Registered XML Schema to Obtain Annotations" on page 7-43 for an example of querying a schema to retrieve DOM fidelity values
- "DOM Fidelity" on page 7-16

Optimization of Oracle SQL Functions that Modify XML Data

In most cases, the Oracle SQL functions that modify XML data materialize a copy of the entire input XML document in memory, then update the copy. However, functions `updateXML`, `insertChildXML`, `insertChildXMLbefore`, `insertChildXMLafter`, and `deleteXML`—that is, all *except* `insertXMLbefore`, `insertXMLafter`, and `appendChildXML`—are optimized for SQL `UPDATE` operations on `XMLType` tables and columns that are stored object-relationally or as binary XML.

For structured storage, if particular conditions are met, then the function call can be rewritten to update the object-relational columns directly with the values. For binary

XML storage, data preceding the targeted update is not modified, and, if SecureFile LOBs are used (the default behavior), then sliding inserts are used to update only the portions of the data that need changing.

See Also:

- "Updating XML Schema-Based and Non-Schema-Based XML Documents" on page 3-57 for more about piecewise updating
- Chapter 3, "Using Oracle XML DB" and Chapter 8, "XPath Rewrite for Structured Storage" for information about the conditions for XPath rewrite
- "Performance Tuning for XQuery" on page 5-30

As an example with object-relational storage, the XPath argument to `updateXML` in Example 4–15 is processed by Oracle XML DB and rewritten into equivalent object-relational SQL code, as illustrated in Example 4–16.

Example 4–15 XPath Expressions in UPDATEXML Expression

```
SELECT XMLCast(XMLQuery('$p/PurchaseOrder/User'
PASSING po.OBJECT_VALUE AS "p" RETURNING CONTENT)
       AS VARCHAR2(30))
  FROM purchaseorder po
  WHERE XMLExists('$p/PurchaseOrder[Reference="SBELL-2002100912333601PDT"]'
  PASSING po.OBJECT_VALUE AS "p");

XMLCAST(XMLQUERY('$P/PURCHASEO
------------------------------
SBELL

UPDATE purchaseorder
  SET OBJECT_VALUE = updateXML(OBJECT_VALUE, '/PurchaseOrder/User/text()', 'SVOLLMAN')
  WHERE XMLExists('$p/PurchaseOrder[Reference="SBELL-2002100912333601PDT"]'
  PASSING OBJECT_VALUE AS "p");

SELECT XMLCast(XMLQuery('$p/PurchaseOrder/User'
PASSING po.OBJECT_VALUE AS "p" RETURNING CONTENT)
       AS VARCHAR2(30))
  FROM purchaseorder po
  WHERE XMLExists('$p/PurchaseOrder[Reference="SBELL-2002100912333601PDT"]'
  PASSING po.OBJECT_VALUE AS "p");

XMLCAST(XMLQUERY('$P/PURCHASEO
------------------------------
SVOLLMAN
```

Example 4–16 Object Relational Equivalent of UPDATEXML Expression

```
SELECT XMLCast(XMLQuery('$p/PurchaseOrder/User'
                        PASSING po.OBJECT_VALUE AS "p" RETURNING CONTENT)
               AS VARCHAR2(30))
  FROM purchaseorder po
  WHERE XMLExists('$p/PurchaseOrder[Reference="SBELL-2002100912333601PDT"]'
                  PASSING po.OBJECT_VALUE AS "p");

XMLCAST(XMLQUERY('$P/PURCHASEO
------------------------------
SBELL
```

```
UPDATE purchaseorder p
   SET p."XMLDATA"."USERID" = 'SVOLLMAN'
   WHERE p."XMLDATA"."REFERENCE" = 'SBELL-2002100912333601PDT';

SELECT XMLCast(XMLQuery('$p/PurchaseOrder/User'
                        PASSING po.OBJECT_VALUE AS "p" RETURNING CONTENT)
               AS VARCHAR2(30))
  FROM purchaseorder po
  WHERE XMLExists('$p/PurchaseOrder[Reference="SBELL-2002100912333601PDT"]'
                  PASSING po.OBJECT_VALUE AS "p");


XMLCAST(XMLQUERY('$P/PURCHASEO
------------------------------
SVOLLMAN
```

> **Note:** The use of `XMLDATA` for DML is shown here only as an illustration of internal Oracle XML DB behavior. Do *not* use `XMLDATA` yourself for DML operations. You can use `XMLDATA` directly only for DDL operations, never for DML operations.
>
> More generally, in your code, do not rely on the current mapping between the XML Schema object model and the SQL object model. This Oracle XML DB implementation mapping might change in the future.

Creating XML Views using Oracle SQL Functions that Modify XML Data

You can use the Oracle SQL functions that modify XML data (`updateXML`, `insertChildXML`, `insertChildXMLbefore`, `insertChildXMLafter`, `insertXMLbefore`, `insertXMLafter`, `appendChildXML`, and `deleteXML`) to create new views of XML data.

Example 4–17 creates a view of table `purchaseorder` using Oracle SQL function `updateXML`.

Example 4–17 Creating a View using UPDATEXML

```
CREATE OR REPLACE VIEW purchaseorder_summary OF XMLType AS
  SELECT updateXML(OBJECT_VALUE,
                   '/PurchaseOrder/Actions', NULL,
                   '/PurchaseOrder/ShippingInstructions', NULL,
                   '/PurchaseOrder/LineItems', NULL) AS XML
  FROM purchaseorder p;

SELECT OBJECT_VALUE FROM purchaseorder_summary
  WHERE XMLExists('$p/PurchaseOrder[Reference="DAUSTIN-20021009123335811PDT"]'
                  PASSING OBJECT_VALUE AS "p");

OBJECT_VALUE
------------------------------------------------------------------------------
<PurchaseOrder
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation=
      "http://localhost:8080/source/schemas/poSource/xsd/purchaseOrder.xsd">
  <Reference>DAUSTIN-20021009123335811PDT</Reference>
  <Actions/>
  <Reject/>
  <Requestor>David L. Austin</Requestor>
```

```
  <User>DAUSTIN</User>
  <CostCenter>S30</CostCenter>
  <ShippingInstructions/>
  <SpecialInstructions>Courier</SpecialInstructions>
  <LineItems/>
</PurchaseOrder>
```

INSERTCHILDXML SQL Function

Oracle SQL function `insertChildXML` inserts new children (one or more elements of the same type or a single attribute) under parent XML elements. The XML document that is the target of the insertion can be schema-based or non-schema-based.

A *copy* of the input `XMLType` instance is modified and returned. The original data is unaffected. You can then use the returned data with SQL operation `UPDATE` to modify database data.

Function `insertChildXML` has the following parameters (in order):

- ***target-data*** (`XMLType`) – The XML data containing the target parent element.
- ***parent-xpath*** (`VARCHAR2`) – An XPath 1.0 expression that locates the parent elements within *`target-data`*. The *`child-data`* is inserted under *each* parent element.

  If *`parent-xpath`* matches an empty sequence of element nodes, then no insertion is done, and *`target-data`* is returned unchanged (no error is raised). If *`parent-xpath`* does not match a sequence of element nodes (in particular, if *`parent-xpath`* matches one or more *attribute* or *text* nodes), then an error is raised.
- ***child-name*** (`VARCHAR2`) – The name of the child elements or attribute to insert. An attribute name is distinguished from an element name by having an at-sign (`@`) prefix as part of *`child-name`*, for example, `@my_attribute` versus `my_element`. (The at-sign is not part of the attribute name, but serves in the argument to indicate that *`child-name`* refers to an attribute.)
- ***child-data*** (`XMLType` or `VARCHAR2`) – The child XML data to insert:
  - If one or more *elements* are being inserted, then this is of data type `XMLType`, and it contains element nodes. *Each* of the top-level element nodes in *`child-data`* must have the same name (tag) as *`child-name`* (or else an error is raised).
  - If an *attribute* is being inserted, then this is of data type `VARCHAR2`, and it represents the (scalar) attribute value. If an attribute of the same name already exists at the insertion location, then an error is raised.
- ***namespace*** (`VARCHAR2`, *optional*) – The XML namespace for parameters *`parent-xpath`* and *`child-data`*.

XML data *`child-data`* is inserted as one or more child elements, or a single child attribute, under *each* of the parent elements located at *`parent-xpath`*.

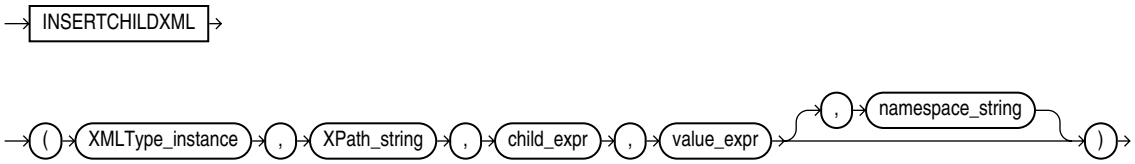
In order of decreasing precedence, function `insertChildXML` has the following behavior for `NULL` arguments:

- If *`child-name`* is `NULL`, then an error is raised.
- If *`target-data`* or *`parent-xpath`* is `NULL`, then `NULL` is returned.
- If *`child-data`* is `NULL`, then:

- If *child-name* names an element, then no insertion is done, and *target-data* is returned unchanged.
- If *child-name* names an attribute, then an empty attribute value is inserted, for example, `my_attribute = ""`.

Figure 4–4 shows the syntax.

Figure 4–4 INSERTCHILDXML Syntax



If *target-data* is XML *schema-based*, then the schema is consulted to determine the insertion positions. For example, if the schema constrains child elements named *child-name* to be the first child elements of a *parent-xpath*, then the insertion takes this into account. Similarly, if the *child-name* or *child-data* argument is inappropriate for an associated schema, then an error is raised.

If the parent element does *not* yet have a child corresponding in name and kind to *child-name* (and if such a child is permitted by the associated XML schema, if any), then *child-data* is inserted as new child elements, or a new attribute value, named *child-name*.

If the parent element already has a child *attribute* named *child-name* (without the at-sign), then an error is raised. If the parent element already has a child *element* named *child-name* (and if more than one child element is permitted by the associated XML schema, if any), then *child-data* is inserted so that its elements become child elements named *child-name*, but their positions in the sequence of children are unpredictable.

If you need to insert elements into an existing, non-empty collection of child elements, and the order is important to you, then use SQL/XML function `appendChildXML` or `insertXMLbefore`.

Example 4–18 shows how to use a SQL `UPDATE` statement and Oracle SQL function `insertChildXML` to insert a new `LineItem` element as a child of element `LineItems`.

Example 4–18 Inserting a LineItem Element into a LineItems Element

```
SELECT XMLQuery('$p/PurchaseOrder/LineItems/LineItem[@ItemNumber=222]'
                PASSING po.OBJECT_VALUE AS "p" RETURNING CONTENT)
  FROM purchaseorder po
  WHERE XMLExists('$p/PurchaseOrder[Reference="AMCEWEN-20021009123336171PDT"]'
                  PASSING po.OBJECT_VALUE AS "p");

XMLQUERY('$P/PURCHASEORDER/LINEITEMS/LINEITEM[@ITEMNUMBER=222]'
----------------------------------------------------------------

1 row selected.

UPDATE purchaseorder
  SET OBJECT_VALUE =
        insertChildXML(OBJECT_VALUE,
                       '/PurchaseOrder/LineItems',
                       'LineItem',
```

```
                        XMLType('<LineItem ItemNumber="222">
                                   <Description>The Harder They Come</Description>
                                   <Part Id="953562951413"
                                         UnitPrice="22.95"
                                         Quantity="1"/>
                                 </LineItem>'))
  WHERE XMLExists('$p/PurchaseOrder[Reference="AMCEWEN-20021009123336171PDT"]'
                  PASSING OBJECT_VALUE AS "p");

SELECT XMLQuery('$p/PurchaseOrder/LineItems/LineItem[@ItemNumber=222]'
                PASSING po.OBJECT_VALUE AS "p" RETURNING CONTENT)
  FROM purchaseorder po
  WHERE XMLExists('$p/PurchaseOrder[Reference="AMCEWEN-20021009123336171PDT"]'
                  PASSING po.OBJECT_VALUE AS "p");

XMLQUERY('$P/PURCHASEORDER/LINEITEMS/LINEITEM[@ITEMNUMBER=222]'
------------------------------------------------------------
<LineItem ItemNumber="222">
  <Description>The Harder They Come</Description>
  <Part Id="953562951413" UnitPrice="22.95" Quantity="1"/>
</LineItem>

1 row selected.
```

If the XML data to be updated is XML schema-based and it refers to a namespace, then the data to be inserted must also refer to the same namespace. Otherwise, an error is raised because the inserted data does not conform to the XML schema.

Example 4–19 is the same as Example 4–18, except that the `LineItem` element to be inserted refers to a namespace. This assumes that the relevant XML schema requires a namespace for this element.

Example 4–19 Inserting an Element that Uses a Namespace

```
UPDATE purchaseorder
  SET OBJECT_VALUE =
        insertChildXML(OBJECT_VALUE,
                       '/PurchaseOrder/LineItems',
                       'LineItem',
                       XMLType('<LineItem xmlns="films.xsd" ItemNumber="222">
                                   <Description>The Harder They Come</Description>
                                   <Part Id="953562951413"
                                         UnitPrice="22.95"
                                         Quantity="1"/>
                                 </LineItem>'))
  WHERE XMLExists('$p/PurchaseOrder[Reference="AMCEWEN-20021009123336171PDT"]'
                  PASSING OBJECT_VALUE AS "p");
```

Note that this use of namespaces is different from the use of a namespace *argument* to function `insertChildXML`. Namespaces supplied in that optional argument apply only to the XPath argument, not to the content to be inserted.

INSERTCHILDXMLBEFORE SQL Function

Oracle SQL function `insertChildXMLbefore` inserts one or more collection elements as children of target parent elements. The insertion for each target occurs immediately before a specified existing collection element. The existing XML document that is the target of the insertion can be schema-based or non-schema-based.

A *copy* of the input `XMLType` instance is modified and returned. The original data is unaffected. You can then use the returned data with SQL operation `UPDATE` to modify database data.

Function `insertChildXMLbefore` has the following parameters (in order):

- ***target-data*** (`XMLType`) – The XML data that is the target of the insertion.
- ***parent-xpath*** (`VARCHAR2`) – An XPath 1.0 expression that locates the parent elements within *`target-data`*. The *`child-data`* is inserted under *each* parent element.

  If *`parent-xpath`* matches an empty sequence of element nodes, then no insertion is done, and *`target-data`* is returned unchanged (no error is raised). If *`parent-xpath`* does not match a sequence of element nodes (in particular, if *`parent-xpath`* matches one or more *attribute* or *text* nodes), then an error is raised.
- ***child-xpath*** (`VARCHAR2`) – A relative XPath 1.0 expression that locates the existing child that will become the successor of the inserted *`child-data`*. It must name a child element of the element indicated by *`parent-xpath`*, and it can include a predicate.
- ***child-data*** (`XMLType`) – The child element XML data to insert. This is of data type `XMLType`, and it contains element nodes. *Each* of the top-level element nodes in *`child-data`* must have the same data type as the element indicated by *`child-xpath`* (or else an error is raised).
- ***namespace*** (*optional*, `VARCHAR2`) – The namespace for parameters *`parent-xpath`*, *`child-xpath`*, and *`child-data`*.

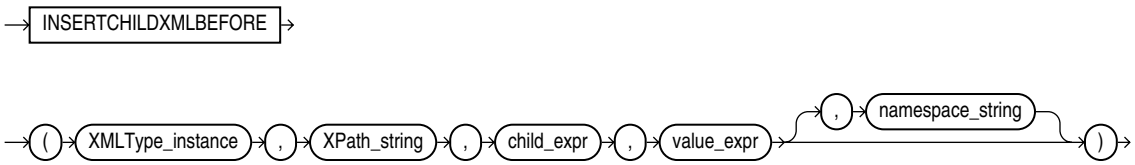
XML data *`child-data`* is inserted as one or more child elements under *each* of the parent elements located at *`parent-xpath`*.

In order of decreasing precedence, function `insertChildXMLbefore` has the following behavior for `NULL` arguments:

- If *`child-name`* is `NULL`, then an error is raised.
- If *`target-data`* or *`parent-xpath`* is `NULL`, then `NULL` is returned.
- If *`child-data`* is `NULL`, then no insertion is done, and *`target-data`* is returned unchanged.

Figure 4–5 shows the syntax.

Figure 4–5 INSERTCHILDXMLBEFORE Syntax



INSERTCHILDXMLAFTER SQL Function

Oracle SQL function `insertChildXMLafter` inserts one or more collection elements as children of target parent elements. The insertion for each target occurs immediately after a specified existing collection element. The existing XML document that is the target of the insertion can be schema-based or non-schema-based.

A *copy* of the input `XMLType` instance is modified and returned. The original data is unaffected. You can then use the returned data with SQL operation `UPDATE` to modify database data.

Function `insertChildXMLafter` has the following parameters (in order):

- ***target-data*** (`XMLType`) – The XML data that is the target of the insertion.
- ***parent-xpath*** (`VARCHAR2`) – An XPath 1.0 expression that locates the parent elements within *`target-data`*. The *`child-data`* is inserted under *each* parent element.

  If *`parent-xpath`* matches an empty sequence of element nodes, then no insertion is done, and *`target-data`* is returned unchanged (no error is raised). If *`parent-xpath`* does not match a sequence of element nodes (in particular, if *`parent-xpath`* matches one or more *attribute* or *text* nodes), then an error is raised.
- ***child-xpath*** (`VARCHAR2`) – A relative XPath 1.0 expression that locates the existing child that will become the predecessor of the inserted *`child-data`*. It must name a child element of the element indicated by *`parent-xpath`*, and it can include a predicate.
- ***child-data*** (`XMLType`) – The child element XML data to insert. This is of data type `XMLType`, and it contains element nodes. *Each* of the top-level element nodes in *`child-data`* must have the same data type as the element indicated by *`child-xpath`* (or else an error is raised).
- ***namespace*** (*optional*, `VARCHAR2`) – The namespace for parameters *`parent-xpath`*, *`child-xpath`*, and *`child-data`*.

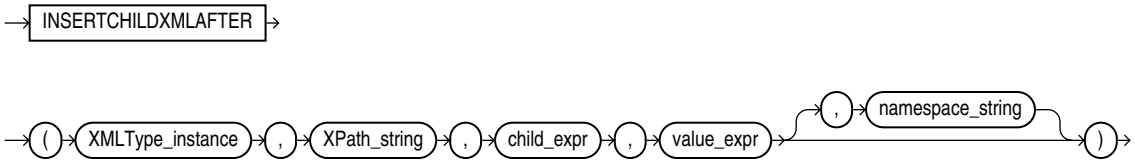
XML data *`child-data`* is inserted as one or more child elements under *each* of the parent elements located at *`parent-xpath`*.

In order of decreasing precedence, function `insertChildXMLafter` has the following behavior for `NULL` arguments:

- If *`child-name`* is `NULL`, then an error is raised.
- If *`target-data`* or *`parent-xpath`* is `NULL`, then `NULL` is returned.
- If *`child-data`* is `NULL`, then no insertion is done, and *`target-data`* is returned unchanged.

Figure 4–6 shows the syntax.

Figure 4–6 INSERTCHILDXMLAFTER Syntax



INSERTXMLBEFORE SQL Function

Oracle SQL function `insertXMLbefore` inserts one or more nodes of any kind immediately before a target node that is not an attribute node. The XML document that is the target of the insertion can be schema-based or non-schema-based.

A *copy* of the input `XMLType` instance is modified and returned. The original data is unaffected. You can then use the returned data with SQL operation `UPDATE` to modify database data.

Function `insertXMLbefore` has the following parameters (in order):

- ***target-data*** (`XMLType`) – The XML data that is the target of the insertion.
- ***successor-xpath*** (`VARCHAR2`) – An XPath 1.0 expression that locates zero or more nodes in *`target-data`* of any kind *except* attribute nodes. *`XML-data`* is inserted immediately before *each* of these nodes. Thus, the nodes in *`XML-data`* become preceding siblings of each of the *`successor-xpath`* nodes.

  If *`successor-xpath`* matches an empty sequence of nodes, then no insertion is done, and *`target-data`* is returned unchanged (no error is raised). If *`successor-xpath`* does not match a sequence of nodes that are not attribute nodes, then an error is raised.
- ***XML-data*** (`XMLType`) – The XML data to be inserted: one or more nodes of *any kind*. The order of the nodes is preserved after the insertion.
- ***namespace*** (*optional*, `VARCHAR2`) – The namespace for parameter *`successor-xpath`*.

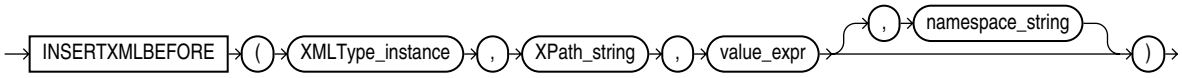
The *`XML-data`* nodes are inserted immediately before *each* of the non-attribute nodes located at *`successor-xpath`*.

Function `insertXMLbefore` has the following behavior for `NULL` arguments:

- If *`target-data`* or *`parent-xpath`* is `NULL`, then `NULL` is returned.
- Otherwise, if *`child-data`* is `NULL`, then no insertion is done, and *`target-data`* is returned unchanged.

Figure 4–7 shows the syntax.

Figure 4–7 INSERTXMLBEFORE Syntax



****

Example 4–20 Inserting a LineItem Element Before the First LineItem ELement

```
SELECT XMLQuery('$p/PurchaseOrder/LineItems/LineItem[1]'
                PASSING po.OBJECT_VALUE AS "p" RETURNING CONTENT)
  FROM purchaseorder po
  WHERE XMLExists('$p/PurchaseOrder[Reference="AMCEWEN-20021009123336171PDT"]'
                  PASSING po.OBJECT_VALUE AS "p");

XMLQUERY('$P/PURCHASEORDER/LINEITEMS/LINEITEM[1]'PASSINGPO.OBJECT_
----------------------------------------------------------------
<LineItem ItemNumber="1">
  <Description>Salesman</Description>
  <Part Id="37429158920" UnitPrice="39.95" Quantity="2"/>
</LineItem>

UPDATE purchaseorder
  SET OBJECT_VALUE =
      insertXMLbefore(OBJECT_VALUE,
```

```
                        '/PurchaseOrder/LineItems/LineItem[1]',
                        XMLType('<LineItem ItemNumber="314">
                                  <Description>Brazil</Description>
                                  <Part Id="314159265359"
                                        UnitPrice="69.95"
                                        Quantity="2"/>
                                </LineItem>'))
  WHERE XMLExists('$p/PurchaseOrder[Reference="AMCEWEN-20021009123336171PDT"]'
                  PASSING OBJECT_VALUE AS "p");

SELECT XMLQuery('$p/PurchaseOrder/LineItems/LineItem[position() <= 2]'
                PASSING po.OBJECT_VALUE AS "p" RETURNING CONTENT)
  FROM purchaseorder po
  WHERE XMLExists('$p/PurchaseOrder[Reference="AMCEWEN-20021009123336171PDT"]'
                   PASSING po.OBJECT_VALUE AS "p");

XMLQUERY('$P/PURCHASEORDER/LINEITEMS/LINEITEM[POSITION()<=2]'PASSINGPO.OBJECT_
--------------------------------------------------------------------------------
<LineItem ItemNumber="314">
  <Description>Brazil</Description>
  <Part Id="314159265359" UnitPrice="69.95" Quantity="2"/>
</LineItem>
<LineItem ItemNumber="1">
  <Description>Salesman</Description>
  <Part Id="37429158920" UnitPrice="39.95" Quantity="2"/>
</LineItem>
```

> **Note:** Queries that use Oracle SQL function `insertXMLbefore` are *not* optimized. For this reason, Oracle recommends that you use function `insertChildXML`, `insertChildXMLbefore`, or `insertChildXMLafter` instead. See "Performance Tuning for XQuery" on page 5-30.

INSERTXMLAFTER SQL Function

Oracle SQL function `insertXMLafter` inserts one or more nodes of any kind immediately after a target node that is not an attribute node. The XML document that is the target of the insertion can be schema-based or non-schema-based. It is thus similar to `insertXMLbefore`, but it inserts after, not before, the target node.

A *copy* of the input `XMLType` instance is modified and returned. The original data is unaffected. You can then use the returned data with SQL operation `UPDATE` to modify database data.

Function `insertXMLafter` has the following parameters (in order):

- ***target-data*** (`XMLType`) – The XML data that is the target of the insertion.
- ***successor-xpath*** (`VARCHAR2`) – An XPath 1.0 expression that locates zero or more nodes in *`target-data`* of any kind *except* attribute nodes. *`XML-data`* is inserted immediately after *each* of these nodes. Thus, the nodes in *`XML-data`* become succeeding siblings of each of the *`successor-xpath`* nodes.

  If *`successor-xpath`* matches an empty sequence of nodes, then no insertion is done, and *`target-data`* is returned unchanged (no error is raised). If *`successor-xpath`* does not match a sequence of nodes that are not attribute nodes, then an error is raised.
- ***XML-data*** (`XMLType`) – The XML data to be inserted: one or more nodes of *any kind*. The order of the nodes is preserved after the insertion.

- ***namespace*** (*optional*, `VARCHAR2`) – The namespace for parameter *`successor-xpath`*.

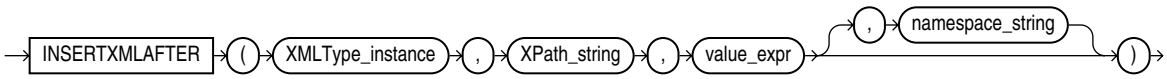
The *`XML-data`* nodes are inserted immediately after *each* of the non-attribute nodes located at *`successor-xpath`*.

Function `insertXMLafter` has the following behavior for `NULL` arguments:

- If *`target-data`* or *`parent-xpath`* is `NULL`, then `NULL` is returned.
- Otherwise, if *`child-data`* is `NULL`, then no insertion is done, and *`target-data`* is returned unchanged.

Figure 4–8 shows the syntax.

Figure 4–8 INSERTXMLAFTER Syntax



****

> **Note:** Queries that use Oracle SQL function `insertXMLafter` are *not* optimized. For this reason, Oracle recommends that you use function `insertChildXML`, `insertChildXMLbefore`, or `insertChildXMLafter` instead. See "Performance Tuning for XQuery" on page 5-30.

APPENDCHILDXML SQL Function

Oracle SQL function `appendChildXML` inserts one or more nodes of any kind as the last children of a given element node. The XML document that is the target of the insertion can be schema-based or non-schema-based.

A *copy* of the input `XMLType` instance is modified and returned. The original data is unaffected. You can then use the returned data with SQL operation `UPDATE` to modify database data.

Function `appendChildXML` has the following parameters (in order):

- ***target-data*** (`XMLType`)– The XML data containing the target parent element.
- ***parent-xpath*** (`VARCHAR2`) – An XPath 1.0 expression that locates zero or more *element* nodes in *`target-data`* that are the targets of the insertion operation. The *`child-data`* is inserted as the last child or children of *each* of these parent elements.

  If *`parent-xpath`* matches an empty sequence of element nodes, then no insertion is done, and *`target-data`* is returned unchanged (no error is raised). If *`parent-xpath`* does not match a sequence of element nodes (in particular, if *`parent-xpath`* matches one or more *attribute* or *text* nodes), then an error is raised.
- ***child-data*** (`XMLType`) – Child data to be inserted: one or more nodes of *any kind*. The order of the nodes is preserved after the insertion.
- ***namespace*** (*optional*, `VARCHAR2`) – The namespace for parameter *`parent-xpath`*.

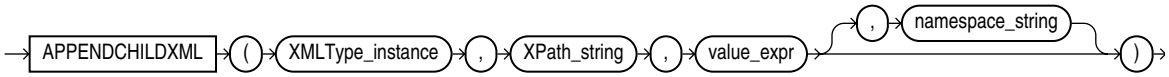
XML data *child-data* is inserted as the last child or children of *each* of the element nodes indicated by *parent-xpath*.

Function appendChildXML has the following behavior for NULL arguments:

- If *target-data* or *parent-xpath* is NULL, then NULL is returned.
- Otherwise, if *child-data* is NULL, then no insertion is done, and *target-data* is returned unchanged.

Figure 4–9 shows the syntax.

Figure 4–9 APPENDCHILDXML Syntax



Example 4–21 Inserting a Date Element as the Last Child of an Action Element

```
SELECT XMLQuery('$p/PurchaseOrder/Actions/Action[1]'
                PASSING po.OBJECT_VALUE AS "p" RETURNING CONTENT)
  FROM purchaseorder po
  WHERE XMLExists('$p/PurchaseOrder[Reference="AMCEWEN-20021009123336171PDT"]'
                  PASSING po.OBJECT_VALUE AS "p");

XMLQUERY('$P/PURCHASEORDER/ACTIONS/ACTION[1]'PASSINGPO.OBJECT_VALUE
--------------------------------------------------------------------
<Action>
  <User>KPARTNER</User>
</Action>

UPDATE purchaseorder
  SET OBJECT_VALUE =
      appendChildXML(OBJECT_VALUE,
                     'PurchaseOrder/Actions/Action[1]',
                     XMLType('<Date>2002-11-04</Date>'))
  WHERE XMLExists('$p/PurchaseOrder[Reference="AMCEWEN-20021009123336171PDT"]'
                  PASSING OBJECT_VALUE AS "p");

SELECT XMLQuery('$p/PurchaseOrder/Actions/Action[1]'
                PASSING po.OBJECT_VALUE AS "p" RETURNING CONTENT)
  FROM purchaseorder po
  WHERE XMLExists('$p/PurchaseOrder[Reference="AMCEWEN-20021009123336171PDT"]'
                  PASSING po.OBJECT_VALUE AS "p");

XMLQUERY('$P/PURCHASEORDER/ACTIONS/ACTION[1]'PASSINGPO.OBJECT_VALUE
--------------------------------------------------------------------
<Action>
  <User>KPARTNER</User>
  <Date>2002-11-04</Date>
</Action>
```

Note: Queries that use Oracle SQL function appendChildXML are *not* optimized. For this reason, Oracle recommends that you use function insertChildXML, insertChildXMLbefore, or insertChildXMLafter instead. See "Performance Tuning for XQuery" on page 5-30.

DELETEXML SQL Function

Oracle SQL function `deleteXML` deletes XML nodes of any kind. The XML document that is the target of the deletion can be schema-based or non-schema-based.

A *copy* of the input `XMLType` instance is modified and returned. The original data is unaffected. You can then use the returned data with SQL operation `UPDATE` to modify database data.

Function `deleteXML` has the following parameters (in order):

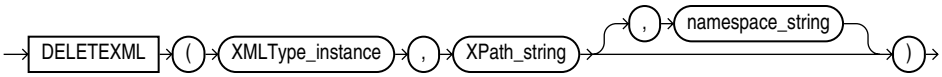
- ***target-data*** (`XMLType`) – The XML data containing the target nodes (to be deleted).
- ***xpath*** (`VARCHAR2`) – An XPath 1.0 expression that locates zero or more nodes in *`target-data`* that are the targets of the deletion operation. *Each* of these nodes is deleted.

  If *`xpath`* matches an empty sequence of nodes, then no deletion is done, and *`target-data`* is returned unchanged (no error is raised). If *`xpath`* matches the top-level element node, then an error is raised.
- ***namespace*** (*optional*, `VARCHAR2`) – The namespace for parameter *`xpath`*.

The XML nodes located at *`xpath`* are deleted from *`target-data`*. Function `deleteXML` returns `NULL` if *`target-data`* or *`xpath`* is `NULL`.

Figure 4–10 shows the syntax.

Figure 4–10 DELETEXML Syntax



Example 4–22 Deleting LineItem Element Number 222

```
SELECT XMLQuery('$p/PurchaseOrder/LineItems/LineItem[@ItemNumber=222]'
                PASSING po.OBJECT_VALUE AS "p" RETURNING CONTENT)
  FROM purchaseorder po
  WHERE XMLExists('$p/PurchaseOrder[Reference="AMCEWEN-20021009123336171PDT"]'
                  PASSING po.OBJECT_VALUE AS "p");

XMLQUERY('$P/PURCHASEORDER/LINEITEMS/LINEITEM[@ITEMNUMBER=222]'PASSINGPO
--------------------------------------------------------------------------
<LineItem ItemNumber="222">
  <Description>The Harder They Come</Description>
  <Part Id="953562951413" UnitPrice="22.95" Quantity="1"/>
</LineItem>

UPDATE purchaseorder
  SET OBJECT_VALUE =
      deleteXML(OBJECT_VALUE,
                '/PurchaseOrder/LineItems/LineItem[@ItemNumber="222"]')
  WHERE XMLExists('$p/PurchaseOrder[Reference="AMCEWEN-20021009123336171PDT"]'
                  PASSING OBJECT_VALUE AS "p");

SELECT XMLQuery('$p/PurchaseOrder/LineItems/LineItem[@ItemNumber=222]'
                PASSING po.OBJECT_VALUE AS "p" RETURNING CONTENT)
  FROM purchaseorder po
  WHERE XMLExists('$p/PurchaseOrder[Reference="AMCEWEN-20021009123336171PDT"]'
                  PASSING po.OBJECT_VALUE AS "p");
```

```
XMLQUERY('$P/PURCHASEORDER/LINEITEMS/LINEITEM[@ITEMNUMBER=222]'PASSINGPO
--------------------------------------------------------------------------

1 row selected.
```

5

Using XQuery with Oracle XML DB

This chapter describes how to use the XQuery language with Oracle XML DB. It covers Oracle XML DB support for the language, including SQL/XML functions `XMLQuery` and `XMLTable` and the SQL*Plus `XQUERY` command.

This chapter contains these topics:

- Overview of XQuery in Oracle XML DB
- Overview of the XQuery Language
- SQL/XML Functions XMLQUERY and XMLTABLE
- When To Use XQuery
- Predefined Namespaces and Prefixes
- URI Scheme oradb: Querying Table or View Data with XQuery
- Oracle XQuery Extension Functions
- XMLQUERY and XMLTABLE Examples
- Performance Tuning for XQuery
- XQuery Static Type-Checking in Oracle XML DB
- SQL*Plus XQUERY Command
- Using XQuery with PL/SQL, JDBC, and ODP.NET
- Oracle XML DB Support for XQuery

Overview of XQuery in Oracle XML DB

Oracle XML DB support for the XQuery language is provided through a native implementation of SQL/XML functions `XMLQuery` and `XMLTable`. As a convenience, SQL*Plus command `XQUERY` is also provided, which lets you enter XQuery expressions directly—in effect, this command turns SQL*Plus into an XQuery command-line interpreter.

Oracle XML DB compiles XQuery expressions that are passed as arguments to SQL/XML functions `XMLQuery`, `XMLTable`, `XMLExists`, and `XMLCast`. This compilation produces SQL query blocks and operator trees that use SQL/XML functions and XPath functions. A SQL statement that includes `XMLQuery`, `XMLTable`, `XMLExists`, or `XMLCast` is compiled and optimized as a whole, leveraging both relational database and XQuery-specific optimization technologies. Depending on the XML storage and indexing methods used, XPath functions can be further optimized. The resulting optimized operator tree is executed in a streaming fashion.

See Also:

- SQL/XML Functions XMLQUERY and XMLTABLE and SQL*Plus XQUERY Command
- Oracle XQuery Extension Functions for Oracle-specific XQuery functions that extend the language
- Oracle XML DB Support for XQuery for details on Oracle XML DB support for XQuery

Overview of the XQuery Language

Oracle XML DB supports the latest version of the XQuery language specification, W3C XQuery 1.0 Recommendation. This section presents a brief overview of the language. For more information, consult a recent book on the language or refer to the standards documents that define it, which are available at `http://www.w3c.org`.

Functional Language Based on Sequences

XQuery 1.0 is the W3C language designed for querying XML data. It is similar to SQL in many ways, but just as SQL is designed for querying structured, relational data, XQuery is designed especially for querying semi-structured, XML data from a variety of data sources. You can use XQuery to query XML data wherever it is found, whether it is stored in database tables, available through Web Services, or otherwise created on the fly. In addition to querying XML data, XQuery can be used to *construct* XML data. In this regard, XQuery can serve as an alternative or a complement to both XSLT and the other SQL/XML publishing functions, such as `XMLElement`.

XQuery builds on the Post-Schema-Validation Infoset (PSVI) data model, which unites the XML Information Set (Infoset) data model and the XML Schema type system. XQuery defines a new data model based on *sequences*: the result of *each* XQuery expression is a sequence. XQuery is all about manipulating sequences. This makes XQuery similar to a set-manipulation language, except that sequences are ordered and can contain duplicate items. XQuery sequences differ from the sequences in some other languages in that nested XQuery sequences are always *flattened* in their effect.

In many cases, sequences can be treated as unordered, to maximize optimization – where this is available, it is under your control. This **unordered mode** can be applied to join order in the treatment of nested iterations (`for`), and it can be applied to the treatment of XPath expressions (for example, in `/a/b`, the matching `b` elements can be processed without regard to document order).

An XQuery **sequence** consists of zero or more **items**, which can be either *atomic* (scalar) values or XML *nodes*. Items are typed using a rich type system that is based upon the types of XML Schema. This type system is a major change from that of XPath 1.0, which is limited to simple scalar types such as Boolean, number, and string.

XQuery is a *functional* language. As such, it consists of a set of possible *expressions* that are *evaluated* and return *values* (which, for XQuery, are sequences). As a functional language, XQuery is also **referentially transparent**, generally: the *same expression* evaluated in the *same context* returns the *same value*.

Exceptions to this desirable mathematical property include the following:

- XQuery expressions that derive their value from interaction with the external environment. For example, an expression such as `fn:current-time(...)` or `fn:doc(...)` does not necessarily always return the same value, since it depends

on external conditions that can change (the time changes; the content of the target document might change).

In some cases, like that of `fn:doc`, XQuery is defined to be referentially transparent within the execution of a single query: within a query, each invocation of `fn:doc` with the same argument results in the same document.

- XQuery expressions that are defined to be dependent on the particular XQuery language implementation. The result of evaluating such expressions might vary between implementations. Function `fn:doc` is an example of a function that is essentially implementation-defined.

Referential transparency applies also to XQuery *variables*: the same variable in the same context has the same value. Functional languages are like mathematics formalisms in this respect and unlike procedural, or imperative, programming languages. A variable in a procedural language is really a name for a memory location; it has a *current* value, or state, as represented by its content at any time. A variable in a declarative language such as XQuery is really a name for a *static* value.

XQuery Expressions

XQuery expressions are case-sensitive. The expressions include the following:

- **primary expression –** literal, variable, or function application. A variable name starts with a dollar-sign (`$`) – for example, `$foo`. Literals include numerals, strings, and character or entity references.
- **XPath expression –** Any XPath expression. The XPath 2.0 standard is a subset of XQuery.
- **FLWOR expression –** The most important XQuery expression, composed of the following, in order, from which FLWOR takes its name: `for`, `let`, `where`, `order by`, `return`.
- **XQuery sequence** – The comma (`,`) constructor creates sequences. Sequence-manipulating functions such as `union` and `intersect` are also available. All XQuery sequences are effectively **flat**: a nested sequence is treated as its flattened equivalent. Thus, for instance, `(1, 2, (3, 4, (5), 6), 7)` is treated as `(1, 2, 3, 4, 5, 6, 7)`. A singleton sequence, such as `(42)`, acts the same in most XQuery contexts as does its single item, `42`. Remember that the result of any XQuery expression is a sequence.
- **Direct (literal) constructions** – XML element and attribute syntax automatically constructs elements and attributes: what you see is what you get. For example, the XQuery expression `<a>33</a>` constructs the XML element `<a>33</a>`.
- **Computed (dynamic) constructions –** You can construct XML data at run time using computed values. For example, the following XQuery expression constructs this XML data: `<foo toto="5"><bar>tata titi</bar> why? </foo>`.

  ```
  <foo>{attribute toto {2+3}, element bar {"tata", "titi"}, text {" why? "}</foo>
  ```

  In this example, element `foo` is a direct construction; the other constructions are computed. In practice, the arguments to computed constructors are not literals (such as `toto` and `"tata"`), but expressions to be evaluated (such as `2+3`). Both the name and the value arguments of an element or attribute constructor can be computed. Braces (`{, }`) are used to mark off an XQuery expression to be evaluated.
- **Conditional expression –** As usual, but remember that each part of the expression is itself an arbitrary expression. For instance, in this conditional expression, each of

these subexpressions can be any XQuery expression: `something`, `somethingElse`, `expression1`, and `expression2`.

```
if (something < somethingElse) then expression1 else expression2
```

- **Arithmetic, relational expression –** As usual, but remember that each relational expression returns a (Boolean[1]) value. Examples:

```
2 + 3
42 < $a + 5
(1, 4) = (1, 2)
5 > 3 eq true()
```

- **Quantifier expression –** Universal (`every`) and existential (`some`) quantifier functions provide shortcuts to using a FLWOR expression in some cases. Examples:

```
every $foo in doc("bar.xml")//Whatever satisfies $foo/@bar > 42
some $toto in (42, 5), $titi in (123, 29, 5) satisfies $toto = $titi
```

- **Regular expression –** XQuery regular expressions are based on XML Schema 1.0 and Perl. (See Support for XQuery Functions and Operators on page 5-43.)

- **Type expression –** An XQuery expression that represents an XQuery type. Examples: `item()`, `node()`, `attribute()`, `element()`, `document-node()`, `namespace()`, `text()`, `xs:integer`, `xs:string`.[2]

  Type expressions can have **occurrence indicators**: **?** (optional: zero or one), ***** (zero or more), **+** (one or more). Examples: `document-node(element())*`, `item()+`, `attribute()?`.

  XQuery also provides operators for working with types. These include `cast as`, `castable as`, `treat as`, `instance of`, `typeswitch`, and `validate`. For example, `"42" cast as xs:integer` is an expression whose value is the integer 2. (It is not, strictly speaking, a type expression, because its value does not represent a type.)

FLWOR Expressions

As for XQuery in general, there is a lot to learn about FLWOR expressions. This section provides only a brief overview.

FLWOR is the most general expression syntax in XQuery. FLWOR (pronounced "flower") stands for `for`, `let`, `where`, `order by`, and `return`. A FLWOR expression has at least one `for` or `let` clause and a `return` clause; single `where` and `order by` clauses are optional.

- **`for`** – Bind one or more variables each to any number of values, in turn. That is, for each variable, iterate, binding the variable to a different value for each iteration.

  At each iteration, the variables are bound in the order they appear, so that the value of a variable `$earlier` that is listed before a variable `$later` in the `for` list, can be used in the binding of variable `$later`. For example, during its second iteration, this expression binds `$i` to `4` and `$j` to `6` (2+4):

[1] The value returned is a sequence, as always. However, in XQuery, a sequence of one item is equivalent to that item itself. In this case, the single item is a Boolean value.

[2] Namespace prefix `xs` is predefined for the XML Schema namespace, `http://www.w3.org/2001/XMLSchema`.

```
for $i in (3, 4), $j in ($i, 2+$i)
```

- **`let`** – Bind one or more variables.

  Just as with `for`, a variable can be bound by `let` to a value computed using another variable that is listed previously in the binding list of the `let` (or an enclosing `for` or `let`). For example, this expression binds `$j` to `5` (3+2):

  ```
  let $i := 3, $j := $i + 2
  ```

- **`where`** – Filter the `for` and `let` variable bindings according to some condition. This is similar to a SQL `WHERE` clause.
- **`order by`** – Sort the result of `where` filtering.
- **`return`** – Construct a result from the ordered, filtered values. This is the result of the FLWOR expression as a whole. It is a flattened sequence.

Expressions `for` and `let` act similarly to a SQL `FROM` clause. Expression `where` acts like a SQL `WHERE` clause Expression `order by` is similar to `ORDER BY` in SQL. Expression `return` is like `SELECT` in SQL. Except for the two keywords whose names are the same in both languages (`where`, `order by`), FLWOR clause order is more or less opposite to the SQL clause order, but the meanings of the corresponding clauses are quite similar.

Note that using a FLWOR expression (with `order by`) is the *only* way to construct an XQuery sequence in any order other than document order.

SQL/XML Functions XMLQUERY and XMLTABLE

SQL/XML functions `XMLQuery`, `XMLTable`, `XMLExists`, and `XMLCast` are defined by the SQL/XML standard as a general interface between the SQL and XQuery languages. As is the case for the other SQL/XML functions, these functions let you take advantage of the power and flexibility of both SQL and XML. Using these functions, you can construct XML data using relational data, query relational data as if it were XML, and construct relational data from XML data.

SQL functions `XMLExists` and `XMLCast` are documented elsewhere in this manual. This section presents functions `XMLQuery` and `XMLTable`, but many of the examples in this chapter use also `XMLExists`, and `XMLCast`. In terms of typical use:

- `XMLQuery` and `XMLCast` are typically used in a `SELECT` list.
- `XMLTable` is typically used in a SQL `FROM` clause.
- `XMLExists` is typically used in a SQL `WHERE` clause.

Both `XMLQuery` and `XMLTable` evaluate an XQuery expression. In the XQuery language, an expression always returns a sequence of items. Function `XMLQuery` aggregates the items in this sequence to return a single XML document or fragment. Function `XMLTable` returns a SQL table whose rows each contain one item from the XQuery sequence.

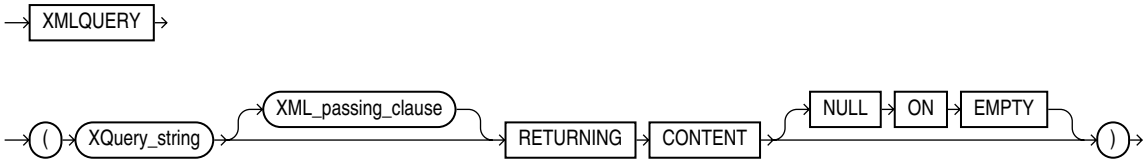
See Also:

- *Oracle Database SQL Language Reference* for information about Oracle support for the SQL/XML standard
- `http://www.sqlx.org` for information about SQL/XML functions
- `http://www.w3.org` for information about the XQuery language
- "Generating XML using SQL Functions" on page 18-2 for information about using other SQL/XML functions with Oracle XML DB
- "Querying XMLType Data using SQL/XML Functions XMLExists and XMLCast" on page 4-2

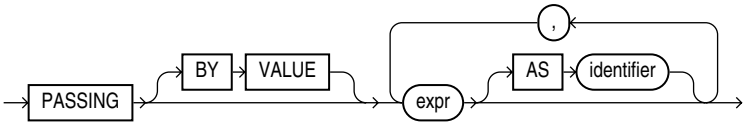
XMLQUERY SQL/XML Function in Oracle XML DB

You use SQL/XML function `XMLQuery` to construct or query XML data. This function takes as arguments an *XQuery expression*, as a string literal, and an optional XQuery *context item*, as a SQL expression. The context item establishes the XPath context in which the XQuery expression is evaluated. Additionally, `XMLQuery` accepts as arguments any number of SQL expressions whose values are bound to XQuery variables during the XQuery expression evaluation. The function returns the result of evaluating the XQuery expression, as an `XMLType` instance.

Figure 5–1 XMLQUERY Syntax



XML_passing_clause ::=



- *`XQuery_string`* is a complete XQuery expression, possibly including a prolog, as a literal string.
- The *`XML_passing_clause`* is the keyword `PASSING` followed by one or more SQL expressions (*`expr`*) that each return an `XMLType` instance or an instance of a SQL scalar data type (that is, not an object or collection data type). Each expression (*`expr`*) can be a table or view column value, a PL/SQL variable, or a bind variable with proper casting. All but possibly one of the expressions must each be followed by the keyword `AS` and an XQuery *`identifier`*. The result of evaluating each *`expr`* is bound to the corresponding *`identifier`* for the evaluation of *`XQuery_string`*. If there is an *`expr`* that is not followed by an `AS` clause, then the result of evaluating that *`expr`* is used as the *context* item for evaluating *`XQuery_string`*. Oracle XML DB supports only passing `BY VALUE`, not passing `BY REFERENCE`, so the clause `BY VALUE` is implicit and can be omitted.

- `RETURNING CONTENT` indicates that the value returned by an application of `XMLQuery` is an instance of parameterized XML type `XML(CONTENT)`, not parameterized type `XML(SEQUENCE)`. It is a document fragment that conforms to the *extended* Infoset data model. As such, it is a single document node with any number of children. The children can each be of any XML node type; in particular, they can be text nodes.

  Oracle XML DB supports only the `RETURNING CONTENT` clause of SQL/XML function `XMLQuery`; it does *not* support the `RETURNING SEQUENCE` clause.

You can pass an `XMLType` column, table, or view as the context-item argument to function `XMLQuery`—see, for example, Example 5–8.

To query a relational table or view as if it were XML data, without having to first create a SQL/XML view on top of it, use XQuery function `fn:collection` within an XQuery expression, passing as argument a URI that uses the URI-scheme name `oradb` together with the database location of the data. See "URI Scheme oradb: Querying Table or View Data with XQuery" on page 5-10.

Note: Prior to Oracle Database 11*g* Release 2, some users employed Oracle SQL functions `extract` and `extractValue` to do some of what can be done better using SQL/XML functions `XMLQuery` and `XMLCast`. SQL functions `extract` and `extractValue` are *deprecated* in Oracle Database 11*g* Release 2.

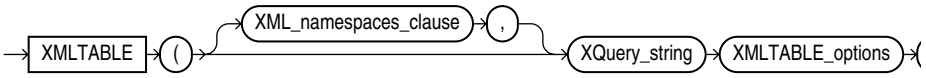
See Also:

- `http://www.sqlx.org` for information about the definition of SQL/XML function `XMLQuery`
- *Oracle Database SQL Language Reference* for reference information about SQL/XML function `XMLQuery` in Oracle Database

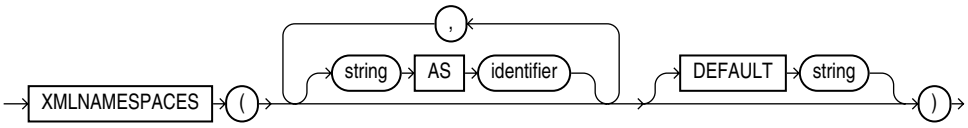
XMLTABLE SQL/XML Function in Oracle XML DB

You use SQL/XML function `XMLTable` to decompose the result of an XQuery-expression evaluation into the relational rows and columns of a new, virtual table. You can then insert the virtual table into a pre-existing database table, or you can query it using SQL—in a join expression, for example (see Example 5–9). You use `XMLTable` in a SQL `FROM` clause.

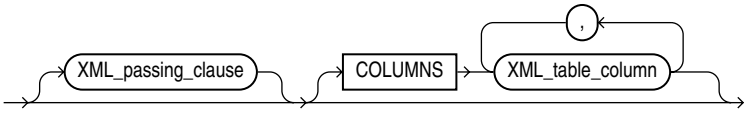
Figure 5–2 XMLTABLE Syntax



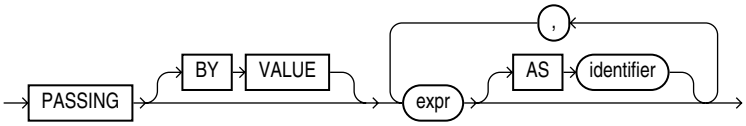
XML_namespaces_clause ::=



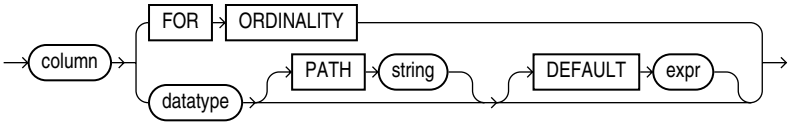
XMLTABLE_options ::=



XML_passing_clause ::=



XML_table_column ::=



- *XQuery_string* is a complete XQuery expression, possibly including a prolog, as a literal string. The value of the expression serves as input to the XMLTable function; it is this XQuery result that is decomposed and stored as relational data.
- The optional XMLNAMESPACES clause contains XML namespace declarations that are referenced by *XQuery_string* and by the XPath expression in the PATH clause of *XML_table_column*.
- The *XML_passing_clause* is the keyword PASSING followed by one or more SQL expressions (*expr*) that each return an XMLType instance or an instance of a SQL scalar data type (that is, not an object or collection data type). Each expression (*expr*) can be a table or view column value, a PL/SQL variable, or a bind variables with proper casting. All but possibly one of the expressions must each be followed by the keyword AS and an XQuery *identifier*. The result of evaluating each *expr* is bound to the corresponding *identifier* for the evaluation of *XQuery_string*. If there is an *expr* that is not followed by an AS clause, then the result of evaluating that *expr* is used as the *context* item for evaluating *XQuery_string*. Oracle XML DB supports only passing BY VALUE, not passing BY REFERENCE, so the clause BY VALUE is implicit and can be omitted.
- The optional COLUMNS clause defines the columns of the virtual table to be created by XMLTable.
  - If you omit the COLUMNS clause, then XMLTable returns a row with a single XMLType pseudo-column, named COLUMN_VALUE.
  - FOR ORDINALITY specifies that *column* is to be a column of generated row numbers (SQL data type NUMBER). There must be at most one FOR ORDINALITY clause.
  - You must specify the *datatype* of each resulting *column* except the FOR ORDINALITY column. However, see next item.
  - When XMLTable is used with XML schema-based storage of XMLType, *datatype* is optional. If not present, the data type is inferred from the XML schema. If Oracle XML DB is unable to determine the proper type for a node, a default type of VARCHAR2(4000) is used.

This is an *Oracle extension*; in the SQL/XML standard, *datatype* is always required.

Note: The inferred data type might change as the result of upgrading Oracle XML DB or applying a patch. In particular, a new release or patch set might be able to determine the data type when the previous release was unable to do so (and so defaulted to VARCHAR2(4000)). To protect against such an eventuality, specify an explicit data type with *datatype*.

- The optional PATH clause specifies that the portion of the XQuery result that is addressed by XQuery expression *string* is to be used as the *column* content. You can use multiple PATH clauses to split the XQuery result into different virtual-table columns.

  If you omit PATH, then the XQuery expression *column* is assumed. For example, these two expressions are equivalent:

  ```
  XMLTable(... COLUMNS foo)
  XMLTable(... COLUMNS foo PATH 'FOO')
  ```

  The XQuery expression *string* must represent a *relative* path; it is relative to the path *XQuery_string*.

- The optional DEFAULT clause specifies the value to use when the PATH expression results in an empty sequence (or NULL). Its *expr* is an XQuery expression that is evaluated to produce the default value.

  See Also:

  - http://www.sqlx.org for information about the definition of SQL/XML function XMLTable
  - *Oracle Database SQL Language Reference* for reference information about SQL/XML function XMLTable in Oracle Database

---

Note: Prior to Oracle Database 11*g* Release 2, some users employed Oracle SQL function XMLSequence within a SQL TABLE collection expression, that is, TABLE(XMLSequence(...)), to do some of what can be done better using SQL/XML function XMLTable. Function XMLSequence is *deprecated* in Oracle Database 11*g* Release 2.

See *Oracle Database SQL Language Reference* for information about the SQL TABLE collection expression.

When To Use XQuery

You can use XQuery to do many of the same things that you might do using the SQL/XML generation functions or XSLT; there is a great deal of overlap. The decision to use one or the other tool to accomplish a given task can be based on many considerations, most of which are not specific to Oracle Database. Please consult external documentation on this general question.

One general rule of thumb is that XQuery is often used when the focus is the world of XML data, while the SQL/XML generation functions (XMLElement, XMLAgg, and so on) are often used when the focus is the world of relational data.

Other things being equal, if a query constructs an XML document from fragments extracted from existing XML documents, then it is likely that an XQuery FLOWR expression is simpler (simplifying code maintenance) than extracting scalar values from relational data and constructing appropriate XML data using SQL/XML generation functions. If, instead, a query constructs an XML document from existing relational data, the SQL/XML generation functions can often be more suitable.

With respect to Oracle XML DB, you can expect the same general level of performance using the SQL/XML generation functions as with `XMLQuery` and `XMLTable`—all are subject to rewrite optimizations.

Predefined Namespaces and Prefixes

The following namespaces and prefixes are predefined for use with XQuery in Oracle XML DB:

Table 5–1 Predefined Namespaces and Prefixes

Prefix	Namespace	Description
`ora`	`http://xmlns.oracle.com/xdb`	Oracle XML DB namespace
`local`	`http://www.w3.org/2003/11/xpath-local-functions`	XPath local function declaration namespace
`fn`	`http://www.w3.org/2003/11/xpath-functions`	XPath function namespace
`xml`	`http://www.w3.org/XML/1998/namespace`	XML namespace
`xs`	`http://www.w3.org/2001/XMLSchema`	XML Schema namespace
`xsi`	`http://www.w3.org/2001/XMLSchema-instance`	XML Schema instance namespace

You can use these prefixes in XQuery expressions without first declaring them in the XQuery-expression prolog. You can redefine any of them *except* `xml` in the prolog. All of these prefixes except `ora` are predefined in the XQuery standard.

URI Scheme oradb: Querying Table or View Data with XQuery

You can use XQuery functions `fn:doc` and `fn:collection` to query resources in Oracle XML DB Repository—see"Querying XML Data in Oracle XML DB Repository using XQuery" on page 5-17. This section is about using XQuery function `fn:collection` to query data in database tables and views.

To do this, you pass function `fn:collection` a URI argument that specifies the table or view to query. The Oracle URI scheme `oradb` identifies this usage: without it, the argument is treated as a repository location.

The table or view that is queried can be relational or of type `XMLType`. If relational, its data is converted on the fly and treated as XML. The result returned by `fn:collection` is always an XQuery sequence.

- For an `XMLType` table, the root element of each XML document returned by `fn:collection` is the same as the root element of an XML document in the table.
- For a relational table, the root element of each XML document returned by `fn:collection` is `ROW`. The children of the `ROW` element are elements with the same names (uppercase) as columns of the table. The content of a child element corresponds to the column data. That content is an XML element if the column is

of type `XMLType`; otherwise (the column is a scalar type), the content is of type `xs:string`.

The format of the URI argument passed to `fn:collection` is as follows:

- For an `XMLType` or relational table or view, *`TABLE`*, in database schema *`DB-SCHEMA`*:

  `oradb:``/`*`DB-SCHEMA`*`/`*`TABLE`*`/`

  You can use **`PUBLIC`** for *`DB-SCHEMA`* if *`TABLE`* is a public synonym or *`TABLE`* is a table or view that is accessible to the database user currently logged in.

- For an `XMLType` column in a *relational* table or view:

  `oradb:``/`*`DB-SCHEMA`*`/`*`REL-TABLE`*`/`**`ROW`**`PRED/`*`X-COL`*

  `REL-TABLE` is a relational table or view; *`PRED`* is an XPath predicate that does not involve any `XMLType` columns; and *`X-COL`* is an `XMLType` column in *`REL-TABLE`*. *`PRED`* is optional; *`DB-SCHEMA`*, *`REL-TABLE`*, and *`X-COL`* are required.

Optional XPath predicate *`PRED`* must satisfy the following conditions:

- It does not involve any `XMLType` columns.
- It involves only conjunctions (`and`) and disjunctions (`or`) of general equality and inequality comparisons (`=`, `!=`, `>`, `<`, `>=`, and `<=`).
- For each comparison operation: Either both sides name (non-XML) columns in *`REL-TABLE`* or one side names such a column and the other is a value of the proper type, as specified in Table 5–2. Use of any other type raises an error.

Table 5–2 oradb Expressions: Column Types for Comparisons

Relational Column Type	XQuery Value Type
`VARCHAR2`, `CHAR`	`xs:string` or string literal
`NUMBER`, `FLOAT`, `BINARY_FLOAT`, `BINARY_DOUBLE`	`xs:decimal`, `xs:float`, `xs:double`, or numeric literal
`DATE`, `TIMESTAMP`, `TIMESTAMP WITH TIMEZONE`, `TIMESTAMP WITH LOCAL TIMEZONE`	`xs:date`, `xs:time`, or `xs:dateTime`
`INTERVAL YEAR TO MONTH`	`xs:yearMonthDuration`
`INTERVAL DAY TO SECOND`	`xs:dayTimeDuration`
`RAW`	`xs:hexBinary`
`ROWID`	`xs:string` or string literal

For example, this XQuery expression represents all XML documents in `XMLType` column `warehouse_spec` of table `oe.warehouses`, for the rows where column `warehouse_id` has a value less than 6:

```
fn:collection('oradb:/OE/WAREHOUSES/ROW[WAREHOUSE_ID < 6]/WAREHOUSE_SPEC')
```

See Also: "Querying Table or View Data using XQuery" on page 5-19

Oracle XQuery Extension Functions

Oracle XML DB adds some XQuery functions to those provided in the W3C standard. These additional functions are in the Oracle XML DB namespace,

`http://xmlns.oracle.com/xdb`, which uses the predefined prefix **`ora`**. This section describes these Oracle extension functions.

ora:contains XQuery Function

ora:contains Syntax

```
ora:contains (input_text, text_query [, policy_name] [, policy_owner])
```

XQuery and XPath function `ora:contains` can be used in an XQuery expression in a call to SQL/XML function `XMLQuery`, `XMLTable`, or `XMLExists`. It is used to restrict a structural search with a full-text predicate. Function `ora:contains` returns a positive integer when the *`input_text`* matches *`text_query`* (the higher the number, the more relevant the match), and zero otherwise. When used in an XQuery expression (that is not also an XPath expression), the XQuery return type is `xs:integer()`; when used in an XPath expression outside of an XQuery expression, the XPath return type is `number`.

Argument *`input_text`* must evaluate to a single text node or an attribute. The syntax and semantics of *`text_query`* in `ora:contains` are the same as *`text_query`* in `contains`, with a few restrictions.

See Also: "ora:contains XQuery Function" on page 12-17

ora:matches XQuery Function

ora:matches Syntax

```
ora:matches (target_string, match_pattern [, match_parameter])
```

XQuery function `ora:matches` lets you use a regular expression to match text in a string. It returns `true()` if its *`target_string`* argument matches its regular-expression *`match_pattern`* argument and `false()` otherwise. If *`target_string`* is the empty sequence, `false()` is returned. Optional argument *`match_parameter`* is a code that qualifies matching: case-sensitivity and so on.

The behavior of XQuery function `ora:matches` is the same as that of SQL condition `REGEXP_LIKE`, but the types of its arguments are XQuery types instead of SQL data types. The argument types are as follows:

- *`target_string`* – `xs:string?`[3]
- *`match_pattern`* – `xs:string`
- *`match_parameter`* – `xs:string`

See Also: *Oracle Database SQL Language Reference* for information about SQL condition `REGEXP_LIKE`

ora:replace XQuery Function

ora:replace Syntax

```
ora:replace (target_string, match_pattern, replace_string [, match_parameter])
```

[3] The question mark (`?`) here is a zero-or-one occurrence indicator that indicates that the argument can be the empty sequence. See "XQuery Expressions" on page 5-3.

XQuery function `ora:replace` lets you use a regular expression to replace matching text in a string. *Each* occurrence in *`target_string`* that matches regular-expression *`match_pattern`* is replaced by *`replace_string`*. It returns the new string that results from the replacement. If *`target_string`* is the empty sequence, then the empty string (`""`) is returned. Optional argument *`match_parameter`* is a code that qualifies matching: case-sensitivity and so on.

The behavior of XQuery function `ora:matches` is the same as that of SQL function `regexp_replace`, but the types of its arguments are XQuery types instead of SQL data types. The argument types are as follows:

- *`target_string`* – `xs:string?`[4]
- *`match_pattern`* – `xs:string`
- *`replace_string`* – `xs:string`
- *`match_parameter`* – `xs:string`

In addition, `ora:replace` requires argument *`replace_string`* (it is optional in `regexp_replace`) and it does not use arguments for position and number of occurrences – search starts with the first character and all occurrences are replaced.

> **See Also:** *Oracle Database SQL Language Reference* for information about SQL function `regexp_replace`

ora:sqrt XQuery Function

ora:sqrt Syntax

```
ora:sqrt (number)
```

XQuery function `ora:sqrt` returns the square root of its numeric argument, which can be of XQuery type `xs:decimal, xs:float`, or `xs:double`. The returned value is of the same XQuery type as the argument.

ora:tokenize XQuery Function

ora:tokenize Syntax

```
ora:tokenize (target_string, match_pattern [, match_parameter])
```

XQuery function `ora:tokenize` lets you use a regular expression to split the input string *`target_string`* into a sequence of strings. It treats each substring that matches the regular-expression *`match_pattern`* as a separator indicating where to split.

It returns the sequence of tokens as an XQuery value of type `xs:string*` (a sequence of `xs:string` values). If *`target_string`* is the empty sequence, it is returned. Optional argument *`match_parameter`* is a code that qualifies matching: case-sensitivity and so on.

The argument types are as follows:

- *`target_string`* – `xs:string?`[5]

[4] The question mark (`?`) here is a zero-or-one occurrence indicator that indicates that the argument can be the empty sequence. See "XQuery Expressions" on page 5-3.

[5] The question mark (`?`) here is a zero-or-one occurrence indicator that indicates that the argument can be the empty sequence. See "XQuery Expressions" on page 5-3.

- *match_pattern* – xs:string
- *match_parameter* – xs:string

Oracle XQuery Extension-Expression Pragmas

The W3C XQuery specification lets an implementation provide implementation-defined extension expressions. An XQuery extension expression is an XQuery expression that is enclosed in braces ({, }) and prefixed by an implementation-defined pragma.

The Oracle implementation provides the pragmas described in this section. No other pragmas are recognized than those listed here. Use of any other pragma, or use of any of these pragmas with incorrect pragma content (for example, (#ora:view_on_null something_else #)), raises an error.

In the ora:view_on_null examples here, assume that table null_test has columns a and b of type VARCHAR2(10), and that column b (but not a) is empty.

- (#ora:defaultTable #) – Specify the default table used to store repository data. Use this to improve the performance of repository queries that use Query function fn:doc or fn:collection. See "Using Oracle XQuery Pragma ora:defaultTable" on page 5-37.
- (#ora:invalid_path empty #) – Treat an invalid XPath expression as if its targeted nodes do not exist. For example:

```
SELECT XMLQuery('(#ora:invalid_path empty #)
                 {exists($p/PurchaseOrder//NotInTheSchema)}'
                PASSING OBJECT_VALUE AS "p" RETURNING CONTENT)
  FROM oe.purchaseorder p;
```

  The XML schema for table oe.purchaseorder does not allow any such node NotInTheSchema as a descendant of node PurchaseOrder. Without the pragma, the use of this invalid XPath expression would raise an error. But with the pragma, the calling context acts just as if the XPath expression had targeted no nodes. That calling context in this example is XQuery function exists, which returns XQuery Boolean value false when passed an empty node sequence. (XQuery function exists is used in this example only to illustrate the behavior; the pragma is not especially related to function exists.)

- (#ora:view_on_null empty #) – XQuery function fn:collection returns an empty XML element for each NULL column. For example, the following query returns <ROW><A>x</A>**<B></B>**</ROW>:

```
SELECT XMLQuery('(#ora:view_on_null empty #)
                 {for $i in fn:collection("oradb:/PUBLIC/NULL_TEST")/ROW
                  return $i}'
                RETURNING CONTENT)
  FROM DUAL;
```

- (#ora:view_on_null null #) – XQuery function fn:collection returns no element for a NULL column. For example, the following query returns <ROW><A>x</A></ROW>:

```
SELECT XMLQuery('(#ora:view_on_null null #)
                 {for $i in fn:collection("oradb:/PUBLIC/NULL_TEST")/ROW
                  return $i}'
                RETURNING CONTENT)
  FROM DUAL;
```

- `(#ora:xq_proc #)` – Do not optimize XQuery procedure calls in the XQuery expression that follows the pragma; use functional evaluation instead.

  This has the same effect as the SQL hint `/*+ NO_XML_QUERY_REWRITE */`, but the scope of the pragma is only the XQuery expression that follows it (not an entire SQL statement).

  See Also: "Turning Off Use of XMLIndex" on page 6-30 for information about optimizer hint `NO_XML_QUERY_REWRITE`

- `(#ora:xq_qry #)` – Try to optimize the XQuery expression that follows the pragma. That is, if possible, do not evaluate it functionally.

  As an example of using both `ora:xq_proc` and `ora:xq_qry`, in the following query the XQuery expression argument to `XMLQuery` will in general be evaluated functionally, but the `fn:collection` subexpressions that are preceded by pragma `ora:xq_qry` will be optimized, if possible.

```
SELECT XMLQuery('(#ora:xq_proc#)   (: Do not optimize the XQuery expression :)
                {for $i in (#ora:xq_qry#)   (: Optimize this subexpression :)
                           {fn:collection("oradb:/HR/REGIONS")},
                     $j in (#ora:xq_qry#)   (: Optimize this subexpression :)
                           {fn:collection("oradb:/HR/COUNTRIES")}
                 where $i/ROW/REGION_ID = $j/ROW/REGION_ID
                   and $i/ROW/REGION_NAME = $regionname
                 return $j}'
         PASSING CAST('&REGION' AS VARCHAR2(40)) AS "regionname"
           RETURNING CONTENT)
  AS asian_countries FROM DUAL;
```

XMLQUERY and XMLTABLE Examples

XQuery is a very general and expressive language, and SQL/XML functions `XMLQuery`, `XMLTable`, and `XMLExists` combine that power of expression and computation with the similar strengths of SQL. This section illustrates some of what you can do with these two SQL/XML functions. See "XMLEXISTS SQL/XML Function" on page 4-3 for information about `XMLExists`.

You typically use XQuery with Oracle XML DB in the following ways. The examples here are organized to reflect these different uses.

- Query XML data in Oracle XML DB Repository.

  See "Querying XML Data in Oracle XML DB Repository using XQuery".

- Query a relational table or view as if it were XML data. To do this, you use Oracle XQuery function `fn:collection`, passing as argument a URI that uses the URI-scheme name `oradb` together with the database location of the data.

  See "Querying Table or View Data using XQuery".

- Query `XMLType` relational data, possibly decomposing the resulting XML into relational data using function `XMLTable`.

  See "Using XQuery with XMLType Data".

Example 5–1 creates Oracle XML DB Repository resources that are used in some of the other examples in this chapter.

Example 5–1 Creating Resources for Examples

```
DECLARE
```

```
  res BOOLEAN;
  empsxmlstring VARCHAR2(300):=
    '<?xml version="1.0"?>
     <emps>
       <emp empno="1" deptno="10" ename="John" salary="21000"/>
       <emp empno="2" deptno="10" ename="Jack" salary="310000"/>
       <emp empno="3" deptno="20" ename="Jill" salary="100001"/>
     </emps>';
  empsxmlnsstring VARCHAR2(300):=
    '<?xml version="1.0"?>
     <emps xmlns="http://example.com">
       <emp empno="1" deptno="10" ename="John" salary="21000"/>
       <emp empno="2" deptno="10" ename="Jack" salary="310000"/>
       <emp empno="3" deptno="20" ename="Jill" salary="100001"/>
     </emps>';
  deptsxmlstring VARCHAR2(300):=
    '<?xml version="1.0"?>
     <depts>
       <dept deptno="10" dname="Administration"/>
       <dept deptno="20" dname="Marketing"/>
       <dept deptno="30" dname="Purchasing"/>
     </depts>';
BEGIN
  res := DBMS_XDB.createResource('/public/emps.xml',   empsxmlstring);
  res := DBMS_XDB.createResource('/public/empsns.xml', empsxmlnsstring);
  res := DBMS_XDB.createResource('/public/depts.xml',  deptsxmlstring);
END;
/
```

XQuery Is About Sequences

It is important to keep in mind that XQuery is a general *sequence*-manipulation language. Its expressions and their results are not necessarily XML data. An XQuery sequence can contain items of any XQuery type, which includes numbers, strings, Boolean values, dates, and various types of XML node (`document-node()`, `element()`, `attribute()`, `text()`, `namespace()`, and so on). Example 5–2 provides a sampling.

Example 5–2 XMLQuery Applied to a Sequence of Items of Different Types

```
SELECT XMLQuery('(1, 2 + 3, "a", 100 to 102, <A>33</A>)'
                RETURNING CONTENT) AS output
  FROM DUAL;

OUTPUT
--------------------------
1 5 a 100 101 102<A>33</A>

1 row selected.
```

Example 5–2 applies SQL/XML function `XMLQuery` to an XQuery sequence that contains items of several different kinds:

- an integer literal: `1`
- a arithmetic expression: `2 + 3`
- a string literal: `"a"`
- a sequence of integers: `100 to 102`

- a constructed XML element node: `<A>33</A>`

Example 5–2 also shows construction of a sequence using the comma operator (**,**) and parentheses (**(**, **)**) for grouping.

The sequence expression `100 to 102` evaluates to the sequence `(100, 101, 102)`, so the argument to `XMLQuery` here is a sequence that contains a nested sequence. The sequence argument is automatically flattened, as is always the case for XQuery sequences. The argument is, in effect, `(1, 5, "a", 100, 101, 102, <A>33</A>)`.

Querying XML Data in Oracle XML DB Repository using XQuery

This section presents examples of using XQuery with XML data in Oracle XML DB Repository. You use XQuery functions `fn:doc` and `fn:collection` to query file and folder resources in the repository, respectively. The examples in this section use XQuery function `fn:doc` to obtain a repository file that contains XML data, and then bind XQuery variables to parts of that data using `for` and `let` FLWOR-expression clauses.

> **See Also:** XQuery Functions fn:doc, fn:collection, and fn:doc-available

Example 5–3 queries two XML-document resources in Oracle XML DB Repository: `/public/emps.xml` and `/public/depts.xml`. It illustrates the use of `fn:doc` and each of the possible FLWOR-expression clauses.

Example 5–3 FLOWR Expression using for, let, order by, where, and return

```
SELECT XMLQuery('for $e in doc("/public/emps.xml")/emps/emp
                 let $d :=
                   doc("/public/depts.xml")//dept[@deptno = $e/@deptno]/@dname
                 where $e/@salary > 100000
                 order by $e/@empno
                 return <emp ename="{$e/@ename}" dept="{$d}"/>'
                RETURNING CONTENT) FROM DUAL;

XMLQUERY('FOR$EINDOC("/PUBLIC/EMPS.XML")/EMPS/EMPLET$D:=DOC("/PUBLIC/DEPTS.XML")
--------------------------------------------------------------------------------
<emp ename="Jack" dept="Administration"></emp><emp ename="Jill" dept="Marketing"
></emp>

1 row selected.
```

In Example 5–3, the various FLWOR clauses perform these operations:

- **`for`** iterates over the `emp` elements in `/public/emps.xml`, binding variable `$e` to the value of each such element, in turn. That is, it iterates over a general list of employees, binding `$e` to each employee.
- **`let`** binds variable `$d` to a *sequence* consisting of all of the values of `dname` attributes of those `dept` elements in `/public/emps.xml` whose `deptno` attributes have the same value as the `deptno` attribute of element `$e` (this is a join operation). That is, it binds `$d` to the names of all of the departments that have the same department number as the department of employee `$e`. (It so happens that the `dname` value is unique for each `deptno` value in `depts.xml`.) Note that, unlike `for`, `let` never iterates over values; `$d` is bound only once in this example.

- Together, `for` and `let` produce a stream of tuples (`$e`, `$d`), where `$e` represents an employee and `$d` represents the names of all of the departments to which that employee belongs—in this case, the unique name of the employee's unique department.
- **`where`** filters this tuple stream, keeping only tuples with employees whose salary is greater than 100,000.
- **`order by`** sorts the filtered tuple stream by employee number, `empno` (in ascending order, by default).
- **`return`** constructs `emp` elements, one for each tuple. Attributes `ename` and `dept` of these elements are constructed using attribute `ename` from the input and `$d`, respectively. Note that the element and attribute names `emp` and `ename` in the output have no necessary connection with the same names in the input document `emps.xml`.

Example 5–4 also uses each of the FLWOR-expression clauses. It shows the use of XQuery functions `doc`, `count`, `avg`, and `integer`, which are in the namespace for built-in XQuery functions, `http://www.w3.org/2003/11/xpath-functions`. This namespace is bound to the prefix `fn`.

Example 5–4 FLOWR Expression using Built-In Functions

```
SELECT XMLQuery('for $d in fn:doc("/public/depts.xml")/depts/dept/@deptno
                 let $e := fn:doc("/public/emps.xml")/emps/emp[@deptno = $d]
                 where fn:count($e) > 1
                 order by fn:avg($e/@salary) descending
                 return
                   <big-dept>{$d,
                              <headcount>{fn:count($e)}</headcount>,
                              <avgsal>{xs:integer(fn:avg($e/@salary))}</avgsal>}
                   </big-dept>'
                RETURNING CONTENT) FROM DUAL;

XMLQUERY('FOR$DINFN:DOC("/PUBLIC/DEPTS.XML")/DEPTS/DEPT/@DEPTNOLET$E:=FN:DOC("/P
--------------------------------------------------------------------------------
<big-dept deptno="10"><headcount>2</headcount><avgsal>165500</avgsal></big-dept>

1 row selected.
```

In Example 5–4, the various FLWOR clauses perform these operations:

- **`for`** iterates over `deptno` attributes in input document `/public/depts.xml`, binding variable `$d` to the value of each such attribute, in turn.
- **`let`** binds variable `$e` to a sequence consisting of all of the `emp` elements in input document `/public/emps.xml` whose `deptno` attributes have value `$d` (this is a join operation).
- Together, `for` and `let` produce a stream of tuples (`$d`, `$e`), where `$d` represents a department number and `$e` represents the set of employees in that department.
- **`where`** filters this tuple stream, keeping only tuples with more than one employee.
- **`order by`** sorts the filtered tuple stream by average salary in descending order. The average is computed by applying XQuery function `avg` (in namespace `fn`) to the values of attribute `salary`, which is attached to the `emp` elements of `$e`.
- **`return`** constructs `big-dept` elements, one for each tuple produced by `order by`. The `text()` node of `big-dept` contains the department number, bound to `$d`. A `headcount` child element contains the number of employees, bound to `$e`,

as determined by XQuery function `count`. An `avgsal` child element contains the computed average salary.

Querying Table or View Data using XQuery

This section presents examples of using XQuery to query relational data as if it were XML data.

Example 5–5 uses Oracle XQuery function `fn:collection` in a FLWOR expression to query two relational tables, `regions` and `countries`. Both tables belong to sample database schema `HR`. The example also passes scalar SQL value `Asia` to XQuery variable `$regionname`. Any SQL expression can be evaluated to produce a value passed to XQuery using `PASSING`. In this case, the value comes from a SQL*Plus variable, `REGION`. You must cast the value to the scalar SQL data type expected, in this case, `VARCHAR2(40)`.

Example 5–5 Querying Relational Tables as XML

```
DEFINE REGION = 'Asia'
SELECT XMLQuery('for $i in fn:collection("oradb:/HR/REGIONS"),
                    $j in fn:collection("oradb:/HR/COUNTRIES")
                  where $i/ROW/REGION_ID = $j/ROW/REGION_ID
                    and $i/ROW/REGION_NAME = $regionname
                  return $j'
                PASSING CAST('&REGION' AS VARCHAR2(40)) AS "regionname"
                RETURNING CONTENT) AS asian_countries
  FROM DUAL;
```

This produces the following result. (The result is shown here pretty-printed, for clarity.)

```
ASIAN_COUNTRIES
----------------------------------------
<ROW>
  <COUNTRY_ID>AU</COUNTRY_ID>
  <COUNTRY_NAME>Australia</COUNTRY_NAME>
  <REGION_ID>3</REGION_ID>
</ROW>
<ROW>
  <COUNTRY_ID>CN</COUNTRY_ID>
  <COUNTRY_NAME>China</COUNTRY_NAME>
  <REGION_ID>3</REGION_ID>
</ROW>
<ROW>
  <COUNTRY_ID>HK</COUNTRY_ID>
  <COUNTRY_NAME>HongKong</COUNTRY_NAME>
  <REGION_ID>3</REGION_ID>
</ROW>
<ROW>
  <COUNTRY_ID>IN</COUNTRY_ID>
  <COUNTRY_NAME>India</COUNTRY_NAME>
  <REGION_ID>3</REGION_ID>
</ROW>
<ROW>
  <COUNTRY_ID>JP</COUNTRY_ID>
  <COUNTRY_NAME>Japan</COUNTRY_NAME>
  <REGION_ID>3</REGION_ID>
</ROW>
<ROW>
  <COUNTRY_ID>SG</COUNTRY_ID>
```

```
  <COUNTRY_NAME>Singapore</COUNTRY_NAME>
  <REGION_ID>3</REGION_ID>
</ROW>

1 row selected.
```

In Example 5–5, the various FLWOR clauses perform these operations:

- **for** iterates over sequences of XML elements returned by calls to `fn:collection`. In the first call, each element corresponds to a row of relational table `hr.regions` and is bound to variable `$i`. Similarly, in the second call to `fn:collection`, `$j` is bound to successive rows of table `hr.countries`. Since `regions` and `countries` are not `XMLType` tables, the top-level element corresponding to a row in each table is `ROW` (a wrapper element). Iteration over the row elements is unordered.
- **where** filters the rows from both tables, keeping only those pairs of rows whose `region_id` is the same for each table (it performs a join on `region_id`) and whose `region_name` is `Asia`.
- **return** returns the filtered rows from table `hr.countries` as an XML document containing XML fragments with `ROW` as their top-level element.

Example 5–6 uses `fn:collection` within nested FLWOR expressions to query relational data.

Example 5–6 Using Relational Data in a Nested FLWOR Query

```
CONNECT hr
Enter password: password

Connected.

GRANT SELECT ON LOCATIONS TO OE
/
CONNECT oe
Enter password: password

Connected.

SELECT XMLQuery(
         'for $i in fn:collection("oradb:/OE/WAREHOUSES")/ROW
          return <Warehouse id="{$i/WAREHOUSE_ID}">
                   <Location>
                     {for $j in fn:collection("oradb:/HR/LOCATIONS")/ROW
                      where $j/LOCATION_ID eq $i/LOCATION_ID
                      return ($j/STREET_ADDRESS, $j/CITY, $j/STATE_PROVINCE)}
                   </Location>
                 </Warehouse>'
         RETURNING CONTENT) FROM DUAL;
```

This query is an example of using nested FLWOR expressions. It accesses relational table `warehouses`, which is in sample database schema `oe`, and relational table `locations`, which is in sample database schema `HR`. To run this example as user `oe`, you must first connect as user `hr` and grant permission to user `oe` to perform `SELECT` operations on table `locations`.

This produces the following result. (The result is shown here pretty-printed, for clarity.)

```
XMLQUERY('FOR$IINFN:COLLECTION("ORADB:/OE/WAREHOUSES")/ROWRETURN<WAREHOUSEID="{$
```

```
--------------------------------------------------------------------------------
<Warehouse id="1">
  <Location>
    <STREET_ADDRESS>2014 Jabberwocky Rd</STREET_ADDRESS>
    <CITY>Southlake</CITY>
    <STATE_PROVINCE>Texas</STATE_PROVINCE>
  </Location>
</Warehouse>
<Warehouse id="2">
  <Location>
    <STREET_ADDRESS>2011 Interiors Blvd</STREET_ADDRESS>
    <CITY>South San Francisco</CITY>
    <STATE_PROVINCE>California</STATE_PROVINCE>
  </Location>
</Warehouse>
<Warehouse id="3">
  <Location>
    <STREET_ADDRESS>2007 Zagora St</STREET_ADDRESS>
    <CITY>South Brunswick</CITY>
    <STATE_PROVINCE>New Jersey</STATE_PROVINCE>
  </Location>
</Warehouse>
<Warehouse id="4">
  <Location>
    <STREET_ADDRESS>2004 Charade Rd</STREET_ADDRESS>
    <CITY>Seattle</CITY>
    <STATE_PROVINCE>Washington</STATE_PROVINCE>
  </Location>
</Warehouse>
<Warehouse id="5">
  <Location>
    <STREET_ADDRESS>147 Spadina Ave</STREET_ADDRESS>
    <CITY>Toronto</CITY>
    <STATE_PROVINCE>Ontario</STATE_PROVINCE>
  </Location>
</Warehouse>
<Warehouse id="6">
  <Location>
    <STREET_ADDRESS>12-98 Victoria Street</STREET_ADDRESS>
    <CITY>Sydney</CITY>
    <STATE_PROVINCE>New South Wales</STATE_PROVINCE>
  </Location>
</Warehouse>
<Warehouse id="7">
  <Location>
    <STREET_ADDRESS>Mariano Escobedo 9991</STREET_ADDRESS>
    <CITY>Mexico City</CITY>
    <STATE_PROVINCE>Distrito Federal,</STATE_PROVINCE>
  </Location>
</Warehouse>
<Warehouse id="8">
  <Location>
    <STREET_ADDRESS>40-5-12 Laogianggen</STREET_ADDRESS>
    <CITY>Beijing</CITY>
  </Location>
</Warehouse>
<Warehouse id="9">
  <Location>
    <STREET_ADDRESS>1298 Vileparle (E)</STREET_ADDRESS>
    <CITY>Bombay</CITY>
```

```
    <STATE_PROVINCE>Maharashtra</STATE_PROVINCE>
  </Location>
</Warehouse>

1 row selected.
```

In Example 5–6, the various FLWOR clauses perform these operations:

- The outer **for** iterates over the sequence of XML elements returned by `fn:collection`: each element corresponds to a row of relational table `oe.warehouses` and is bound to variable `$i`. Since `warehouses` is not an `XMLType` table, the top-level element corresponding to a row is `ROW`. The iteration over the row elements is unordered.
- The inner **for** iterates, similarly, over a sequence of XML elements returned by `fn:collection`: each element corresponds to a row of relational table `hr.locations` and is bound to variable `$j`.
- **where** filters the tuples (`$i`, `$j`), keeping only those whose `location_id` child is the same for `$i` and `$j` (it performs a join on `location_id`).
- The inner **return** constructs an XQuery sequence of elements `STREET_ADDRESS`, `CITY`, and `STATE_PROVINCE`, all of which are children of locations-table `ROW` element `$j`; that is, they are the values of the locations-table columns of the same name.
- The outer **return** wraps the result of the inner `return` in a `Location` element, and wraps that in a `Warehouse` element. It provides the `Warehouse` element with an `id` attribute whose value comes from the `warehouse_id` column of table `warehouses`.

See Also: Example 5–15 for the execution plan of Example 5–6

Example 5–7 uses SQL/XML function `XMLTable` to decompose the result of an XQuery query to produce virtual relational data. The XQuery expression used in this example is identical to the one used in Example 5–6; the result of evaluating the XQuery expression is a sequence of `Warehouse` elements. Function `XMLTable` produces a virtual relational table whose rows are those `Warehouse` elements. More precisely, the value of pseudocolumn `COLUMN_VALUE` for each virtual-table row is an XML fragment (of type `XMLType`) with a single `Warehouse` element.

Example 5–7 Querying a Relational Table as XML using XMLTable

```
SELECT *
  FROM XMLTable(
         'for $i in fn:collection("oradb:/OE/WAREHOUSES")/ROW
          return <Warehouse id="{$i/WAREHOUSE_ID}">
                   <Location>
                     {for $j in fn:collection("oradb:/HR/LOCATIONS")/ROW
                      where $j/LOCATION_ID eq $i/LOCATION_ID
                      return ($j/STREET_ADDRESS, $j/CITY, $j/STATE_PROVINCE)}
                   </Location>
                 </Warehouse>');
```

This produces the same result as Example 5–6, except that each `Warehouse` element is output as a separate row, instead of all `Warehouse` elements being output together in a single row.

```
COLUMN_VALUE
-------------------------------------------------------
```

```
<Warehouse id="1">
  <Location>
    <STREET_ADDRESS>2014 Jabberwocky Rd</STREET_ADDRESS>
    <CITY>Southlake</CITY>
    <STATE_PROVINCE>Texas</STATE_PROVINCE>
  </Location>
</Warehouse>
<Warehouse id="2">
  <Location>
    <STREET_ADDRESS>2011 Interiors Blvd</STREET_ADDRESS>
    <CITY>South San Francisco</CITY>
    <STATE_PROVINCE>California</STATE_PROVINCE>
  </Location>
</Warehouse>
. . .

9 rows selected.
```

See Also: Example 5–16 for the execution plan of Example 5–7

Using XQuery with XMLType Data

This section presents examples of using XQuery with `XMLType` relational data.

The query in Example 5–8 passes an `XMLType` column, `warehouse_spec`, as *context* item to XQuery, using function `XMLQuery` with the `PASSING` clause. It constructs a `Details` element for each of the warehouses whose area is greater than 80,000: `/Warehouse/ Area > 80000`.

Example 5–8 Querying an XMLType Column using XMLQuery PASSING Clause

```
SELECT warehouse_name,
       XMLQuery(
         'for $i in /Warehouse
          where  $i/Area > 80000
          return <Details>
                   <Docks num="{$i/Docks}"/>
                   <Rail>{if ($i/RailAccess = "Y") then "true" else "false"}
                   </Rail>
                 </Details>'
         PASSING warehouse_spec RETURNING CONTENT) big_warehouses
  FROM oe.warehouses;
```

This produces the following output:

```
WAREHOUSE_NAME
--------------
BIG_WAREHOUSES
--------------
Southlake, Texas


San Francisco


New Jersey
<Details><Docks num=""></Docks><Rail>false</Rail></Details>

Seattle, Washington
<Details><Docks num="3"></Docks><Rail>true</Rail></Details>
```

```
Toronto


Sydney


Mexico City


Beijing


Bombay


9 rows selected.
```

In Example 5–8, function `XMLQuery` is applied to the `warehouse_spec` column in each row of table `warehouses`. The various FLWOR clauses perform these operations:

- **`for`** iterates over the `Warehouse` elements in each row of column `warehouse_spec` (the passed context item): each such element is bound to variable `$i`, in turn. The iteration is unordered.
- **`where`** filters the `Warehouse` elements, keeping only those whose `Area` child has a value greater than 80,000.
- **`return`** constructs an XQuery sequence of `Details` elements, each of which contains a `Docks` and a `Rail` child elements. The `num` attribute of the constructed `Docks` element is set to the `text()` value of the `Docks` child of `Warehouse`. The `text()` content of `Rail` is set to `true` or `false`, depending on the value of the `RailAccess` attribute of element `Warehouse`.

The `SELECT` statement in Example 5–8 applies to each row in table `warehouses`. The `XMLQuery` expression returns the *empty sequence* for those rows that do not match the XQuery expression. Only the warehouses in New Jersey and Seattle satisfy the XQuery query, so they are the only warehouses for which `<Details>...</Details>` is returned.

Example 5–9 uses SQL/XML function `XMLTable` to query an `XMLType` table, `oe.purchaseorder`, which contains XML Schema-based data. It uses the `PASSING` clause to provide the `purchaseorder` table as the context item for the XQuery-expression argument to `XMLTable`. Pseudocolumn `COLUMN_VALUE` of the resulting virtual table holds a constructed element, `A10po`, which contains the `Reference` information for those purchase orders whose `CostCenter` element has value `A10` and whose `User` element has value `SMCCAIN`. The query performs a join between the virtual table and database table `purchaseorder`.

Example 5–9 Using XMLTABLE with XML Schema-Based Data

```
SELECT xtab.COLUMN_VALUE
  FROM purchaseorder, XMLTable('for $i in /PurchaseOrder
                                where $i/CostCenter eq "A10"
                                  and $i/User eq "SMCCAIN"
                                return <A10po pono="{$i/Reference}"/>'
                               PASSING OBJECT_VALUE) xtab;

COLUMN_VALUE
----------------------------------------------------
<A10po pono="SMCCAIN-20021009123336151PDT"></A10po>
```

```
<A10po pono="SMCCAIN-20021009123336341PDT"></A10po>
<A10po pono="SMCCAIN-20021009123337173PDT"></A10po>
<A10po pono="SMCCAIN-20021009123335681PDT"></A10po>
<A10po pono="SMCCAIN-20021009123335470PDT"></A10po>
<A10po pono="SMCCAIN-20021009123336972PDT"></A10po>
<A10po pono="SMCCAIN-20021009123336842PDT"></A10po>
<A10po pono="SMCCAIN-20021009123336512PDT"></A10po>
<A10po pono="SMCCAIN-200210091233894PDT"></A10po>
<A10po pono="SMCCAIN-20021009123337403PDT"></A10po>

10 rows selected.
```

The `PASSING` clause of function `XMLTable` passes the `OBJECT_VALUE` of `XMLType` table `purchaseorder`, to serve as the XPath context. The `XMLTable` expression thus *depends* on the `purchaseorder` table. Because of this, table `purchaseorder` must appear *before* the `XMLTable` expression in the `FROM` list. This is a general requirement in any situation involving data dependence.

> **Note:** Whenever a `PASSING` clause refers to a column of an `XMLType` table in a query, that table ***must appear before*** the `XMLTable` expression in the query `FROM` list. This is because the `XMLTable` expression *depends* on the `XMLType` table—a *left lateral* (correlated) join is needed, to ensure a one-to-many (1:N) relationship between the `XMLType` table row accessed and the rows generated from it by `XMLTable`.

Example 5–10 is similar to Example 5–9 in its effect. It uses `XMLQuery`, instead of `XMLTable`, to query `oe.purchaseorder`. These two examples differ in their treatment of the empty sequences returned by the XQuery expression. In Example 5–9, these empty sequences are not joined with the `purchaseorder` table, so the overall SQL-query result set has only ten rows. In Example 5–10, these empty sequences are part of the overall result set of the SQL query, which contains 132 rows, one for each of the rows in table `purchaseorder`. All but ten of those rows are empty, and show up in the output as empty lines. To save space here, those empty lines have been removed.

Example 5–10 Using XMLQUERY with Schema-Based Data

```
SELECT XMLQuery('for $i in /PurchaseOrder
                 where $i/CostCenter eq "A10"
                   and $i/User eq "SMCCAIN"
                 return <A10po pono="{$i/Reference}"/>'
                PASSING OBJECT_VALUE
                RETURNING CONTENT)
  FROM purchaseorder;

XMLQUERY('FOR$IIN/PURCHASEORDERWHERE$I/COSTCENTEREQ"A10"AND$I/USEREQ"SMCCAIN"RET
--------------------------------------------------------------------------------
<A10po pono="SMCCAIN-20021009123336151PDT"></A10po>
<A10po pono="SMCCAIN-20021009123336341PDT"></A10po>
<A10po pono="SMCCAIN-20021009123337173PDT"></A10po>
<A10po pono="SMCCAIN-20021009123335681PDT"></A10po>
<A10po pono="SMCCAIN-20021009123335470PDT"></A10po>
<A10po pono="SMCCAIN-20021009123336972PDT"></A10po>
<A10po pono="SMCCAIN-20021009123336842PDT"></A10po>
<A10po pono="SMCCAIN-20021009123336512PDT"></A10po>
<A10po pono="SMCCAIN-200210091233894PDT"></A10po>
```

```
<A10po pono="SMCCAIN-20021009123337403PDT"></A10po>

132 rows selected.
```

See Also: Example 5–17 for the execution plan of Example 5–10

Example 5–11 uses `XMLTable` clauses `PASSING` and `COLUMNS`. The XQuery expression iterates over top-level `PurchaseOrder` elements, constructing a `PO` element for each purchase order with cost center `A10`. The resulting `PO` elements are then passed to `XMLTable` for processing.

Example 5–11 Using XMLTABLE with PASSING and COLUMNS Clauses

```
SELECT xtab.poref, xtab.priority, xtab.contact
  FROM purchaseorder,
       XMLTable('for $i in /PurchaseOrder
                 let $spl := $i/SpecialInstructions
                 where $i/CostCenter eq "A10"
                 return <PO>
                          <Ref>{$i/Reference}</Ref>
                          {if ($spl eq "Next Day Air" or $spl eq "Expedite") then
                             <Type>Fastest</Type>
                           else if ($spl eq "Air Mail") then
                             <Type>Fast</Type>
                           else ()}
                          <Name>{$i/Requestor}</Name>
                        </PO>'
                PASSING OBJECT_VALUE
                COLUMNS poref    VARCHAR2(20) PATH 'Ref',
                        priority VARCHAR2(8)  PATH 'Type' DEFAULT 'Regular',
                        contact  VARCHAR2(20) PATH 'Name') xtab;

POREF                PRIORITY CONTACT
-------------------- -------- --------------------
SKING-20021009123336 Fastest  Steven A. King
SMCCAIN-200210091233 Regular  Samuel B. McCain
SMCCAIN-200210091233 Fastest  Samuel B. McCain
JCHEN-20021009123337 Fastest  John Z. Chen
JCHEN-20021009123337 Regular  John Z. Chen
SKING-20021009123337 Regular  Steven A. King
SMCCAIN-200210091233 Regular  Samuel B. McCain
JCHEN-20021009123338 Regular  John Z. Chen
SMCCAIN-200210091233 Regular  Samuel B. McCain
SKING-20021009123335 Regular  Steven X. King
SMCCAIN-200210091233 Regular  Samuel B. McCain
SKING-20021009123336 Regular  Steven A. King
SMCCAIN-200210091233 Fast     Samuel B. McCain
SKING-20021009123336 Fastest  Steven A. King
SKING-20021009123336 Fastest  Steven A. King
SMCCAIN-200210091233 Regular  Samuel B. McCain
JCHEN-20021009123335 Regular  John Z. Chen
SKING-20021009123336 Regular  Steven A. King
JCHEN-20021009123336 Regular  John Z. Chen
SKING-20021009123336 Regular  Steven A. King
SMCCAIN-200210091233 Regular  Samuel B. McCain
SKING-20021009123337 Regular  Steven A. King
SKING-20021009123338 Fastest  Steven A. King
SMCCAIN-200210091233 Regular  Samuel B. McCain
JCHEN-20021009123337 Regular  John Z. Chen
JCHEN-20021009123337 Regular  John Z. Chen
```

```
JCHEN-20021009123337 Regular  John Z. Chen
SKING-20021009123337 Regular  Steven A. King
JCHEN-20021009123337 Regular  John Z. Chen
SKING-20021009123337 Regular  Steven A. King
SKING-20021009123337 Regular  Steven A. King
SMCCAIN-200210091233 Fast     Samuel B. McCain

32 rows selected.
```

In Example 5–11, data from the children of `PurchaseOrder` is used to construct the children of `PO`, which are `Ref`, `Type`, and `Name`. The content of `Type` is taken from the content of `/PurchaseOrder/SpecialInstructions`, but the classes of `SpecialInstructions` are divided up differently for `Type`.

Function `XMLTable` breaks up the result of XQuery evaluation, returning it as three `VARCHAR2` columns of a virtual table: `poref`, `priority`, and `contact`. The `DEFAULT` clause is used to supply a default `priority` of `Regular`.

In Example 5–12, SQL/XML function `XMLTable` is used to break up the XML data in an `XMLType` collection element, `LineItem`, into separate columns of a virtual table.

Example 5–12 Decomposing XML Collection Elements into Relational Data using XMLTABLE

```
SELECT lines.lineitem, lines.description, lines.partid,
       lines.unitprice, lines.quantity
  FROM purchaseorder,
       XMLTable('for $i in /PurchaseOrder/LineItems/LineItem
                 where $i/@ItemNumber >= 8
                  and $i/Part/@UnitPrice > 50
                  and $i/Part/@Quantity > 2
                 return $i'
                PASSING OBJECT_VALUE
                COLUMNS lineitem    NUMBER       PATH '@ItemNumber',
                        description VARCHAR2(30) PATH 'Description',
                        partid      NUMBER       PATH 'Part/@Id',
                        unitprice   NUMBER       PATH 'Part/@UnitPrice',
                        quantity    NUMBER       PATH 'Part/@Quantity') lines;

LINEITEM DESCRIPTION                          PARTID UNITPRICE QUANTITY
-------- ----------------------------- ------------- --------- --------
      11 Orphic Trilogy                  37429148327        80        3
      22 Dreyer Box Set                  37429158425        80        4
      11 Dreyer Box Set                  37429158425        80        3
      16 Dreyer Box Set                  37429158425        80        3
       8 Dreyer Box Set                  37429158425        80        3
      12 Brazil                          37429138526        60        3
      18 Eisenstein: The Sound Years     37429149126        80        4
      24 Dreyer Box Set                  37429158425        80        3
      14 Dreyer Box Set                  37429158425        80        4
      10 Brazil                          37429138526        60        3
      17 Eisenstein: The Sound Years     37429149126        80        3
      16 Orphic Trilogy                  37429148327        80        4
      13 Orphic Trilogy                  37429148327        80        4
      10 Brazil                          37429138526        60        4
      12 Eisenstein: The Sound Years     37429149126        80        3
      12 Dreyer Box Set                  37429158425        80        4
      13 Dreyer Box Set                  37429158425        80        4

17 rows selected.
```

See Also:

- Example 5–18 for the execution plan of Example 5–12
- "Breaking Up Multiple Levels of XML Data" on page 3-49, for an example of applying `XMLTable` to multiple document levels (multilevel chaining)

Using Namespaces with XQuery

You can use the XQuery `declare namespace` declaration in the prolog of an XQuery expression to define a namespace prefix. You can use `declare` **`default`** `namespace` to establish the namespace as the default namespace for the expression.

Be aware of the following pitfall, if you use SQL*Plus: If the semicolon (`;`) at the end of a namespace declaration terminates a line, SQL*Plus interprets it as a SQL terminator. To avoid this, you can do one of the following:

- Place the text that follows the semicolon on the same line.
- Place a comment, such as **`(: :)`**, after the semicolon, on the same line.
- Turn off the recognition of the SQL terminator with SQL*Plus command `SET SQLTERMINATOR`.

Example 5–13 illustrates use of a namespace declaration in an XQuery expression.

Example 5–13 Using XMLQUERY with a Namespace Declaration

```
SELECT XMLQuery('declare namespace e = "http://example.com";
ERROR:
ORA-01756: quoted string not properly terminated

                 for $i in doc("/public/empsns.xml")/e:emps/e:emp
SP2-0734: unknown command beginning "for $i in ..." - rest of line ignored.
...

-- This works - do not end the line with ";".
SELECT XMLQuery('declare namespace e = "http://example.com"; for
                     $i in doc("/public/empsns.xml")/e:emps/e:emp
                 let $d :=
                   doc("/public/depts.xml")//dept[deptno=$i/@deptno]/@dname
                 where $i/@salary > 100000
                 order by $i/@empno
                 return <emp ename="{$i/@ename}" dept="{$d}"/>'
                RETURNING CONTENT) FROM DUAL;

XMLQUERY('DECLARENAMESPACEE="HTTP://EXAMPLE.COM";FOR$IINDOC("/PUBLIC/EMPSNS.XML"
--------------------------------------------------------------------------------
<emp ename="Jack" dept=""></emp><emp ename="Jill" dept=""></emp>

-- This works too - add a comment after the ";".
SELECT XMLQuery('declare namespace e = "http://example.com";  (: :)
                 for $i in doc("/public/empsns.xml")/e:emps/e:emp
                 let $d := doc("/public/depts.xml")//dept[deptno=$i/@deptno]/@dname
                 where $i/@salary > 100000
                 order by $i/@empno
                 return <emp ename="{$i/@ename}" dept="{$d}"/>'
                RETURNING CONTENT) FROM DUAL;

XMLQUERY('DECLARENAMESPACEE="HTTP://EXAMPLE.COM";(::)FOR$IINDOC("/PUBLIC/EMPSNS.
--------------------------------------------------------------------------------
```

```
<emp ename="Jack" dept=""></emp><emp ename="Jill" dept=""></emp>

1 row selected.

-- This works too - tell SQL*Plus to ignore the ";".
SET SQLTERMINATOR OFF

SELECT XMLQuery('declare namespace e = "http://example.com";
                 for $i in doc("/public/empsns.xml")/e:emps/e:emp
                 let $d :=
                   doc("/public/depts.xml")//dept[deptno=$i/@deptno]/@dname
                 where $i/@salary > 100000
                 order by $i/@empno
                 return <emp ename="{$i/@ename}" dept="{$d}"/>'
                RETURNING CONTENT) FROM DUAL
/

XMLQUERY('DECLARENAMESPACEE="HTTP://EXAMPLE.COM";FOR$IINDOC("/PUBLIC/EMPSNS.XML"
--------------------------------------------------------------------------------
<emp ename="Jack" dept=""></emp><emp ename="Jill" dept=""></emp>
```

An XQuery namespace declaration has no effect outside of its XQuery expression. To declare a namespace prefix for use in an `XMLTable` expression outside of the XQuery expression, use the `XMLNAMESPACES` clause. This clause also covers the XQuery expression argument to `XMLTable`, eliminating the need for a separate declaration in the XQuery prolog.

In Example 5–14, `XMLNAMESPACES` is used to define the prefix `e` for the namespace `http://example.com`. This namespace is used in the `COLUMNS` clause and the XQuery expression of the `XMLTable` expression.

Example 5–14 Using XMLTABLE with the XMLNAMESPACES Clause

```
SELECT * FROM XMLTable(XMLNAMESPACES('http://example.com' AS "e"),
                       'for $i in doc("/public/empsns.xml")
                        return $i/e:emps/e:emp'
                       COLUMNS name VARCHAR2(6) PATH '@ename',
                               id   NUMBER      PATH '@empno');
```

This produces the following result:

```
NAME           ID
------ ----------
John            1
Jack            2
Jill            3

3 rows selected.
```

It is the presence of qualified names `e:ename` and `e:empno` in the `COLUMNS` clause that necessitates using the `XMLNAMESPACES` clause. Otherwise, a prolog namespace declaration (`declare namespace e = "http://example.com"`) would suffice for the XQuery expression itself.

Because the same namespace is used throughout the `XMLTable` expression, a default namespace could be used: `XMLNAMESPACES (DEFAULT 'http://example.com')`. The qualified name `$i/e:emps/e:emp` could then be written without an explicit prefix: `$i/emps/emp`.

Performance Tuning for XQuery

A SQL query that involves XQuery expressions can often be rewritten (optimized) in one or more ways. This optimization is referred to as **XML query rewrite** or optimization. XPath expressions are a proper subset of XQuery expressions.

XPath rewrite is a subset of XML query rewrite that involves rewriting queries that involve XPath expressions. XPath rewrite includes `XMLIndex` optimizations, streaming evaluation of binary XML, and rewrite to underlying object-relational or relational structures in the case of structured storage or `XMLType` views over relational data.

Just as query tuning can improve SQL performance, so it can improve XQuery performance. You tune XQuery performance by choosing appropriate XML storage models and indexes.

As with database queries generally, you determine whether tuning is required by examining the execution plan for a query. If the plan is not optimal, then consult the following documentation for specific tuning information:

- For structured storage: Chapter 8, "XPath Rewrite for Structured Storage"
- For unstructured storage and binary XML storage: Chapter 6, "Indexing XMLType Data"

In addition, be aware that the following expressions can be expensive to process, so they might add performance overhead when processing large volumes of data:

- SQL expressions that use the following Oracle SQL functions, which accept XPath expression arguments:
  - `appendChildXML` (use `insertChildXMLAfter` instead)
  - `insertXMLAfter` (use `insertChildXMLAfter` instead)
  - `insertXMLBefore` (use `insertChildXMLBefore` instead)
- XQuery expressions that use the following axes (use forward and descendent axes instead):
  - `ancestor`
  - `ancestor-or-self`
  - `descendant-or-self`
  - `following`
  - `following-sibling`
  - `namespace`
  - `parent`
  - `preceding`
  - `preceding-sibling`
- XQuery expressions that involve node identity (for example, using the order-comparison operators << and >>)

See Also: "Oracle XML DB Support for XQuery" on page 5-42

The following sections present the execution plans for some of the examples shown earlier in this chapter, to indicate how they are executed.

- "XQuery Optimization over Relational Data" on page 5-31: examples with XQuery expressions that target XML data created on the fly using `fn:collection` together with URI scheme `oradb`.
- "XQuery Optimization over XML Schema-Based XMLType Data" on page 5-32: examples with XQuery expressions that target an XML schema-based `XMLType` table stored object-relationally

See Also:

- Chapter 8, "XPath Rewrite for Structured Storage"
- Chapter 6, "Indexing XMLType Data" for information about using `XMLIndex`
- "How Oracle XML DB Processes XMLType Methods and SQL Functions" on page 3-58 for information about streaming evaluation of binary XML data

Rule-Based and Cost-Based XQuery Optimization

Several competing optimization possibilities can exist for queries with XQuery expressions, depending on various factors such as the `XMLType` storage model and indexing that are used.

By default, Oracle XML DB follows a prioritized set of rules to determine which of the possible optimizations should be used for any given query and context. This behavior is referred to as **rule-based** XML query rewrite.

Alternatively, Oracle XML DB can use **cost-based** XML query rewrite. In this mode, Oracle XML DB estimates the performance of the various XML optimization possibilities for a given query and chooses the combination that is expected to be most performant.

You can impose cost-based optimization for a given SQL statement by using the optimizer hint `/*+ COST_XML_QUERY_REWRITE */`.

XQuery Optimization over Relational Data

Example 5–15 shows the optimization of `XMLQuery` over relational data accessed as XML. Example 5–16 shows the optimization of `XMLTable` in the same context.

Example 5–15 Optimization of XMLQuery over Relational Data

Here again is the query of Example 5–6, together with its execution plan, which shows that the query has been optimized.

```
SELECT XMLQuery(
         'for $i in fn:collection("oradb:/OE/WAREHOUSES")/ROW
          return <Warehouse id="{$i/WAREHOUSE_ID}">
                   <Location>
                     {for $j in fn:collection("oradb:/HR/LOCATIONS")/ROW
                      where $j/LOCATION_ID eq $i/LOCATION_ID
                      return ($j/STREET_ADDRESS, $j/CITY, $j/STATE_PROVINCE)}
                   </Location>
                 </Warehouse>'
         RETURNING CONTENT) FROM DUAL;

PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------------------
Plan hash value: 3341889589
```

```
-----------------------------------------------------------------------------------------
| Id  | Operation                    | Name       | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |            |     1 |       |     2   (0)| 00:00:01 |
|   1 |  SORT AGGREGATE              |            |     1 |    41 |            |          |
|   2 |   TABLE ACCESS BY INDEX ROWID| LOCATIONS  |     1 |    41 |     1   (0)| 00:00:01 |
|*  3 |    INDEX UNIQUE SCAN         | LOC_ID_PK  |     1 |       |     0   (0)| 00:00:01 |
|   4 |  SORT AGGREGATE              |            |     1 |     6 |            |          |
|   5 |   TABLE ACCESS FULL          | WAREHOUSES |     9 |    54 |     2   (0)| 00:00:01 |
|   6 |  FAST DUAL                   |            |     1 |       |     2   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - access("LOCATION_ID"=:B1)

18 rows selected.
```

Example 5–16 Optimization of XMLTable over Relational Data

Here again is the query of Example 5–7, together with its execution plan, which shows that the query has been optimized.

```
SELECT *
  FROM XMLTable(
         'for $i in fn:collection("oradb:/OE/WAREHOUSES")/ROW
          return <Warehouse id="{$i/WAREHOUSE_ID}">
                   <Location>
                     {for $j in fn:collection("oradb:/HR/LOCATIONS")/ROW
                      where $j/LOCATION_ID eq $i/LOCATION_ID
                      return ($j/STREET_ADDRESS, $j/CITY, $j/STATE_PROVINCE)}
                   </Location>
                 </Warehouse>');
```

```
PLAN_TABLE_OUTPUT
-----------------------------------------------------------------------------------------
Plan hash value: 1021775546

-----------------------------------------------------------------------------------------
| Id  | Operation                    | Name       | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |            |     9 |    54 |     2   (0)| 00:00:01 |
|   1 |  SORT AGGREGATE              |            |     1 |    41 |            |          |
|   2 |   TABLE ACCESS BY INDEX ROWID| LOCATIONS  |     1 |    41 |     1   (0)| 00:00:01 |
|*  3 |    INDEX UNIQUE SCAN         | LOC_ID_PK  |     1 |       |     0   (0)| 00:00:01 |
|   4 |  TABLE ACCESS FULL           | WAREHOUSES |     9 |    54 |     2   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - access("LOCATION_ID"=:B1)

16 rows selected.
```

XQuery Optimization over XML Schema-Based XMLType Data

Example 5–17 shows the optimization of `XMLQuery` over an XML schema-based `XMLType` table. Example 5–18 shows the optimization of `XMLTable` in the same context.

Example 5–17 Optimization of XMLQuery with Schema-Based XMLType Data

Here again is the query of Example 5–10, together with its execution plan, which shows that the query has been optimized.

```
SELECT XMLQuery('for $i in /PurchaseOrder
                 where $i/CostCenter eq "A10"
                   and $i/User eq "SMCCAIN"
                 return <A10po pono="{$i/Reference}"/>'
                PASSING OBJECT_VALUE
                RETURNING CONTENT)
  FROM purchaseorder;

PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------
Plan hash value: 3611789148

--------------------------------------------------------------------------------
| Id  | Operation           | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |               |     1 |   530 |     5   (0)| 00:00:01 |
|   1 |  SORT AGGREGATE     |               |     1 |       |            |          |
|*  2 |   FILTER            |               |       |       |            |          |
|   3 |    FAST DUAL        |               |     1 |       |     2   (0)| 00:00:01 |
|*  4 |    TABLE ACCESS FULL| PURCHASEORDER |     1 |   530 |     5   (0)| 00:00:01 |
--------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - filter(:B1='SMCCAIN' AND :B2='A10')
   4 - filter(SYS_CHECKACL("ACLOID","OWNERID",xmltype('<privilege
             xmlns="http://xmlns.oracle.com/xdb/acl.xsd"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://xmlns.oracle.com/xdb/acl.xsd
             http://xmlns.oracle.com/xdb/acl.xsd DAV:http://xmlns.oracle.com/xdb/dav.xsd">
             <read-properties/><read-contents/></privilege>'))=1)

22 rows selected.
```

Example 5–18 Optimization of XMLTable with Schema-Based XMLType Data

Here again is the query of Example 5–12, together with its execution plan, which shows that the query has been optimized. The XQuery result is never materialized. Instead, the underlying storage columns for the XML collection element `LineItem` are used to generate the overall result set.

```
SELECT lines.lineitem, lines.description, lines.partid,
       lines.unitprice, lines.quantity
  FROM purchaseorder,
       XMLTable('for $i in /PurchaseOrder/LineItems/LineItem
                 where $i/@ItemNumber >= 8
                  and $i/Part/@UnitPrice > 50
                  and $i/Part/@Quantity > 2
                 return $i'
                PASSING OBJECT_VALUE
                COLUMNS lineitem    NUMBER       PATH '@ItemNumber',
                        description VARCHAR2(30) PATH 'Description',
                        partid      NUMBER       PATH 'Part/@Id',
                        unitprice   NUMBER       PATH 'Part/@UnitPrice',
                        quantity    NUMBER       PATH 'Part/@Quantity') lines;
```

```
-----------------------------------------------------------------------------------------------
| Id  | Operation                     | Name           | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |                |     4 |   384 |     7   (0)| 00:00:01 |
|   1 |  NESTED LOOPS                 |                |       |       |            |          |
|   2 |   NESTED LOOPS                |                |     4 |   384 |     7   (0)| 00:00:01 |
|*  3 |    TABLE ACCESS FULL          | PURCHASEORDER  |     1 |    37 |     5   (0)| 00:00:01 |
|*  4 |    INDEX RANGE SCAN           | SYS_C005478    |    17 |       |     1   (0)| 00:00:01 |
|*  5 |   TABLE ACCESS BY INDEX ROWID| LINEITEM_TABLE |     3 |   177 |     2   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - filter(SYS_CHECKACL("ACLOID","OWNERID",xmltype('<privilege
              xmlns="http://xmlns.oracle.com/xdb/acl.xsd"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xsi:schemaLocation="http://xmlns.oracle.com/xdb/acl.xsd
              http://xmlns.oracle.com/xdb/acl.xsd
DAV:http://xmlns.oracle.com/xdb/dav.xsd"><read-prop
              erties/><read-contents/></privilege>'))=1)
   4 - access("NESTED_TABLE_ID"="PURCHASEORDER"."SYS_NC0003400035$")
   5 - filter("SYS_NC00013$">50 AND "SYS_NC00012$">2 AND "ITEMNUMBER">=8 AND
              "SYS_NC_TYPEID$" IS NOT NULL)

25 rows selected.
```

This example traverses table `oe.purchaseorder` completely. The `XMLTable` expression is evaluated for each purchase-order document. It is more efficient to have the `XMLTable` expression, not the `purchaseorder` table, drive the SQL-query execution.

Although the XQuery expression has been rewritten to relational expressions, you can improve this optimization by creating an *index* on the underlying relational data—you can optimize this query in the same way that you would optimize a purely SQL query. That is always the case with XQuery in Oracle XML DB: the optimization techniques you use are the same as those you use in SQL.

The `UnitPrice` attribute of collection element `LineItem` is an appropriate index target. The governing XML schema specifies that an ordered collection table (OCT) is used to store the `LineItem` elements.

However, the name of this OCT was generated by Oracle XML DB when the XML purchase-order documents were decomposed as XML schema-based data. Instead of using table `purchaseorder` from sample database schema `HR`, you could manually create a new `purchaseorder` table (in a different database schema) with the same properties and same data, but having OCTs with user-friendly names. Refer to Example 3–13 on page 3-28 for how to do this.

Assuming that a `purchaseorder` table has been created as in Example 3–13, the following statement creates the appropriate index:

```
CREATE INDEX unitprice_index ON lineitem_table("PART"."UNITPRICE");
```

With this index defined, the query of Example 5–12 results in the following execution plan, which shows that the `XMLTable` expression has driven the overall evaluation.

```
PLAN_TABLE_OUTPUT
-----------------------------------------------------------------------------------------------
Plan hash value: 1578014525
```

```
-----------------------------------------------------------------------------------------
| Id  | Operation          | Name              | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |                   |     3 |   624 |     8   (0)| 00:00:01 |
|   1 |  NESTED LOOPS      |                   |     3 |   624 |     8   (0)| 00:00:01 |
|*  2 |   INDEX UNIQUE SCAN| SYS_IOT_TOP_49323 |     3 |   564 |     5   (0)| 00:00:01 |
|*  3 |    INDEX RANGE SCAN| UNITPRICE_INDEX   |    20 |       |     2   (0)| 00:00:01 |
|*  4 |   INDEX UNIQUE SCAN| SYS_C004411       |     1 |       |     0   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("SYS_NC00013$">50)
       filter("ITEMNUMBER">=8 AND "SYS_NC00012$">2)
   3 - access("SYS_NC00013$">50)
   4 - access("NESTED_TABLE_ID"="PURCHASEORDER"."SYS_NC0003400035$")

Note
-----
   - dynamic sampling used for this statement

23 rows selected.
```

Diagnosing XQuery Optimization: XMLOptimizationCheck

You can examine an execution plan for your SQL code to determine whether XQuery optimization occurs or the plan is instead suboptimal. In the latter case, a note such as the following appears immediately after the plan:

```
Unoptimized XML construct detected (enable XMLOptimizationCheck
for more information)
```

You can also compare the execution plan output with the plan output that you see after you use the optimizer hint `NO_XML_QUERY_REWRITE`, which turns off XQuery optimization.

In addition, you can use the SQL*Plus `SET` command with system variable `XMLOptimizationCheck` to turn on an **XML diagnosability mode** for SQL:

```
SET XMLOptimizationCheck ON
```

When this mode is on, the plan of execution is automatically checked for XQuery optimization, and if the plan is suboptimal then an error is raised and diagnostic information is written to the trace file indicating which operators are not rewritten.

The main advantage of `XMLOptimizationCheck` is that it brings a potential problem to your attention immediately. For this reason, you might find it preferable to leave it turned on at all times. Then, if an application change or a database change for some reason prevents a SQL operation from rewriting, execution is stopped instead of performance being negatively impacted without your being aware of the cause.

Note:

- You can use event 19027 with level 8192 (0x2000) to dump a trace that indicates why a particular expression is not rewritten.
- `XMLOptimizationCheck` was not available prior to Oracle Database 11g Release 2 (11.2.0.2). Users of older releases directly manipulated event 19201 to obtain XQuery optimization information.

See Also: "Turning Off Use of XMLIndex" on page 6-30 for information about optimizer hint `NO_XML_QUERY_REWRITE`

Improving Performance for fn:doc and fn:collection on Repository Data

In Oracle XML DB, you can use XQuery functions `fn:doc` and `fn:collection` to reference documents and collections in Oracle XML DB Repository. When repository XML data is stored object-relationally or as binary XML, queries that use `fn:doc` and `fn:collection` are evaluated functionally; that is, they are not optimized to access the underlying storage tables directly.

To improve the performance of such queries, you must link them to the actual database tables that hold the repository data being queried. You can do that in either of the following ways:

- Join view `RESOURCE_VIEW` with the `XMLType` table that holds the data, and then use the Oracle SQL functions `equals_path` and `under_path` instead of the XQuery functions `fn:doc` and `fn:collection`, respectively. These SQL functions reference repository resources in a performant way.
- Use the Oracle XQuery extension-expression pragma `ora:defaultTable`.

Both methods have the same effect. Oracle recommends that you use the `ora:defaultTable` pragma because it lets you continue to use the XQuery standard functions `fn:doc` and `fn:collection` and it simplifies your code.

These two methods are illustrated in the examples of this section.

Using equals_path and under_path Instead of fn:doc and fn:collection

SQL function `equals_path` references a resource located at a specified repository path, and SQL function `under_path` references a resource located under a specified repository path. Example 5–19 and Example 5–20 illustrate this for functions `fn:doc` and `equals_path`; functions `fn:collection` and `under_path` are treated similarly.

Example 5–19 Unoptimized Repository Query using fn:doc

```
SELECT XMLQuery('let $val :=
                     fn:doc("/home/OE/PurchaseOrders/2002/Sep/VJONES-20021009123337583PDT.xml")
                     /PurchaseOrder/LineItems/LineItem[@ItemNumber =19]
                 return $val' RETURNING CONTENT)
  FROM DUAL;
```

Example 5–20 Optimized Repository Query using EQUALS_PATH

```
SELECT XMLQuery('let $val := $DOC/PurchaseOrder/LineItems/LineItem[@ItemNumber = 19]
                 return $val' PASSING OBJECT_VALUE AS "DOC" RETURNING CONTENT)
  FROM RESOURCE_VIEW rv, purchaseorder p
  WHERE ref(p) = XMLCast(XMLQuery('declare default element namespace
                                   "http://xmlns.oracle.com/xdb/XDBResource.xsd"; (: :)
                                   fn:data[6](/Resource/XMLRef)' PASSING rv.RES RETURNING CONTENT)
                         AS REF XMLType)
    AND equals_path(rv.RES, '/home/OE/PurchaseOrders/2002/Sep/VJONES-20021009123337583PDT.xml')
        = 1;
```

[6] XQuery function `fn:data` is used here to atomize its argument, in this case returning the `XMLRef` node's typed atomic value.

Using Oracle XQuery Pragma ora:defaultTable

Oracle XQuery extension-expression pragma `ora:defaultTable` lets you specify the default table used to store repository data that you query. The query is rewritten to automatically join the default table to view `RESOURCE_VIEW` and use Oracle SQL functions `equals_path` and `under_path` instead of XQuery functions `fn:doc` and `fn:collection`, respectively. The effect is thus the same as coding the query manually to use an explicit join and `equals_path` or `under_path`. Example 5–21 illustrates this; the query is rewritten automatically to what is shown in Example 5–20.

Example 5–21 Repository Query using Oracle XQuery Pragma ora:defaultTable

```
SELECT XMLQuery('for $doc in (#ora:defaultTable PURCHASEORDER #)
                            {fn:doc("/home/OE/PurchaseOrders/2002/Sep/VJONES-20021009123337583PDT.xml")}
                  let $val := $doc/PurchaseOrder/LineItems/LineItem[@ItemNumber = 19]
                    return $val}'
                RETURNING CONTENT)
  FROM DUAL;
```

For clarity of scope Oracle recommends that you apply pragma `ora:defaultTable` directly to the relevant document or collection expression, `fn:doc` or `fn:collection`, rather than to a larger expression.

XQuery Static Type-Checking in Oracle XML DB

Oracle XML DB type-checks *all* XQuery expressions. Doing this at run time can be costly, however. As an optimization technique, whenever there is sufficient static type information available for a given query at compile time, Oracle XML DB performs *static* (compile time) type-checking of that query. Whenever sufficient static type information is not available for a given query at compile time, Oracle XML DB uses dynamic (run-time) type checking for that query.

Static type-checking can save execution time by raising errors at compile time. Static type-checking errors include both data-type errors and the use of XPath expressions that are invalid with respect to an XML schema.

Typical ways of providing sufficient static type information at query compile time include the following:

- Using XQuery with `fn:doc` or `fn:collection` over relational data.
- Using XQuery to query an `XMLType` table, column, or view whose XML Schema information is available at query compile time.

This section presents examples that demonstrate the utility of static type-checking and the use of these two means of communicating type information.

The XML data produced on the fly by `fn:collection` together with URI scheme `oradb` has `ROW` as its top-level element, but the query of Example 5–22 incorrectly lacks that `ROW` wrapper element. This omission raises a query compile-time error. Forgetting that `fn:collection` with `oradb` wraps relational data in this way is an easy mistake to make, and one that could be difficult to diagnose without static type-checking. Example 5–5 shows the correct code.

Example 5–22 Static Type-Checking of XQuery Expressions: oradb URI scheme

```
-- This produces a static-type-check error, because "ROW" is missing.
SELECT XMLQuery('for $i in fn:collection("oradb:/HR/REGIONS"),
                     $j in fn:collection("oradb:/HR/COUNTRIES")
                  where $i/REGION_ID = $j/REGION_ID and $i/REGION_NAME = "Asia"
                  return $j'
                 RETURNING CONTENT) AS asian_countries
```

```
  FROM DUAL;
SELECT XMLQuery('for $i in fn:collection("oradb:/HR/REGIONS"),
*
ERROR at line 1:
ORA-19276: XPST0005 - XPath step specifies an invalid element/attribute name:
(REGION_ID)
```

In Example 5–23, XQuery static type-checking finds a mismatch between an XPath expression and its target XML schema-based data. Element `CostCenter` is misspelled here as `costcenter` (XQuery and XPath are case-sensitive). Example 5–11 shows the correct code.

Example 5–23 Static Type-Checking of XQuery Expressions: Schema-Based XML

```
-- This results in a static-type-check error: CostCenter is not the right case.
SELECT xtab.poref, xtab.usr, xtab.requestor
  FROM purchaseorder,
       XMLTable('for $i in /PurchaseOrder where $i/costcenter eq "A10" return $i'
                PASSING OBJECT_VALUE
                COLUMNS poref     VARCHAR2(20) PATH 'Reference',
                        usr       VARCHAR2(20) PATH 'User' DEFAULT 'Unknown',
                        requestor VARCHAR2(20) PATH 'Requestor') xtab;
  FROM purchaseorder,
       *
ERROR at line 2:
ORA-19276: XPST0005 - XPath step specifies an invalid element/attribute name:
(costcenter)
```

SQL*Plus XQUERY Command

Example 5–24 shows how you can enter an XQuery expression directly at the SQL*Plus command line, by preceding the expression with the SQL*Plus command **`XQUERY`** and following it with a slash (`/`) on a line by itself. Oracle Database treats XQuery expressions submitted with this command the same way it treats XQuery expressions in SQL/XML functions `XMLQuery` and `XMLTable`. Execution is identical, with the same optimizations.

Example 5–24 Using the SQL*Plus XQUERY Command

```
SQL> XQUERY for $i in fn:collection("oradb:/HR/DEPARTMENTS")
  2  where $i/ROW/DEPARTMENT_ID < 50
  3  return $i
  4  /

Result Sequence
-------------------------------------------------------------------------------
<ROW><DEPARTMENT_ID>10</DEPARTMENT_ID><DEPARTMENT_NAME>Administration</DEPARTMEN
T_NAME><MANAGER_ID>200</MANAGER_ID><LOCATION_ID>1700</LOCATION_ID></ROW>

<ROW><DEPARTMENT_ID>20</DEPARTMENT_ID><DEPARTMENT_NAME>Marketing</DEPARTMENT_NAM
E><MANAGER_ID>201</MANAGER_ID><LOCATION_ID>1800</LOCATION_ID></ROW>

<ROW><DEPARTMENT_ID>30</DEPARTMENT_ID><DEPARTMENT_NAME>Purchasing</DEPARTMENT_NA
ME><MANAGER_ID>114</MANAGER_ID><LOCATION_ID>1700</LOCATION_ID></ROW>

<ROW><DEPARTMENT_ID>40</DEPARTMENT_ID><DEPARTMENT_NAME>Human Resources</DEPARTME
NT_NAME><MANAGER_ID>203</MANAGER_ID><LOCATION_ID>2400</LOCATION_ID></ROW>
```

There are also a few SQL*Plus `SET` commands that you can use for settings that are specific to XQuery. Use `SHOW XQUERY` to see the current settings.

- **`SET XQUERY BASEURI`** – Set the base URI for `XQUERY`. URIs in XQuery expressions are relative to this URI.
- **`SET XQUERY CONTEXT`** – Specify a context item for subsequent `XQUERY` evaluations.

See Also: *SQL*Plus User's Guide and Reference*

Using XQuery with PL/SQL, JDBC, and ODP.NET

Previous sections in this chapter have shown how to invoke XQuery from SQL. This section provides examples of using XQuery with the Oracle APIs for PL/SQL, JDBC, and Oracle Data Provider for .NET (ODP.NET).

Example 5–25 shows how to use XQuery with PL/SQL, in particular, how to bind *dynamic variables* to an XQuery expression using the `XMLQuery PASSING` clause. The bind variables `:1` and `:2` are bound to the PL/SQL bind arguments `nbitems` and `partid`, respectively. These are then passed to XQuery as XQuery variables `itemno` and `id`, respectively.

Example 5–25 Using XQuery with PL/SQL

```
DECLARE
  sql_stmt VARCHAR2(2000); -- Dynamic SQL statement to execute
  nbitems  NUMBER := 3; -- Number of items
  partid   VARCHAR2(20):= '715515009058'; -- Part ID
  result   XMLType;
  doc      DBMS_XMLDOM.DOMDocument;
  ndoc     DBMS_XMLDOM.DOMNode;
  buf      VARCHAR2(20000);
BEGIN
  sql_stmt :=
    'SELECT XMLQuery(
              ''for $i in fn:collection("oradb:/OE/PURCHASEORDER") ' ||
               'where count($i/PurchaseOrder/LineItems/LineItem) = $itemno ' ||
                 'and $i/PurchaseOrder/LineItems/LineItem/Part/@Id = $id ' ||
               'return $i/PurchaseOrder/LineItems'' ' ||
              'PASSING :1 AS "itemno", :2 AS "id" ' ||
              'RETURNING CONTENT) FROM DUAL';

  EXECUTE IMMEDIATE sql_stmt INTO result USING nbitems, partid;
  doc  := DBMS_XMLDOM.newDOMDocument(result);
  ndoc := DBMS_XMLDOM.makeNode(doc);
  DBMS_XMLDOM.writeToBuffer(ndoc, buf);
  DBMS_OUTPUT.put_line(buf);
END;
/
```

This produces the following output:

```
<LineItems>
  <LineItem ItemNumber="1">
    <Description>Samurai 2: Duel at Ichijoji Temple</Description>
    <Part Id="37429125526" UnitPrice="29.95" Quantity="3"/>
  </LineItem>
  <LineItem ItemNumber="2">
    <Description>The Red Shoes</Description>
```

```
      <Part Id="37429128220" UnitPrice="39.95" Quantity="4"/>
    </LineItem>
    <LineItem ItemNumber="3">
      <Description>A Night to Remember</Description>
      <Part Id="715515009058" UnitPrice="39.95" Quantity="1"/>
    </LineItem>
  </LineItems>
  <LineItems>
    <LineItem ItemNumber="1">
      <Description>A Night to Remember</Description>
      <Part Id="715515009058" UnitPrice="39.95" Quantity="2"/>
    </LineItem>
    <LineItem ItemNumber="2">
      <Description>The Unbearable Lightness Of Being</Description>
      <Part Id="37429140222" UnitPrice="29.95" Quantity="2"/>
    </LineItem>
    <LineItem ItemNumber="3">
      <Description>Sisters</Description>
      <Part Id="715515011020" UnitPrice="29.95" Quantity="4"/>
    </LineItem>
  </LineItems>

PL/SQL procedure successfully completed.
```

Example 5–26 shows how to use XQuery with JDBC, binding variables by position with the `PASSING` clause of SQL/XML function `XMLTable`.

Example 5–26 Using XQuery with JDBC

```
import java.sql.*;
import oracle.sql.*;
import oracle.jdbc.*;
import oracle.xdb.XMLType;
import java.util.*;

public class QueryBindByPos
{
  public static void main(String[] args) throws Exception, SQLException
  {
    System.out.println("*** JDBC Access of XQuery using Bind Variables ***");
    DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
    OracleConnection conn
      = (OracleConnection)
        DriverManager.getConnection("jdbc:oracle:oci8:@localhost:1521:ora11gR1", "oe", "oe");
    String xqString
      = "SELECT COLUMN_VALUE" +
          "FROM XMLTable('for $i in fn:collection(\"oradb:/OE/PURCHASEORDER\") " +
                         "where $i/PurchaseOrder/Reference= $ref " +
                         "return $i/PurchaseOrder/LineItems' " +
                        "PASSING ? AS \"ref\")";
    OraclePreparedStatement stmt = (OraclePreparedStatement)conn.prepareStatement(xqString);
    String refString = "EABEL-20021009123336251PDT"; // Set the filter value
    stmt.setString(1, refString); // Bind the string
    ResultSet rs = stmt.executeQuery();
    while (rs.next())
    {
       XMLType desc = (XMLType) rs.getObject(1);
       System.out.println("LineItem Description: " + desc.getStringVal());
       desc.close();
    }
```

```
    rs.close();
    stmt.close();
  }
}
```

This produces the following output:

```
*** JDBC Access of Database XQuery with Bind Variables ***
LineItem Description: Samurai 2: Duel at Ichijoji Temple
LineItem Description: The Red Shoes
LineItem Description: A Night to Remember
```

Example 5–27 shows how to use XQuery with ODP.NET and the C# language. The C# input parameters `:nbitems` and `:partid` are passed to XQuery as XQuery variables `itemno` and `id`, respectively.

Example 5–27 Using XQuery with ODP.NET and C#

```
using System;
using System.Data;
using System.Text;
using System.IO;
using System.Xml;
using Oracle.DataAccess.Client;
using Oracle.DataAccess.Types;

namespace XQuery
{
  /// <summary>
  /// Demonstrates how to bind variables for XQuery calls
  /// </summary>
  class XQuery
  {
    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    static void Main(string[] args)
    {
      int rows = 0;
      StreamReader sr = null;

      // Create the connection.
      string constr = "User Id=oe;Password=***********;Data Source=ora11gr2"; // Replace with real password.
      OracleConnection con = new OracleConnection(constr);
      con.Open();

      // Create the command.
      OracleCommand cmd = new OracleCommand("", con);

      // Set the XML command type to query.
      cmd.CommandType   = CommandType.Text;

      // Create the SQL query with the XQuery expression.
      StringBuilder blr = new StringBuilder();
      blr.Append("SELECT COLUMN_VALUE FROM XMLTable");
      blr.Append("(\'for $i in fn:collection(\"oradb:/OE/PURCHASEORDER\") ");
      blr.Append("   where count($i/PurchaseOrder/LineItems/LineItem) = $itemno ");
      blr.Append("      and $i/PurchaseOrder/LineItems/LineItem/Part/@Id = $id ");
      blr.Append("   return $i/PurchaseOrder/LineItems\' ");
      blr.Append("  PASSING :nbitems AS \"itemno\", :partid AS \"id\")");

      cmd.CommandText = blr.ToString();
      cmd.Parameters.Add(":nbitems", OracleDbType.Int16, 3, ParameterDirection.Input);
      cmd.Parameters.Add(":partid", OracleDbType.Varchar2, "715515009058", ParameterDirection.Input);

      // Get the XML document as an XmlReader.
```

```
      OracleDataReader dr = cmd.ExecuteReader();
      dr.Read();

      // Get the XMLType column as an OracleXmlType
      OracleXmlType xml = dr.GetOracleXmlType(0);

      // Print the XML data in the OracleXmlType object
      Console.WriteLine(xml.Value);
      xml.Dispose();

      // Clean up.
      cmd.Dispose();
      con.Close();
      con.Dispose();
    }
  }
}
```

This produces the following output:

```
<LineItems>
  <LineItem ItemNumber="1">
    <Description>Samurai 2: Duel at Ichijoji Temple</Description>
    <Part Id="37429125526" UnitPrice="29.95" Quantity="3"/>
  </LineItem>
  <LineItem ItemNumber="2">
    <Description>The Red Shoes</Description>
    <Part Id="37429128220" UnitPrice="39.95" Quantity="4"/>
  </LineItem>
  <LineItem ItemNumber="3">
    <Description>A Night to Remember</Description>
    <Part Id="715515009058" UnitPrice="39.95" Quantity="1"/>
  </LineItem>
</LineItems>
```

See Also:

- Chapter 13, "PL/SQL APIs for XMLType"
- Chapter 15, "Java DOM API for XMLType"
- Chapter 17, "Using Oracle Data Provider for .NET with Oracle XML DB"

Oracle XML DB Support for XQuery

This section describes Oracle XML DB for the XQuery language.

Support for XQuery and SQL

Support for the XQuery language in Oracle XML DB is designed to provide the best fit between the worlds of relational storage and querying XML data. That is, Oracle XML DB is a general XQuery implementation, but it is in addition specifically designed to make relational and XQuery queries work well together.

The specific properties of the Oracle XML DB XQuery implementation are described in this section. The XQuery standard explicitly calls out certain aspects of the language processing as implementation-defined or implementation-dependent. There are also some features that are specified by the XQuery standard but are not supported by Oracle XML DB.

Implementation Choices Specified in the XQuery Standard

The XQuery specification specifies that each of the following aspects of language processing is to be defined by the implementation.

- *Implicit time zone support* – In Oracle XML DB, the implicit time zone is always assumed to be `Z`, and instances of `xs:date`, `xs:time`, and `xs:datetime` that are missing time zones are automatically converted to UTC.

XQuery Features Not Supported by Oracle XML DB

The following features specified by the XQuery standard are not supported by Oracle XML DB:

- *Copy-namespace mode* – Oracle XML DB supports only `preserve` and `inherit` for a `copy-namespaces` declaration. If an existing element node is copied by an element constructor or a document constructor, all in-scope namespaces of the original element are retained in the copy. Otherwise, the copied node inherits all in-scope namespaces of the constructed node. An error is raised if you specify `no-preserve` or `no-inherit`.
- *Version encoding* – Oracle XML DB does not support an optional encoding declaration in a version declaration. That is, you cannot include `(encoding` *`an-encoding`*`)` in a declaration `xquery version` *`a-version`*`;`. In particular, you cannot specify an encoding used in the query. An error is raised if you include an encoding declaration.
- *xml:id* – Oracle XML DB does not support use of `xml:id`. If you use `xml:id`, then an error is raised.
- XQuery prolog default-collation declaration.
- XQuery prolog boundary-space declaration.
- XQuery data type `xs:duration`. Use either `xs:yearMonthDuration` or `xs:DayTimeDuration` instead.

XQuery Optional Features

The following optional features specified by the XQuery standard are not supported by Oracle XML DB:

- Schema Validation Feature
- Module Feature

Support for XQuery Functions and Operators

Oracle XML DB supports all of the XQuery functions and operators included in the latest *XQuery 1.0 and XPath 2.0 Functions and Operators* specification, with the following exceptions. There is *no* support for the following:

- The XQuery regular-expression functions: `fn:matches`, `fn:replace`, and `fn:tokenize`. Use Oracle XQuery functions `ora:matches`, `ora:replace`, and `ora:tokenize` instead, respectively.
- Functions `fn:id` and `fn:idref`.
- Function `fn:collection` without arguments.
- Optional collation parameters for XQuery functions.

XQuery Functions fn:doc, fn:collection, and fn:doc-available

Oracle XML DB supports XQuery functions `fn:doc`, `fn:collection`, and `fn:doc-available` for all resources in Oracle XML DB Repository.

Function `fn:doc` returns the repository *file* resource that is targeted by its URI argument; it must be a file of well-formed XML data. Function `fn:collection` is similar, but works on repository *folder* resources (each file in the folder must contain well-formed XML data).

When used with Oracle URI scheme `oradb`, `fn:collection` can return XML data derived on the fly from existing relational data that is not in the repository.

XQuery function `fn:collection` raises an error when used with URI scheme `oradb`, if its targeted table or view, or a targeted column, does not exist. Functions `fn:doc` and `fn:collection` do *not* raise an error if the repository resource passed as argument is not found. Instead, they return an empty sequence.

You can determine whether a given document exists using XQuery function `fn:doc-available`. It returns `true` if its document argument exists, `false` if not.

> **See Also:** `http://www.w3.org` for the definitions of XQuery functions and operators

6

Indexing XMLType Data

You can create indexes on your XML data, to focus on particular parts of it that you query often and thus improve performance. This chapter includes guidelines for doing this. It describes various ways that you can index `XMLType` data, whether schema-based or non-schema-based, and regardless of the `XMLType` storage model you use (binary XML, unstructured, hybrid, or structured).

This chapter contains these topics:

- Oracle XML DB Tasks Involving Indexes
- Overview of Indexing XMLType Data
- Indexing XMLType Data Stored Object-Relationally
- XMLIndex
- Oracle Text Indexes on XML Data

Note: The execution plans shown here are for illustration only. If you run the examples presented here in your environment then your execution plans might not be identical.

See Also:

- *Oracle Database Concepts* for an overview of indexing
- *Oracle Database Advanced Application Developer's Guide* for information about using indexes in application development

Oracle XML DB Tasks Involving Indexes

Table 6–1 identifies the documentation for some basic user tasks involving indexes for XML data.

Table 6–1 Basic XML Indexing Tasks

For information about how to...	See...
Index `XMLType` data stored object-relationally	"Indexing XMLType Data Stored Object-Relationally" on page 6-6, "Guideline: Create indexes on ordered collection tables" on page 8-6
Create, drop, or rename an `XMLIndex` index	Example 6–6 on page 6-18, Example 6–8 on page 6-18
Create an `XMLIndex` index on hybrid storage (`CLOB` embedded in object-relational storage)	Example 6–9 on page 6-19
Obtain the name of an `XMLIndex` index for a given table or column	Example 6–7 on page 6-18
Determine whether a given `XMLIndex` index is used in evaluating a query	"How to Tell Whether XMLIndex is Used" on page 6-25
Turn off use of an `XMLIndex` index	"Turning Off Use of XMLIndex" on page 6-30

Table 6–2 identifies the documentation for some user tasks involving `XMLIndex` indexes that have a *structured* component.

Table 6–2 Tasks Involving XMLIndex Indexes with a Structured Component

For information about how to...	See...
Create an `XMLIndex` index with a structured component	Example 6–22 on page 6-23, Example 6–26 on page 6-25
Drop the structured component of an `XMLIndex` index (drop all structure groups)	Example 6–23 on page 6-24
Ensure data type correspondence between a query and an `XMLIndex` index with a structured component	"Data Type Considerations for XMLIndex Structured Component" on page 6-11
Create a B-tree index on a content table of an `XMLIndex` structured component	Example 6–24 on page 6-24
Create an Oracle Text `CONTEXT` index on a content table of an `XMLIndex` structured component	Example 6–25 on page 6-24

Table 6–3 identifies the documentation for some user tasks involving `XMLIndex` indexes that have an *unstructured* component.

Table 6–3 Tasks Involving XMLIndex Indexes with an Unstructured Component

For information about how to...	See...
Create an `XMLIndex` index with an unstructured component	Example 6–9 on page 6-19, Example 6–11 on page 6-19, Example 6–13 on page 6-20, Example 6–33 on page 6-32, Example 6–35 on page 6-32, Example 6–38 on page 6-37, Example 6–39 on page 6-37, Example 6–40 on page 6-39
Drop the unstructured component of an `XMLIndex` index (drop the path table)	Example 6–14 on page 6-20
Name the path table when creating an `XMLIndex` index	Example 6–11 on page 6-19
Specify storage options when creating an `XMLIndex` index	Example 6–13 on page 6-20
Show all existing secondary indexes on an `XMLIndex` path table	Example 6–15 on page 6-20, Example 6–21 on page 6-22

Table 6–3 (Cont.) Tasks Involving XMLIndex Indexes with an Unstructured Component

For information about how to...	See...
Obtain the name of a path table for an `XMLIndex` index	Example 6–12 on page 6-19
Obtain the name of an `XMLIndex` index with an unstructured component, given its path table	Example 6–28 on page 6-27
Create a secondary index on an `XMLIndex` path table	"Using XMLIndex with an Unstructured Component" on page 6-19
Obtain information about all of the secondary indexes on an `XMLIndex` path table	Example 6–21 on page 6-22
Create a function-based index on a path-table `VALUE` column	Example 6–16 on page 6-21
Create a numeric index on a path-table `VALUE` column	Example 6–18 on page 6-21
Create a date index on a path-table `VALUE` column	Example 6–19 on page 6-22
Create an Oracle Text `CONTEXT` index on a path-table `VALUE` column	Example 6–20 on page 6-22
Exclude or include particular XPath expressions from use by an `XMLIndex` index	"XMLIndex Path Subsetting: Specifying the Paths You Want to Index" on page 6-31
Specify namespace prefixes for XPath expressions used for `XMLIndex`	"XMLIndex Path Subsetting: Specifying the Paths You Want to Index" on page 6-31
Exclude or include particular XPath expressions from use by an `XMLIndex` index	"XMLIndex Path Subsetting: Specifying the Paths You Want to Index" on page 6-31
Specify namespace prefixes for XPath expressions used for `XMLIndex`	"XMLIndex Path Subsetting: Specifying the Paths You Want to Index" on page 6-31

Table 6–4 identifies the documentation for some other user tasks involving `XMLIndex` indexes.

Table 6–4 Miscellaneous Tasks Involving XMLIndex Indexes

For information about how to...	See...
Create an `XMLIndex` on Oracle XML DB Repository	"Using XMLIndex on Oracle XML DB Repository" on page 6-34
Query Oracle XML DB Repository using `XMLIndex`	"Using XMLIndex on Oracle XML DB Repository" on page 6-34
Specify that an `XMLIndex` index should be created and maintained using parallel processes	"XMLIndex Partitioning and Parallelism" on page 6-36
Change the parallelism of an `XMLIndex` path table to tune index performance	"XMLIndex Partitioning and Parallelism" on page 6-36
Schedule maintenance for an `XMLIndex` index	"Asynchronous (Deferred) Maintenance of XMLIndex Indexes" on page 6-38
Manually synchronize an `XMLIndex` index and its base table	"Asynchronous (Deferred) Maintenance of XMLIndex Indexes" on page 6-38
Collect statistics on a table or index for the cost-based optimizer	Example 6–42 on page 6-39
Create an Oracle Text `CONTEXT` index	Example 6–43 on page 6-49
Create an Oracle Text `CONTEXT` index on a content table of an `XMLIndex` structured component	Example 6–25 on page 6-24
Use an Oracle Text `CONTEXT` index for full-text search of XML data	Example 6–44 on page 6-49
Show whether an Oracle Text `CONTEXT` index is used in a query	Example 6–44 on page 6-49

Overview of Indexing XMLType Data

Database indexes improve performance by providing faster access to table data. The use of indexes is particularly recommended for online transaction processing (OLTP) environments involving few updates.

The principle way you index XML data is using `XMLIndex`. You can also use Oracle Text `CONTEXT` indexes to supplement the use of `XMLIndex`.

XMLIndex Addresses the Fine-Grained Structure of XML Data

You can create indexes on one or more table columns, or on a functional expression. XML data, however, has its own, fine-grained structure, which is not necessarily reflected in the structure of the database tables used to store it. For this reason, effectively indexing XML data can be a bit different from indexing most database data.

For structured XML storage, XML objects such as elements and attributes correspond to object-relational columns and tables, so creating *B-tree indexes* on those columns and tables provides an excellent way to effectively index the corresponding XML objects. Here, the storage model directly reflects the fine-grained structure of the XML data, so there is no special problem for indexing structured XML data. See "Indexing XMLType Data Stored Object-Relationally" on page 6-6.

For unstructured, hybrid, and binary XML storage models, indexing a database column using the standard sorts of index (B-tree, bitmap) is generally not helpful for accessing particular parts of an XML document. If an `XMLType` column that contains an XML document is stored as a `CLOB` instance, then the details within that document are inaccessible to the column index—the entire document acts as a single unit as far as the column index is concerned. In **hybrid storage**, part of an XML document is broken up and stored object-relationally (structured storage), but one or more XML fragments are stored as `CLOB` instances (unstructured storage). A typical use case here is mapping an XML-schema `complexType` or a complex element to `CLOB` storage, because the entire fragment is generally accessed as a unit. For standard indexes, it acts as a unit for indexing as well.

`XMLIndex` provides a general, XML-specific index that indexes the internal structure of XML data. One of its main purposes is to overcome the indexing limitation presented by unstructured, hybrid, and binary XML storage.

- An `XMLIndex` index with an *unstructured component* indexes the XML *tags* of your document and identifies document fragments based on XPath expressions that target them. It can also index scalar node *values*, to provide quick lookup based on individual values or ranges of values. It also records document *hierarchy* information for each node it indexes: relations parent–child, ancestor–descendant, and sibling. This index component is particularly useful for queries that extract XML fragments from documents that have little or variable structure.
- An `XMLIndex` index with a *structured component* indexes highly structured and predictable parts of XML data that is nevertheless for the most part unstructured. This index component is particularly useful for queries that project and use such islands of structured content.

See Also: "XMLIndex" on page 6-7

Oracle Text Indexes

Besides accessing XML nodes such as elements and attributes, it is sometimes important to provide fast access to particular passages of text within XML text nodes. This is the purpose of *Oracle Text indexes*: they index full-text strings. An Oracle Text

`CONTEXT` index enables Oracle SQL function `contains` for full-text search over XML. With structured storage, XPath rewrite can often rewrite queries that use XPath function `ora:contains` to queries that use SQL function `contains`, so in those cases too an Oracle Text index can be employed.

Full-text indexing is particularly useful for *document-centric* applications, which often contain a mix of XML elements and text-node content. Full-text searching can often be made more powerful, more focused, by combining it with structural XML searching, that is, by restricting it to certain parts of an XML document, which are identified by using XPath expressions.

See Also: "Oracle Text Indexes on XML Data" on page 6-48

Optimization Chooses the Right Indexes to Use

Which indexes are used when more than one might apply in a given case? Cost-based optimization determines the index or indexes to use, so that performance is maximized. Oracle Text indexes apply only to text, which, for XML data, means text nodes. Whenever text nodes are targeted and a corresponding Oracle Text index is defined, it is used. If other indexes are also appropriate in a particular context, then they can be used as well. However, just because an index is defined and it might appear applicable in a given situation does not mean that it will be used—it will not be used if the cost-based optimizer deems that its use is not cost-effective.

Deprecated Indexes for XML Data

In releases prior to Oracle Database 11*g* Release 1 (11.1), `CTXXPath` indexes were sometimes appropriate for use with `XMLType` data. In releases prior to Oracle Database 11*g* Release 2 (11.2), function-based indexes were sometimes appropriate for use with `XMLType` data. These indexing methods are no longer recommended for use with `XMLType` data.

Function-Based Indexes

In releases prior to Oracle Database 11*g* Release 2 (11.2), function-based indexes were sometimes appropriate for use with `XMLType` data when an XPath expression targeted a singleton node. Oracle recommends that you use the structured component of `XMLIndex` instead. Doing so obviates the overhead associated with maintenance operations on function-based indexes, and it increases the number of situations in which the optimizer can correctly select the index. No changes to existing DML statements are required as a result of this.

It continues to be the case that, for structured storage, defining an index for (deprecated) Oracle SQL function `extractValue` often leads, by XPath rewrite, to automatic creation of B-tree indexes on the underlying objects (instead of a function-based index on `extractValue`). The XPath target here must be a *singleton* element or attribute. A similar shortcut exists for `XMLCast` applied to `XMLQuery`.

See Also:

- "Indexing XMLType Data Stored Object-Relationally" on page 6-6
- "XMLIndex Structured Component" on page 6-10

CTXXPath Indexes

Another type of index that is available for indexing XML data, `CTXXPath`, is *deprecated*, starting with Oracle Database 11*g* Release 1 (11.1). It has been superseded by `XMLIndex`, and it is made available *only for use with older database releases*. It cannot

help in extracting an XML fragment, and it acts only as a preliminary filter for equality predicates; after such filtering, XPath expressions are evaluated functionally (that is, without the benefit of XPath rewrite).

Note: `CTXSYS.CTXXPath` indexing was deprecated in Oracle Database 11g Release 1 (11.1). The functionality that was provided by `CTXXPath` is now provided by `XMLIndex`.

Oracle recommends that you replace `CTXXPath` indexes with `XMLIndex` indexes. The intention is that `CTXXPath` will no longer be supported in a future release of the database.

Indexing XMLType Data Stored Object-Relationally

You can effectively index XML data that is stored object-relationally (structured storage) by creating B-tree indexes on the underlying database columns that correspond to XML nodes.

If the data to be indexed is a *singleton*, that is, if it can occur only once in any XML instance document, then you can use a *shortcut* of ostensibly creating a function-based index, where the expression defining the index is a functional application, with an XPath-expression argument that targets the singleton data. A shortcut is defined for `XMLCast` applied to `XMLQuery`, and another shortcut is defined for (deprecated) Oracle SQL function `extractValue`.

In many cases, Oracle XML DB then automatically creates appropriate indexes on the underlying object-relational tables or columns; it does *not* create a function-based index on the targeted `XMLType` data as the `CREATE INDEX` statement would suggest.

In the case of the `extractValue` shortcut, the index created is a B-tree index. In the case of `XMLCast` applied to `XMLQuery`, the index created is a function-based index on the scalar value resulting from the functional expression. "Indexing Non-Repeating text() Nodes or Attribute Values" describes this.

If the data to be indexed is a *collection*, then you cannot use such a shortcut; you must create the B-tree indexes manually. "Indexing Repeating (Collection) Elements" on page 6-7 describes this.

Indexing Non-Repeating text() Nodes or Attribute Values

Table `purchaseorder` in sample database schema `OE` is stored object-relationally. Each purchase-order document has a single `Reference` element; this element is a singleton. You can thus use a shortcut to create an index on the underlying object-relational data.

Example 6–1 shows a `CREATE INDEX` statement that ostensibly tries to create a function-based index using `XMLCast` applied to `XMLQuery`, targeting the text content of element `Reference`. (The content of this element is only text, so targeting the element is the same as targeting its text node using node test `text()`.)

Example 6–2 ostensibly tries to create a function-based index using (deprecated) Oracle SQL function `extractValue`, targeting the same data.

Example 6–1 CREATE INDEX using XMLCAST and XMLQUERY on a Singleton Element

```
CREATE INDEX po_reference_ix ON purchaseorder
  (XMLCast(XMLQuery ('$p/PurchaseOrder/Reference' PASSING po.OBJECT_VALUE AS "p"
                                                  RETURNING CONTENT)
```

```
                AS VARCHAR2(128)));
```

Example 6–2 CREATE INDEX using EXTRACTVALUE on a Singleton Element

```
CREATE INDEX po_reference_ix ON purchaseorder
  (extractValue(OBJECT_VALUE, '/PurchaseOrder/Reference'));
```

In reality, in both Example 6–1 and Example 6–2 no function-based index is created on the targeted `XMLType` data. Instead, Oracle XML DB rewrites the `CREATE INDEX` statements to create indexes on the underlying scalar data.

> **See Also:** Example 8–4 and Example 8–5 on page 8-5 for information about XPath rewrite as it applies to such `CREATE INDEX` statements

In some cases when you use either of these shortcuts, the `CREATE INDEX` statement is not able to create an index on the underlying scalar data as described, and it instead actually does create a function-based index on the referenced `XMLType` data. (This is so, even if the *value* of the index might be a scalar.)

If this happens, drop the index, and create instead an `XMLIndex` index with a structured component that targets the same XPath. As a general rule, Oracle recommends against using a function-based index on `XMLType` data.

This is an instance of a general rule for `XMLType` data, regardless of the storage method used: Use an `XMLIndex` with a structured component instead of a function-based index. This rule applies starting with Oracle Database 11*g* Release 2 (11.2). Respecting this rule obviates the overhead associated with maintenance operations on function-based indexes, and it can increase the number of situations in which the optimizer can correctly select the index.

> **See Also:** "Function-Based Indexes" on page 6-5

Indexing Repeating (Collection) Elements

In structured storage, a collection is stored as an ordered collection table (OCT) of an `XMLType` instance, which means that you can directly access its members. Because the structured storage model directly reflects the fine-grained structure of the XML data, you can create indexes that target individual collection members.

You must create such indexes manually. The special feature of automatically creating B-tree indexes when you ostensibly create a function-based index for (deprecated) Oracle SQL function `extractValue` does *not* apply to collections (the XPath expression passed to `extractValue` must target a singleton).

To create B-tree indexes for a collection, you must understand the structure of the SQL object that is used to manage the collection. Given this information, you can use conventional object-relational SQL code to created the indexes directly on the appropriate SQL-object attributes. Refer to "Guideline: Create indexes on ordered collection tables" on page 8-6 for an example of how to do this.

XMLIndex

This section contains these topics:

- Advantages of XMLIndex
- Structured and Unstructured XMLIndex Components
- XMLIndex Structured Component

- XMLIndex Unstructured Component
- Creating, Dropping, Altering, and Examining an XMLIndex Index
- Using XMLIndex with an Unstructured Component
- Using XMLIndex with a Structured Component
- How to Tell Whether XMLIndex is Used
- Turning Off Use of XMLIndex
- XMLIndex Path Subsetting: Specifying the Paths You Want to Index
- Guidelines for Using XMLIndex with an Unstructured Component
- Guidelines for Using XMLIndex with a Structured Component
- Using XMLIndex on Oracle XML DB Repository
- XMLIndex Partitioning and Parallelism
- Asynchronous (Deferred) Maintenance of XMLIndex Indexes
- Collecting Statistics on XMLIndex Objects for the Cost-Based Optimizer
- Data Dictionary Static Public Views Related to XMLIndex
- PARAMETERS Clause for CREATE INDEX and ALTER INDEX

B-tree indexes can be used advantageously with structured (object-relational) storage—they provide sharp focus by targeting the underlying objects directly. They are generally ineffective, however, in addressing the detailed structure (elements and attributes) of an XML document stored using binary XML or `CLOB` storage, or of an XML fragment stored in a `CLOB` instance embedded in object-relational storage. That is the special domain of `XMLIndex`: unstructured and hybrid storage.

One typical use case for `XMLIndex` is where you generally expect to access certain portions of a document in their entirety, so you pack those portions into one or more `CLOB` instances. You might nevertheless sometimes need to query within these document portions. `XMLIndex` can help here.

Another use case is where an XML schema contains `xsd:any` elements, for lack of any specific knowledge of the document structure and data types involved. The data corresponding to these elements is stored in `CLOB` instances, and `XMLIndex` can be used to speed access to it.

Advantages of XMLIndex

`XMLIndex` is a *domain* index; it is designed specifically for the domain of XML data. It is a *logical* index. An `XMLIndex` index can be used for SQL/XML functions `XMLQuery`, `XMLTable`, `XMLExists`, and `XMLCast`.

XMLIndex presents the following advantages over other indexing methods:

- An `XMLIndex` index is effective in any part of a query; it is not limited to use in a `WHERE` clause. This is not the case for any of the other kinds of indexes you might use with XML data.
- An `XMLIndex` index with an unstructured component can speed access to both `SELECT` list data and `FROM` list data, making it useful for XML *fragment* extraction, in particular. Function-based indexes and `CTXXPath` indexes, both of which are deprecated, cannot be used to extract document fragments.
- You can use an `XMLIndex` index with either XML schema-based or non-schema-based data. You can use it with unstructured storage, hybrid storage,

and binary XML storage. B-tree indexing is appropriate only for XML schema-based data that is stored object-relationally (structured storage); it is ineffective for XML schema-based data stored in a CLOB instance.

- You can use an XMLIndex index for searches with XPath expressions that target *collections*, that is, nodes that occur multiple times within a document. This is not the case for function-based indexes.
- You need no prior knowledge of the XPath expressions that might be used in queries. The unstructured component of an XMLIndex index can be completely general. This is not the case for function-based indexes.
- If you have prior knowledge of the XPath expressions to be used in queries, then you can improve performance either by using a *structured* XMLIndex component that targets fixed, structured islands of data that are queried often.
- XMLIndex indexing—both index creation and index maintenance—can be carried out in parallel, using multiple database processes. This is not the case for function-based and CTXXPATH indexes, which are deprecated.

Structured and Unstructured XMLIndex Components

XMLIndex is used to index XML data that is semi-structured[1], that is, data that generally has little or no fixed structure. It applies to data that is stored using binary XML or CLOB-based storage. This includes XML data stored in CLOB instances that are embedded in object-relational storage (hybrid storage).

Semi-structured XML data can sometimes nevertheless contain islands of predictable, structured data. An XMLIndex index can therefore have two components: a **structured component**, used to index such islands, and an **unstructured component**, used to index data that has little or variable structure.

A structured component can help with queries that project and use islands of structured content. An unstructured component can help with queries that extract XML fragments. Either component can be omitted from a given XMLIndex index.

Unlike a structured component, an unstructured component is general and relatively untargeted. Though you can restrict an unstructured component to apply only to certain XPath subsets, its path table indexes node content that can be of different scalar types, which can require you to create multiple secondary indexes on the VALUE column to deal with the different data types—see "Secondary Indexes on Column VALUE" on page 6-17. Using an unstructured component alone can also lead to inefficiencies involving multiple probes and self-joins of its path table, for queries that project structured islands.

On the other hand, a structured component is not suited for queries that involve little structure or queries that extract XML fragments. Use a structured component to index structured islands of data; use an unstructured component to index data that has little structure.

Figure 6–1 is the same as Figure 1–5 in Chapter 1. The last row indicates the applicability of XMLIndex for different XML data use cases. It shows that XMLIndex is appropriate for semi-structured XML data, however it is stored (last three columns). And an XMLIndex index with a structured component is useful for document-centric data that contains structured islands (fourth column).

[1] In this book, "structured" and "unstructured" generally refer to XMLType storage options; they refer less often to the nature of your data. "Hybrid" refers to object-relational storage with some embedded CLOB storage. "Semi-structured" refers to XML content, regardless of storage. Unstructured storage is CLOB-based storage, and structured storage is object-relational storage.

Figure 6–1 XML Use Cases and XML Indexing

	Data-Centric		Document-Centric	
Use Case	XML schema-based data, with little variation and little structural change over time	XML schema-based data, with some embedded variable data	Variable, free-form data, with some fixed embedded structures	Variable, free-form data
Typical Data	Employee record	Employee record that includes a free-form resume	Technical article, with author, date, and title fields	Web document or book chapter
Storage Model	Object-Relational (Structured)	Hybrid	`CLOB` (Unstructured) or Binary XML	
Indexing	B-tree index	• B-tree index • `XMLIndex` index with unstructured component	`XMLIndex` index with structured and unstructured components	`XMLIndex` index with unstructured component

****

See Also:

- "XMLIndex Structured Component" on page 6-10
- "XMLIndex Unstructured Component" on page 6-13
- "Advantages of XMLIndex" on page 6-8 for a summary of the advantages provided by each `XMLIndex` component type

XMLIndex Structured Component

You create and use the structured component of an `XMLIndex` index for queries that project fixed, structured islands of XML content, even if the surrounding data is relatively unstructured.

A structured `XMLIndex` component organizes such islands in a relational format. In this it is similar to SQL/XML *function* `XMLTable`, and the syntax you use to define the structured component reflects this similarity. The relational tables used to store the indexing data are data-type aware, and each column can be of a different scalar data type.

You can thus think of the act of creating the structured component of an `XMLIndex` index as *decomposing* a structured portion of your XML data into relational format. This differs from the object-relational storage model of `XMLType` in these ways:

- A structured index component *explicitly* decomposes particular *portions* of your data, which you specify—portions that you commonly query. Object-relational `XMLType` storage involves automatic decomposition of an entire `XMLType` table or column.
- The structured component of an `XMLIndex` index applies to both XML schema-based and non-schema-based data. Object-relational `XMLType` storage applies only to data that is based on an XML schema.
- The decomposed data for a structured `XMLIndex` component is stored in addition to the `XMLType` data, as an index, rather than being the storage model for the `XMLType` data itself.

- For a structured `XMLIndex` component, the same data can be projected multiple times, as columns of different data type.

The index content tables used for the structured component of an `XMLIndex` index are part of the index, but because they are normal relational tables you can, in turn, *index* them using any standard relational indexes, including indexes that satisfy primary-key and foreign-key constraints. You can also index them using domain indexes, such as an Oracle Text `CONTEXT` index.

Another way to look at the structured component of an `XMLIndex` index sees that it acts as a *generalized function-based index*. A function-based index is similar to a structured `XMLIndex` component that has only one relational column.

If you find that for a particular application you are creating multiple function-based indexes, then consider using a structured `XMLIndex` index instead. Create also B-tree indexes on the columns of the structured index component.

Note:

- Queries that use SQL/XML function `XMLTable` can typically be rewritten to use the relational indexing tables of an `XMLIndex` structured component. These tables also contain some internal, system-defined columns. These internal columns might change in the future, so do not write code that depends on any assumptions about their existence or contents.
- Queries that use Oracle SQL function `XMLSequence` within a SQL `TABLE` collection expression, that is, `TABLE(XMLSequence(...))`, are *not* rewritten to use the indexing tables of an `XMLIndex` structured component. Oracle SQL function `XMLSequence` is *deprecated* in Oracle Database 11*g* Release 2; use standard SQL/XML function `XMLTable` instead.

  See *Oracle Database SQL Language Reference* for information about the SQL `TABLE` collection expression.

Ignore the Index Content Tables; They Are Transparent

Although the index content tables of an `XMLIndex` structured component are normal relational tables, they are also *read-only*: you cannot add or drop their columns or modify (insert, update, or delete) their rows.

You can thus generally ignore the relational index content tables. You cannot access them, other than to `DESCRIBE` them and create (secondary) indexes on them. You need never explicitly gather statistics on them. You need only collect statistics on the `XMLIndex` index itself or the base table on which the `XMLIndex` index is defined; statistics are collected and maintained on the index content tables transparently.

See Also: "Collecting Statistics on XMLIndex Objects for the Cost-Based Optimizer" on page 6-39

Data Type Considerations for XMLIndex Structured Component

The relational tables that are used for an `XMLIndex` structured component use SQL data types. XQuery expressions that are used in queries use XML data types (XML Schema data types and XQuery data types).

XQuery typing rules can automatically change the data type of a subexpression, to ensure coherence and type-checking. For example, if a document that is queried using

XPath expression `/PurchaseOrder/LineItem[@ItemNumber = 25]` is not XML schema-based, then the subexpression `@ItemNumber` is untyped, and it is then automatically cast to `xs:double` by the XQuery = comparison operator. To index this data using an `XMLIndex` structured component you must use `BINARY_DOUBLE` as the SQL data type.

This is a general rule. For an `XMLIndex` index with structured component to apply to a query, the data types must correspond. Table 6–5 shows the data-type correspondences.

Table 6–5 XML and SQL Data Type Correspondence for XMLIndex

XML Data Type	SQL Data Type
`xs:decimal`	`INTEGER` or `NUMBER`
`xs:double`	`BINARY_DOUBLE`
`xs:float`	`BINARY_FLOAT`
`xs:date`	`DATE`, `TIMESTAMP WITH TIMEZONE`
`xs:dateTime`	`TIMESTAMP`, `TIMESTAMP WITH TIMEZONE`
`xs:dayTimeDuration`	`INTERVAL DAY TO SECOND`
`xs:yearMonthDuration`	`INTERVAL YEAR TO MONTH`

Note: If the XML data type is `xs:date` or `xs:dateTime`, and if you know that the data that you will query and for which you are creating an index will *not* contain a time-zone component, then you can increase performance by using SQL data type `DATE` or `TIMESTAMP`. If the data might contain a time-zone component, then you must use SQL data type `TIMESTAMP WITH TIMEZONE`.

If the XML and SQL data types involved do not have a built-in one-to-one correspondence, then you must make them correspond (according to Table 6–5), in order for the index to be picked up for your query. There are two ways you can do this:

- **Make the index correspond to the query –** Define (or redefine) the column in the structured index component, so that it corresponds to the XML data type. For example, if a query that you want to index uses the XML data type `xs:double`, then define the index to use the corresponding SQL data type, `BINARY_DOUBLE`.
- **Make the query correspond to the index –** In your query, explicitly cast the relevant parts of an XQuery expression to data types that correspond to the SQL data types used in the index content table.

Example 6–3 and Example 6–4 show how you can cast an XQuery expression in your query to match the SQL data type used in the index content table.

Example 6–3 Making Query Data Compatible with Index Data – SQL Cast

```
SELECT count(*) FROM purchaseorder
  WHERE XMLCast(XMLQuery('$p/PurchaseOrder/LineItem/@ItemNumber'
                         PASSING OBJECT_VALUE AS "p" RETURNING CONTENT)
                 AS INTEGER)
        = 25;
```

Example 6–4 Making Query Data Compatible with Index Data – XQuery Cast

```
SELECT count(*) FROM purchaseorder
  WHERE XMLExists('$p/PurchaseOrder/LineItem[xs:decimal(@ItemNumber) = 25]'
                  PASSING OBJECT_VALUE AS "p");
```

Notice that the number `25` plays a different role in these two examples, even though in both cases it is the purchase-order item number. In Example 6–3, `25` is a SQL number of data type `INTEGER`; in Example 6–4, `25` is an XQuery number of data type `xs:decimal`.

In Example 6–3, the `XMLQuery` result is cast to SQL type `INTEGER`, which is compared with the SQL value 25. In Example 6–4, the value of attribute `ItemNumber` is cast (in XQuery) to the XML data type `xs:decimal`, which is compared with the XQuery value 25 and which corresponds to the SQL data type (`INTEGER`) used for the index. There are thus two different kinds of data-type conversion in these examples, but they both convert query data to make it type-compatible with the index content table.

> **See Also:** "Mapping XML Schema Data Types to SQL Data Types" on page 7-45 for information about the built-in correspondence between XML Schema data types and SQL data types

XMLIndex Unstructured Component

Unlike a B-tree index, which you define for a specific database column that represents an individual XML element or attribute, or the `XMLIndex` structured component, which applies to specific, structured document parts, the unstructured component of an `XMLIndex` index is, by default, very general. Unless you specify a more narrow focus by detailing specific XPath expressions to use or not to use in indexing, an unstructured `XMLIndex` component applies to *all possible XPath expressions* for your XML data.

The unstructured component of an `XMLIndex` index has three logical parts:

- A **path index** – This indexes the XML *tags* of a document and identifies its various document *fragments*.
- An **order index** – This indexes the hierarchical *positions* of the nodes in an XML document. It keeps track of parent–child, ancestor–descendant, and sibling relations.
- A **value index** – This indexes the *values* of an XML document. It provides lookup by either value equality or value range. A value index is used for values in query predicates (`WHERE` clause).

The unstructured component of an `XMLIndex` index uses a path table and a set of (local) secondary indexes on the path table, which implement the logical parts described above. Two secondary indexes are created automatically:

- A **pikey index**, which implements the logical indexes for both path and order.
- A real **value index**, which implements the logical value index.

You can modify these two indexes or create additional secondary indexes. The path table and its secondary indexes are all owned by the owner of the base table upon which the `XMLIndex` index is created.

The pikey index handles paths and order relationships together, which gives the best performance in most cases. If you find in some particular case that the value index is not picked up when think it should be, you can replace the pikey index with separate indexes for the paths and order relationships. Such (optional) indexes are called **path**

id and **order key** indexes, respectively. For best results, contact Oracle Support if you find that the pikey index is not sufficient for your needs in some case.

The path table contains one row for each indexed node in the XML document. For each indexed node, the **path table** stores:

- The corresponding *rowid* of the table that stores the document.
- A *locator*, which provides fast access to the corresponding document fragment. For binary XML storage of XML schema-based data, it also stores data-type information.
- An *order key*, to record the hierarchical position of the node in the document. You can think of this as a Dewey decimal key like that used in library cataloging and Internet protocol SNMP. In such a system, the key `3.21.5` represents the node position of the fifth child of the twenty-first child of the third child of the document root node.
- An identifier that represents an XPath *path* to the node.
- The effective *text value* of the node.

Table 6–6 shows the main information[2] that is in the path table.

Table 6–6 XMLIndex Path Table

Column	Data Type	Description
`PATHID`	`RAW(8)`	Unique identifier for the XPath path to the node.
`RID`	`ROWID`	Rowid of the table used to store the XML data.
`ORDER_KEY`	`RAW(1000)`	Decimal order key that identifies the hierarchical position of the node. (Document ordering is preserved.)
`LOCATOR`	`RAW(2000)`	Fragment-location information. Used for fragment extraction. For binary XML storage of XML schema-based data, data-type information is also stored here.
`VALUE`	`VARCHAR2(4000)`	Effective text value the node.

The pikey index uses path table columns `PATHID`, `RID`, and `ORDER_KEY` to represent the path and order indexes. An optional path id index uses columns `PATHID` and `RID` to represent the path index. A value index is an index on the `VALUE` column.

Example 6–5 explores the contents of the path table for two purchase-order documents.

Example 6–5 Path Table Contents for Two Purchase Orders

```
<PurchaseOrder>
 <Reference>SBELL-2002100912333601PDT</Reference>
 <Actions>
  <Action>
   <User>SVOLLMAN</User>
  </Action>
 </Actions>
 . . .
</PurchaseOrder>

<PurchaseOrder>
 <Reference>ABEL-20021127121040897PST</Reference>
```

[2] The actual path table implementation may be slightly different.

```
 <Actions>
  <Action>
   <User>ZLOTKEY</User>
  </Action>
  <Action>
   <User>KING</User>
  </Action>
 </Actions>
 . . .
</PurchaseOrder>
```

An XMLIndex index on an XMLType table or column storing these purchase orders includes a path table that has one row for each indexed node in the XML documents. Suppose that the system assigns the following PATHIDs when indexing the nodes according to their XPath expressions:

PATHID	Indexed XPath
1	/PurchaseOrder
2	/PurchaseOrder/Reference
3	/PurchaseOrder/Actions
4	/PurchaseOrder/Actions/Action
5	/PurchaseOrder/Actions/Action/User

The resulting path table would then be something like this (column LOCATOR is not shown):

PATHID	RID	ORDER_KEY	VALUE
1	R1	1	SBELL-2002100912333601PDTSVOLLMAN
2	R1	1.1	SBELL-2002100912333601PDT
3	R1	1.2	SVOLLMAN
4	R1	1.2.1	SVOLLMAN
5	R1	1.2.1.1	SVOLLMAN
1	R2	1	ABEL-20021127121040897PSTZLOTKEYKING
2	R2	1.1	ABEL-20021127121040897PST
3	R2	1.2	ZLOTKEYKING
4	R2	1.2.1	ZLOTKEY
5	R2	1.2.1.1	ZLOTKEY
4	R2	1.2.2	KING
5	R2	1.2.2.1	KING

Ignore the Path Table – It Is Transparent

Though you can create secondary indexes on path-table columns, you can generally ignore the path table itself. You *cannot access* the path table, other than to DESCRIBE it and create (secondary) indexes on it. You need never explicitly gather statistics on the path table. You need only collect statistics on the XMLIndex index or the base table on which the XMLIndex index is defined; statistics are collected and maintained on the path table and its secondary indexes transparently.

See Also: "Collecting Statistics on XMLIndex Objects for the Cost-Based Optimizer" on page 6-39

Column VALUE of an XMLIndex Path Table

A secondary index on column `VALUE` is used with XPath expressions in a `WHERE` clause that have predicates involving string matches. For example:

```
/PurchaseOrder[Reference/text() = "SBELL-2002100912333601PDT"]
```

Column `VALUE` stores the **effective text value** of an element or an attribute node—comments and processing instructions are ignored during indexing.

- For an *attribute*, the effective text value is the attribute value.
- For a *simple* element (an element that has no children), the effective text value is the concatenation of all of the text nodes of the element.
- For a *complex* element (an element that has children), the effective text value is the concatenation of (1) the text nodes of the element itself and (2) the effective text values of all of its simple-element descendants. (This is a recursive definition.)

The effective text value is limited (truncated), however, to 4000 bytes for a simple element or attribute and to 80 bytes for a complex element.

Column `VALUE` is a fixed size, `VARCHAR2(4000)`. Any overflow (beyond 4000 bytes) during index creation or update is truncated, but the `LOCATOR` value for that row is then flagged so that the full value can be retrieved from the base table when needed.

In addition to the 4000-byte limit for column `VALUE`, there is a limit on the size of a key for the secondary index created on this column. This is the case for B-tree and function-based indexes as well; it is not an `XMLIndex` limitation. The index-key size limit is a function of the block size for your database. It is this limit that determines how much of `VALUE` is indexed.

Thus, only the first 4000 bytes of the effective text value are stored in column `VALUE`, and only the first *N* bytes of column `VALUE` are indexed, where *N* is the index-key size limit ($N < 4000$). Because of the index-key size limit, the index on column `VALUE` acts only as a *preliminary filter* for the effective text value.

For example, suppose that your database block size requires that the `VALUE` index be no larger than 800 bytes, so that only the first 800 bytes of the effective text value is indexed. The first 800 bytes of the effective text value is first tested, using `XMLIndex`, and only if that text prefix matches the query value is the rest of the effective text value tested.

The secondary index on column `VALUE` is an index on SQL function `substr` (substring equality), because that function is used to test the text prefix. This function-based index is created automatically as part of the implementation of `XMLIndex` for column `VALUE`.

For example, the XPath expression `/PurchaseOrder[Reference/text() = :1]` in a query `WHERE` clause might, in effect, be rewritten to a test something like this:

```
substr(VALUE, 1 800) = substr(:1, 1, 800) AND VALUE = :1;
```

This conjunction contains two parts, which are processed from left to right. The first test uses the index on function `substr` as a preliminary filter, to eliminate text whose first 800 bytes do not match the first 800 bytes of the value of bind variable `:1`.

Only the first test uses an index—the full value of column `VALUE` is not indexed. After preliminary filtering by the first test, the second test checks the entire effective text

value—that is, the full value of column VALUE—for full equality with the value of :1. This check does not use an index.

Even if only the first 800 bytes of text is indexed, it is important for query performance that up to 4000 bytes be stored in column VALUE, because that provides quick, direct access to the data, instead of requiring, for example, extracting it from deep within a CLOB-instance XML document. If the effective text value is greater than 4000 bytes, then the second test in the WHERE-clause conjunction requires accessing the base-table data.

Note that neither the VALUE column 4000-byte limit nor the index-key size affect query results in any way; they can affect only performance.

Note: Because of the possibility of the VALUE column being truncated, an Oracle Text CONTEXT index created on the VALUE column might return incorrect results.

As mentioned, XMLIndex can be used with XML schema-based data. If an XML schema specifies a defaultValue value for a given element or attribute, and a particular document does not specify a value for that element or attribute, then the defaultValue value is used for the VALUE column.

Secondary Indexes on Column VALUE

Even if you do not specify a secondary index for column VALUE when you create an XMLIndex index, a default secondary index is created on column VALUE. This default index has the default properties—in particular, it is an index for *text* (string-valued) data only.

You can, however, create a VALUE index of a different type. For example, you can create a number-valued index if that is appropriate for many of your queries. You can create multiple secondary indexes on the VALUE column. An index of a particular type is used only when it is appropriate. For example, a number-valued index is used only when the VALUE column is a number; it is ignored for other values. Secondary indexes on path-table columns are treated like any other secondary indexes—you can alter them, drop them, mark them unusable, and so on.

See Also:

- "Using XMLIndex with an Unstructured Component" on page 6-19 for examples of creating secondary indexes on column VALUE
- "PARAMETERS Clause for CREATE INDEX and ALTER INDEX" on page 6-41 for the syntax of the PARAMETERS clause

XPath Expressions that Are Not Indexed by an XMLIndex Unstructured Component

The following types of XPath expressions are *not* indexed by XMLIndex:

- Applications of XPath functions, *except* ora:contains. In particular, user-defined XPath functions are *not* indexed.
- Axes other than child, descendant, and attribute, that is, axes parent, ancestor, following-sibling, preceding-sibling, following, preceding, and ancestor-or-self.
- Expressions using the union operator, | (vertical bar).

Creating, Dropping, Altering, and Examining an XMLIndex Index

You create an XMLIndex index by declaring the index type to be XDB.XMLIndex, as illustrated in Example 6–6.

Example 6–6 Creating an XMLIndex Index on XMLType Unstructured Storage

```
CREATE INDEX po_xmlindex_ix ON po_clob (OBJECT_VALUE) INDEXTYPE IS XDB.XMLIndex;
```

This creates an XMLIndex index named po_xmlindex_ix on XMLType table po_clob. The index has only an unstructured component, no structured component.

You specify inclusion of a structured component in an XMLIndex index by including a *structured_clause* in the PARAMETERS clause. You specify inclusion of an unstructured component by including a *path_table_clause* in the PARAMETERS clause. You can do this when you create the XMLIndex index or when you modify it. If, as in Example 6–6, you specify neither a *structured_clause* nor a *path_table_clause*, then *only* an unstructured component is included.

If an XMLIndex index has both an unstructured and a structured component, then you can drop either of these components using ALTER INDEX.

See Also:

- "PARAMETERS Clause for CREATE INDEX and ALTER INDEX" on page 6-41
- "structured_clause ::=" on page 6-44
- "path_table_clause ::=" on page 6-43
- "drop_path_table_clause ::=" on page 6-44
- "alter_index_group_clause ::=" on page 6-45

You can obtain the name of an XMLIndex index on a particular XMLType table (or column), as shown in Example 6–7. You can also select INDEX_NAME from DBA_INDEXES or ALL_INDEXES, as appropriate.

Example 6–7 Obtaining the Name of an XMLIndex Index on a Particular Table

```
SELECT INDEX_NAME FROM USER_INDEXES
  WHERE TABLE_NAME = 'PO_CLOB' AND ITYP_NAME = 'XMLINDEX';

INDEX_NAME
---------------
PO_XMLINDEX_IX

1 row selected.
```

You rename or drop an XMLIndex index just as you would any other index, as illustrated in Example 6–8. This renaming changes the name of the XMLIndex index only. It does not change the name of the path table—you can rename the path table separately.

Example 6–8 Renaming and Dropping an XMLIndex Index

```
ALTER INDEX po_xmlindex_ix RENAME TO new_name_ix;

DROP INDEX new_name_ix;
```

Similarly, you can change other index properties using other ALTER INDEX options, such as REBUILD. XMLIndex is no different from other index types in this respect.

The RENAME clause of an ALTER INDEX statement for XMLIndex applies only to the XMLIndex index itself. To rename the path table and secondary indexes, you must determine the names of these objects and use appropriate ALTER TABLE or ALTER INDEX statements on them directly. Similarly, to retrieve the physical properties of the secondary indexes or alter them in any other way, you must obtain their names, as in Example 6–15.

Example 6–6 shows how to create an XMLIndex index on unstructured storage. You can also create an XMLIndex index on CLOB portions of hybrid XMLType storage, that is, on CLOB data that is embedded within object-relational storage. Example 6–9 illustrates this. It assumes that the XML schema used maps the LineItems element to CLOB, such as is shown in Example 6–10.

Example 6–9 Creating an XMLIndex Index on XMLType Hybrid Storage

```
CREATE INDEX po_xmlindex_hybrid_ix ON purchaseorder
  (extract(OBJECT_VALUE, '/PurchaseOrder/LineItems'))
  INDEXTYPE IS XDB.XMLIndex;
```

Example 6–10 XML Schema Fragment that Maps LineItems to CLOB Storage

```
<xs:element name="LineItems" type="LineItemsType" xdb:SQLName="LINEITEMS"
           xdb:SQLType="CLOB"/>
```

See Also: "PARAMETERS Clause for CREATE INDEX and ALTER INDEX" on page 6-41 for the syntax of the PARAMETERS clause

Using XMLIndex with an Unstructured Component

This section covers operations you can perform on an XMLIndex index that has an unstructured component (whether or not it also has a structured component)—see "XMLIndex Unstructured Component" on page 6-13.

To include an unstructured component in an XMLIndex index, you use a *path_table_clause* in the PARAMETERS clause when you create or modify the XMLIndex index—see "path_table_clause ::=" on page 6-43.

If you do not specify a *structured* component, then the index will have an unstructured component, even if you do not specify the path table. It is however generally a good idea to specify the path table, so that it has a recognizable, user-oriented name that you can refer to in other XMLIndex operations.

Example 6–11 shows how to name the path table ("my_path_table") when creating an XMLIndex index with an unstructured component.

Example 6–11 Naming the Path Table of an XMLIndex Index

```
CREATE INDEX po_xmlindex_ix ON po_clob (OBJECT_VALUE) INDEXTYPE IS XDB.XMLIndex
  PARAMETERS ('PATH TABLE my_path_table');
```

If you do not name the path table then its name is generated by the system, using the index name you provide to CREATE INDEX as a base. Example 6–12 shows this for the XMLIndex index created in Example 6–6.

Example 6–12 Determining the System-Generated Name of an XMLIndex Path Table

```
SELECT PATH_TABLE_NAME FROM USER_XML_INDEXES
```

```
  WHERE TABLE_NAME = 'PO_CLOB' AND INDEX_NAME = 'PO_XMLINDEX_IX';

PATH_TABLE_NAME
------------------------------
SYS67567_PO_XMLINDE_PATH_TABLE

1 row selected.
```

By default, the storage options of a path table and its secondary indexes are derived from the storage properties of the base table on which the `XMLIndex` index is created. You can specify different storage options by using a `PARAMETERS` clause when you create the index, as shown in Example 6–13. The `PARAMETERS` clause of `CREATE INDEX` (and `ALTER INDEX`) must be between single quotation marks (`'`).

> **See Also:** "PARAMETERS Clause for CREATE INDEX and ALTER INDEX" on page 6-41 for the syntax of the `PARAMETERS` clause

Example 6–13 Specifying Storage Options When Creating an XMLIndex Index

```
CREATE INDEX po_xmlindex_ix ON po_clob (OBJECT_VALUE) INDEXTYPE IS XDB.XMLIndex
  PARAMETERS
    ('PATH TABLE po_path_table
      (PCTFREE 5 PCTUSED 90 INITRANS 5
       STORAGE (INITIAL 1k NEXT 2k MINEXTENTS 3 BUFFER_POOL KEEP)
       NOLOGGING ENABLE ROW MOVEMENT PARALLEL 3)
      PIKEY INDEX po_pikey_ix (LOGGING PCTFREE 1 INITRANS 3)
      VALUE INDEX po_value_ix (LOGGING PCTFREE 1 INITRANS 3)');
```

Because `XMLIndex` is a logical *domain* index, not a physical index, all physical attributes are either zero (`0`) or `NULL`.

If an `XMLIndex` index has both an unstructured and a structured component, then you can use `ALTER INDEX` to drop the unstructured component. To do this, you drop the path table. Example 6–14 illustrates this. (This assumes that you also have a structured component—Example 6–22 results in an index with both structured and unstructured components.)

Example 6–14 Dropping an XMLIndex Unstructured Component

```
ALTER INDEX po_xmlindex_ix PARAMETERS('DROP PATH TABLE');
```

Note that, in addition to specifying storage options for the path table, Example 6–13 names the secondary indexes on the path table.

Like the name of the path table, the names of the secondary indexes on the path-table columns are generated automatically using the index name as a base, unless you specify them in the `PARAMETERS` clause. Example 6–15 illustrates this, and shows how you can determine these names using public view `USER_IND_COLUMNS`. It also shows that the pikey index uses three columns.

Example 6–15 Determining the Names of the Secondary Indexes of an XMLIndex Index

```
SELECT INDEX_NAME, COLUMN_NAME, COLUMN_POSITION FROM USER_IND_COLUMNS
  WHERE TABLE_NAME IN (SELECT PATH_TABLE_NAME FROM USER_XML_INDEXES
                       WHERE INDEX_NAME = 'PO_XMLINDEX_IX')
  ORDER BY INDEX_NAME, COLUMN_NAME;

INDEX_NAME                     COLUMN_NAME  COLUMN_POSITION
------------------------------ ------------ ---------------
SYS67563_PO_XMLINDE_PIKEY_IX   ORDER_KEY                  3
```

```
SYS67563_PO_XMLINDE_PIKEY_IX   PATHID                    2
SYS67563_PO_XMLINDE_PIKEY_IX   RID                       1
SYS67563_PO_XMLINDE_VALUE_IX   SYS_NC00006$              1

4 rows selected.
```

> **See Also:** Example 6–21 on page 6-22 for a similar, but more complex example

Creating Additional Secondary Indexes on an XMLIndex Path Table

This section adds extra secondary indexes to the `XMLIndex` index created in Example 6–13.

You can create any number of additional secondary indexes on the `VALUE` column of the path table of an `XMLIndex` index. These can be of different types, including function-based indexes and Oracle Text indexes.

Whether or not a given index is used for a given element occurrence when processing a query is determined by whether it is of the appropriate type for that value and whether it is cost-effective to use it.

Example 6–16 creates a function-based index on column `VALUE` of the path table using SQL function `substr`. This might be useful if your queries often use `substr` applied to the text nodes of XML elements.

Example 6–16 Creating a Function-Based Index on Path-Table Column VALUE

```
CREATE INDEX fn_based_ix ON po_path_table (substr(VALUE, 1, 100));
```

If you have many elements whose text nodes represent numeric values, then it can make sense to create a numeric index on the column `VALUE`. However, doing so directly, in a manner analogous to Example 6–16, raises an ORA-01722 error (invalid number) if some of the element values are *not* numbers. This is illustrated in Example 6–17.

Example 6–17 Trying to Create a Numeric Index on Path-Table Column VALUE Directly

```
CREATE INDEX direct_num_ix ON po_path_table (to_binary_double(VALUE));
CREATE INDEX direct_num_ix ON po_path_table (to_binary_double(VALUE))
                                             *
ERROR at line 1:
ORA-01722: invalid number
```

What is needed is an index that is used for numeric-valued elements but is ignored for element occurrences that do not have numeric values. Procedure `createNumberIndex` of package `DBMS_XMLINDEX` exists specifically for this purpose. You pass it the names of the database schema, the `XMLIndex` index, and the numeric index to be created. Creation of a numeric index is illustrated in Example 6–18.

Example 6–18 Creating a Numeric Index on Column VALUE with Procedure createNumberIndex

```
CALL DBMS_XMLINDEX.createNumberIndex('OE', 'PO_XMLINDEX_IX', 'API_NUM_IX');
```

Note that because such an index is specifically designed to ignore elements that do not have numeric values, its use does not detect their presence. If there are non-numeric elements and, for whatever reason, the `XMLIndex` index is not used in some query, then an `ORA-01722` error is raised. However, if the index is used, no such error is

raised, because the index ignores non-numeric data. As always, the use of an index never changes the result set—it never gives you different results, but use of an index can prevent you from detecting erroneous data.

Creating a date-valued index is similar to creating a numeric index; you use procedure `DBMS_XMLINDEX.createDateIndex`. Example 6–19 shows this.

Example 6–19 Creating a Date Index on Column VALUE with Procedure createDateIndex

```
CALL DBMS_XMLINDEX.createDateIndex('OE', 'PO_XMLINDEX_IX', 'API_DATE_IX',
                                   'dateTime');
```

Example 6–20 creates an Oracle Text `CONTEXT` index on column `VALUE`. This is useful for full-text queries on text values of XML elements. XPath predicates that use XPath function `ora:contains` are rewritten to applications of Oracle SQL function `contains` on column `VALUE`. If a `CONTEXT` index is defined on column `VALUE`, then it is used during predicate evaluation. An Oracle Text index is independent of all other `VALUE`-column indexes.

Example 6–20 Creating an Oracle Text CONTEXT Index on Path-Table Column VALUE

```
CREATE INDEX po_otext_ix ON po_path_table (VALUE)
  INDEXTYPE IS CTXSYS.CONTEXT PARAMETERS('TRANSACTIONAL');
```

See Also:

- "From ora:contains to contains" on page 12-24 for information about parameter `TRANSACTIONAL`
- "Column VALUE of an XMLIndex Path Table" on page 6-16 for information about the possibility of an Oracle Text `CONTEXT` index created on the `VALUE` column returning incorrect results
- "Oracle Text Indexes on XML Data" on page 6-48

The query in Example 6–21 shows all of the secondary indexes created on the path table of an `XMLIndex` index. The indexes created explicitly are in bold. Note in particular that some indexes, such as the function-based index created on column `VALUE`, do not appear as such; the column name listed for such an index is a system-generated name such as `SYS_NC00007$`. You *cannot* see these columns by executing a query with `COLUMN_NAME = 'VALUE'` in the `WHERE` clause.

Example 6–21 Showing All Secondary Indexes on an XMLIndex Path Table

```
SELECT c.INDEX_NAME, c.COLUMN_NAME, c.COLUMN_POSITION, e.COLUMN_EXPRESSION
  FROM USER_IND_COLUMNS c LEFT OUTER JOIN USER_IND_EXPRESSIONS e
    ON (c.INDEX_NAME = e.INDEX_NAME)
  WHERE c.TABLE_NAME IN (SELECT PATH_TABLE_NAME FROM USER_XML_INDEXES
                         WHERE INDEX_NAME = 'PO_XMLINDEX_IX')
  ORDER BY c.INDEX_NAME, c.COLUMN_NAME;

INDEX_NAME           COLUMN_NAME  COLUMN_POSITION COLUMN_EXPRESSION
-------------------- ------------ --------------- ----------------------
API_DATE_IX          SYS_NC00009$               1 SYS_EXTRACT_UTC(SYS_XMLCONV("V
                                                   ALUE",3,8,0,0,181))
API_NUM_IX           SYS_NC00008$               1 TO_BINARY_DOUBLE("VALUE")
FN_BASED_IX          SYS_NC00007$               1 SUBSTR("VALUE",1,100)
PO_OTEXT_IX          VALUE                      1
PO_PIKEY_IX          ORDER_KEY                  3
PO_PIKEY_IX          PATHID                     2
```

```
PO_PIKEY_IX          RID                        1
PO_VALUE_IX          SYS_NC00006$               1 SUBSTRB("VALUE",1,1599)

8 rows selected.
```

See Also:

- *Oracle Database PL/SQL Packages and Types Reference* for information on PL/SQL procedures `createNumberIndex` and `createDateIndex` in package `DBMS_XMLINDEX`
- "Oracle Text Indexes Are Used Independently of Other Indexes" on page 6-49 for information on using Oracle Text indexes

Using XMLIndex with a Structured Component

To include a structured component in an `XMLIndex` index, you use a *`structured_clause`* in the `PARAMETERS` clause when you create or modify the `XMLIndex` index—see "structured_clause ::=" on page 6-44.

A *`structured_clause`* specifies the structured islands that you want to index. You use the keyword `GROUP` to specify each structured island: an island thus corresponds syntactically to a structure **group**. If you specify no group explicitly, then the predefined group `DEFAULT_GROUP` is used. For `ALTER INDEX`, you precede the `GROUP` keyword with the modification operation keyword: `ADD_GROUP` specifies a new group (island); `DROP_GROUP` deletes a group.

Why have multiple groups within a single index, instead of simply using multiple `XMLIndex` indexes? The reason is that `XMLIndex` is a domain index, and you can create only one domain index of a given type on a given database column.

The syntax for defining a structure group, that is, indexing a structured island, is similar to the syntax for invoking SQL/XML *function* `XMLTable`: you use keywords `XMLTable` and `COLUMNS` to define relational columns, and you use multilevel chaining of `XMLTable` to handle collections.

Example 6–22 shows the creation of an `XMLIndex` index with only an unstructured component. An unstructured component is created because the `PARAMETERS` clause explicitly names the path table.

Example 6–22 then uses `ALTER INDEX` to add a structured component (group) named `po_item`. This structure group includes two relational tables, each specified with keyword `XMLTable`.

Example 6–22 XMLIndex Index: Adding a Structured Component

```
CREATE INDEX po_xmlindex_ix ON po_clob (OBJECT_VALUE)
  INDEXTYPE IS XDB.XMLIndex PARAMETERS ('PATH TABLE path_tab');

BEGIN
  DBMS_XMLINDEX.registerParameter(
    'myparam',
    'ADD_GROUP GROUP po_item
       XMLTable po_idx_tab ''/PurchaseOrder''
         COLUMNS reference   VARCHAR2(30)  PATH ''Reference'',
                 requestor   VARCHAR2(30)  PATH ''Requestor'',
                 username    VARCHAR2(30)  PATH ''User'',
                 lineitem    XMLType       PATH ''LineItems/LineItem'' VIRTUAL
       XMLTable po_index_lineitem ''/LineItem'' PASSING lineitem
         COLUMNS itemno      BINARY_DOUBLE PATH ''@ItemNumber'',
                 description VARCHAR2(256) PATH ''Description'',
```

```
                 partno      VARCHAR2(14)  PATH ''Part/@Id'',
                 quantity    BINARY_DOUBLE PATH ''Part/@Quantity'',
                 unitprice   BINARY_DOUBLE PATH ''Part/@UnitPrice''');
END;
/

ALTER INDEX po_xmlindex_ix PARAMETERS('PARAM myparam');
```

The top-level table, `po_idx_tab`, has columns `reference`, `requestor`, `username`, and `lineitem`. Column `lineitem` is of type `XMLType`. It represents a collection, so it is passed to the second `XMLTable` construct to form the second-level relational table, `po_index_lineitem`, which has columns `itemno`, `description`, `partno`, `quantity`, and `unitprice`.

The keyword `VIRTUAL` is *required* for an `XMLType` column. It specifies that the `XMLType` column itself is not materialized: its data is stored in the `XMLIndex` index only in the form of the relational columns specified by its corresponding `XMLTable` table.

You cannot create more than one `XMLType` column in a given `XMLTable` clause. To achieve that effect, you must instead define an additional group.

Example 6–22 also illustrates the use of a registered parameter string in the `PARAMETERS` clause. It uses PL/SQL procedure `DBMS_INDEX.registerParameter` to register the parameters string named `myparam`. Then it uses `ALTER INDEX` to update the index parameters to include those in the string `myparam`.

If an `XMLIndex` index has both an unstructured and a structured component, then you can use `ALTER INDEX` to drop the structured component. You do this by dropping *all* of the structure groups that compose the structured component. Example 6–23 shows how to drop the structured component that was added in Example 6–22, by dropping its only structure group, `po_item`.

Example 6–23 Dropping an XMLIndex Structured Component

```
ALTER INDEX po_xmlindex_ix PARAMETERS('DROP_GROUP GROUP po_item');
```

As indicated in section "XMLIndex Structured Component" on page 6-10, because the tables used for the structured component of an `XMLIndex` index are normal relational tables, you can index them using any standard relational indexes.

Example 6–24 and Example 6–25 illustrate this: Example 6–24 creates a B-tree index on the `reference` column of the index content table (structured fragment) for the `XMLIndex` index of Example 6–22. Example 6–25 creates an Oracle Text `CONTEXT` index on the `description` column and then uses a full-text query on the content.

Example 6–24 Creating a B-Tree Index on an XMLIndex Index Content Table

```
CREATE INDEX idx_tab_ref_ix ON po_idx_tab (reference);
```

Example 6–25 Oracle Text CONTEXT Index on an XMLIndex Index Content Table

```
CREATE INDEX idx_tab_desc_ix ON po_index_lineitem (description)
  INDEXTYPE IS CTXSYS.CONTEXT PARAMETERS ('transactional');

SELECT XMLQuery('/PurchaseOrder/LineItems/LineItem'
                PASSING OBJECT_VALUE RETURNING CONTENT)
  FROM po_clob
    WHERE XMLExists('/PurchaseOrder/LineItems/LineItem
                     [ora:contains(Description, "Picnic") > 0]'
                    PASSING OBJECT_VALUE)
```

```
AND XMLExists('/PurchaseOrder[User="SBELL"]' PASSING OBJECT_VALUE);
```

Example 6–26 shows the creation of an XMLIndex index that has only a structured component (no path table clause) and that uses the XMLNamespaces clause to specify namespaces. It specifies that the index data be compressed and use tablespace SYSAUX. The example assumes a binary XML table po_binxml with non XML schema-based data.

Example 6–26 XMLIndex with Only a Structured Component and using Namespaces

```
CREATE INDEX po_struct ON po_binxml (OBJECT_VALUE) INDEXTYPE IS XDB.XMLIndex
  PARAMETERS ('XMLTable po_ptab
                (TABLESPACE "SYSAUX" COMPRESS FOR OLTP)
                XMLNAMESPACES(DEFAULT ''http://www.example.com/po''),
                ''/purchaseOrder''
                COLUMNS orderdate   DATE          PATH ''@orderDate'',
                        id          BINARY_DOUBLE PATH ''@id'',
                        items       XMLType       PATH ''items/item'' VIRTUAL
              XMLTable li_tab
                (TABLESPACE "SYSAUX" COMPRESS FOR OLTP)
                XMLNAMESPACES(DEFAULT ''http://www.example.com/po''),
                ''/item'' PASSING items
                COLUMNS partnum     VARCHAR2(15)  PATH ''@partNum'',
                        description CLOB          PATH ''productName'',
                        usprice     BINARY_DOUBLE PATH ''USPrice'',
                        shipdat     DATE          PATH ''shipDate''');
```

See Also:

- Example 6–30, "Using a Structured XMLIndex Component for a Query with Two Predicates" on page 6-28
- "Oracle Text Indexes on XML Data" on page 6-48
- "Using a Registered PARAMETERS Clause for XMLIndex" on page 6-41
- "structured_clause ::=" on page 6-44
- "Usage of XMLIndex_xmltable_clause" on page 6-48 for information about an XMLType column in an XMLTable clause
- "Usage of column_clause" on page 6-48 for information about keywords COLUMNS and VIRTUAL
- "Data Type Considerations for XMLIndex Structured Component" on page 6-11

How to Tell Whether XMLIndex is Used

It is at query compile time that Oracle Database determines whether or not a given XMLIndex index can be used, that is, whether the query can be rewritten into a query against the index.

For an unstructured XMLIndex component, if it cannot be determined at compile time that an XPath expression in the query is a subset of the paths you specified to be used for XMLIndex indexing, then the unstructured component of the index is not used.

For example, if the path /PurchaseOrder/LineItems//* is included for indexing, then a query with /PurchaseOrder/LineItems/LineItem/Description can use the index, but a query with //Description cannot. The latter also matches potential Description elements that are not children of

/PurchaseOrder/LineItems, and it is not possible at compile time to know if such additional Description elements are present in the data.

To know whether a particular XMLIndex index has been used in resolving a query, you can examine an execution plan for the query.

- If the *unstructured* component of the index is used, then its path table, order key, or path id is referenced in the execution plan. The execution plan does *not* directly indicate that a domain index was used; it does *not* refer to the XMLIndex index by name. See Example 6–27 on page 6-26 and Example 6–29 on page 6-27.
- If the *structured* component of the index is used, then one or more of its index content tables is called out in the execution plan. See Example 6–30 on page 6-28 and Example 6–31 on page 6-29.

See Also:

- *Oracle Database SQL Language Reference*
- *Oracle Database Performance Tuning Guide*

Example 6–27 shows that the XMLIndex index created in Example 6–11 is used in a particular query. The reference to MY_PATH_TABLE in the execution plan here indicates that the XMLIndex index (created in Example 6–11) is used in this query. Similarly, reference to columns LOCATOR, ORDER_KEY, and PATHID indicates the same thing.

Example 6–27 Checking Whether an XMLIndex Unstructured Component Is Used

```
SET AUTOTRACE ON EXPLAIN

SELECT XMLQuery('/PurchaseOrder/Requestor' PASSING OBJECT_VALUE RETURNING CONTENT) FROM po_clob
  WHERE XMLExists('/PurchaseOrder[Reference="SBELL-2002100912333601PDT"]' PASSING OBJECT_VALUE);

XMLQUERY('/PURCHASEORDER/REQUESTOR'PASSINGOBJECT_VALUERETURNINGCONTENT)
-----------------------------------------------------------------------
<Requestor>Sarah J. Bell</Requestor>

1 row selected.


Execution Plan
. . .
-----------------------------------------------------------------------------------------------------------
| Id  | Operation                       | Name                          | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                |                               |     1 |    24 |    28   (4)| 00:00:01 |
|   1 |  SORT GROUP BY                  |                               |     1 |  3524 |            |          |
|*  2 |   TABLE ACCESS BY INDEX ROWID   | MY_PATH_TABLE                 |     2 |  7048 |     3   (0)| 00:00:01 |
|*  3 |    INDEX RANGE SCAN             | SYS67616_PO_XMLINDE_PIKEY_IX  |     1 |       |     2   (0)| 00:00:01 |
|   4 |  NESTED LOOPS                   |                               |     1 |    24 |    28   (4)| 00:00:01 |
|   5 |   VIEW                          | VW_SQ_1                       |     1 |    12 |    26   (0)| 00:00:01 |
|   6 |    HASH UNIQUE                  |                               |     1 |  5046 |            |          |
|   7 |     NESTED LOOPS                |                               |     1 |  5046 |    26   (0)| 00:00:01 |
|*  8 |      TABLE ACCESS BY INDEX ROWID| MY_PATH_TABLE                 |     1 |  3524 |    24   (0)| 00:00:01 |
|*  9 |       INDEX RANGE SCAN          | SYS67616_PO_XMLINDE_VALUE_IX  |    73 |       |     1   (0)| 00:00:01 |
|* 10 |      TABLE ACCESS BY INDEX ROWID| MY_PATH_TABLE                 |     1 |  1522 |     2   (0)| 00:00:01 |
|* 11 |       INDEX RANGE SCAN          | SYS67616_PO_XMLINDE_PIKEY_IX  |     1 |       |     1   (0)| 00:00:01 |
|  12 |   TABLE ACCESS BY USER ROWID    | PO_CLOB                       |     1 |    12 |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
```

```
   2 - filter(SYS_XMLI_LOC_ISNODE("SYS_P0"."LOCATOR")=1)
   3 - access("SYS_P0"."RID"=:B1 AND "SYS_P0"."PATHID"=HEXTORAW('76E2') )
   8 - filter("SYS_P4"."VALUE"='SBELL-2002100912333601PDT' AND "SYS_P4"."PATHID"=HEXTORAW('4F8C')  AND
              SYS_XMLI_LOC_ISNODE("SYS_P4"."LOCATOR")=1)
   9 - access(SUBSTRB("VALUE",1,1599)='SBELL-2002100912333601PDT')
  10 - filter(SYS_XMLI_LOC_ISNODE("SYS_P2"."LOCATOR")=1)
  11 - access("SYS_P4"."RID"="SYS_P2"."RID" AND "SYS_P2"."PATHID"=HEXTORAW('4E36')  AND
              "SYS_P2"."ORDER_KEY"<"SYS_P4"."ORDER_KEY")
       filter("SYS_P4"."ORDER_KEY"<SYS_ORDERKEY_MAXCHILD("SYS_P2"."ORDER_KEY") AND
              SYS_ORDERKEY_DEPTH("SYS_P2"."ORDER_KEY")+1=SYS_ORDERKEY_DEPTH("SYS_P4"."ORDER_KEY"))
. . .
```

Given the name of a path table from an execution plan such as this, you can obtain the name of its `XMLIndex` index as shown in Example 6–28. (This is more or less opposite to the query in Example 6–12.)

Example 6–28 Obtaining the Name of an XMLIndex Index from Its Path-Table Name

```
SELECT INDEX_NAME FROM USER_XML_INDEXES WHERE PATH_TABLE_NAME = 'MY_PATH_TABLE';

INDEX_NAME
------------------------------
PO_XMLINDEX_IX

1 row selected.
```

`XMLIndex` can be used for XPath expressions in the `SELECT` list, the `FROM` list, and the `WHERE` clause of a query, and it is useful for SQL/XML functions `XMLQuery`, `XMLTable`, `XMLExists`, and `XMLCast`. Unlike function-based indexes (and `CTXXPath` indexes, which are deprecated), `XMLIndex` indexes can be used when you extract data from an XML fragment in a document.

Example 6–29 illustrates this.

Example 6–29 Extracting Data from an XML Fragment using XMLIndex

```
SET AUTOTRACE ON EXPLAIN

SELECT li.description, li.itemno
  FROM po_clob, XMLTable('/PurchaseOrder/LineItems/LineItem'
                         PASSING OBJECT_VALUE
                         COLUMNS "DESCRIPTION" VARCHAR(40) PATH 'Description',
                                 "ITEMNO"      INTEGER     PATH '@ItemNumber') li
  WHERE XMLExists('/PurchaseOrder[Reference="SBELL-2002100912333601PDT"]'
                  PASSING OBJECT_VALUE);

DESCRIPTION                                ITEMNO
---------------------------------------- ----------
A Night to Remember                               1
The Unbearable Lightness Of Being                 2
Sisters                                           3

3 rows selected.

Execution Plan

----------------------------------------------------------------------------------------------------
| Id  | Operation                        | Name                        | Rows  | Bytes |Cost (%CPU)| Time     |
----------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                 |                             |     1 |  1546 |   30   (4)|00:00:01 |
|*  1 |  FILTER                          |                             |       |       |           |
|
```

```
|*  2 |   TABLE ACCESS BY INDEX ROWID     | MY_PATH_TABLE                |     1 |  3524 |    3   (0)|00:00:01 |
|*  3 |    INDEX RANGE SCAN               | SYS67616_PO_XMLINDE_PIKEY_IX |     1 |       |    2   (0)|00:00:01 |
|*  4 |  FILTER                           |                              |       |       |           |         |
|*  5 |   TABLE ACCESS BY INDEX ROWID     | MY_PATH_TABLE                |     1 |  3524 |    3   (0)|00:00:01 |
|*  6 |    INDEX RANGE SCAN               | SYS67616_PO_XMLINDE_PIKEY_IX |     1 |       |    2   (0)|00:00:01 |
|   7 |  NESTED LOOPS                     |                              |       |       |           |         |
|   8 |   NESTED LOOPS                    |                              |     1 |  1546 |   30   (4)|00:00:01 |
|   9 |    NESTED LOOPS                   |                              |     1 |    24 |   28   (4)|00:00:01 |
|  10 |     VIEW                          | VW_SQ_1                      |     1 |    12 |   26   (0)|00:00:01 |
|  11 |      HASH UNIQUE                  |                              |     1 |  5046 |           |         |
|  12 |       NESTED LOOPS                |                              |     1 |  5046 |   26   (0)|00:00:01 |
|* 13 |        TABLE ACCESS BY INDEX ROWID| MY_PATH_TABLE                |     1 |  3524 |   24   (0)|00:00:01 |
|* 14 |         INDEX RANGE SCAN          | SYS67616_PO_XMLINDE_VALUE_IX |    73 |       |    1   (0)|00:00:01 |
|* 15 |        TABLE ACCESS BY INDEX ROWID| MY_PATH_TABLE                |     1 |  1522 |    2   (0)|00:00:01 |
|* 16 |         INDEX RANGE SCAN          | SYS67616_PO_XMLINDE_PIKEY_IX |     1 |       |    1   (0)|00:00:01 |
|  17 |     TABLE ACCESS BY USER ROWID    | PO_CLOB                      |     1 |    12 |    1   (0)|00:00:01 |
|* 18 |    INDEX RANGE SCAN               | SYS67616_PO_XMLINDE_PIKEY_IX |     1 |       |    1   (0)|00:00:01 |
|* 19 |   TABLE ACCESS BY INDEX ROWID     | MY_PATH_TABLE                |     1 |  1522 |    2   (0)|00:00:01 |
-------------------------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter(:B1<SYS_ORDERKEY_MAXCHILD(:B2))
   2 - filter(SYS_XMLI_LOC_ISNODE("SYS_P2"."LOCATOR")=1)
   3 - access("SYS_P2"."RID"=:B1 AND "SYS_P2"."PATHID"=HEXTORAW('28EC')  AND "SYS_P2"."ORDER_KEY">:B2 AND
              "SYS_P2"."ORDER_KEY"<SYS_ORDERKEY_MAXCHILD(:B3))
       filter(SYS_ORDERKEY_DEPTH("SYS_P2"."ORDER_KEY")=SYS_ORDERKEY_DEPTH(:B1)+1)
   4 - filter(:B1<SYS_ORDERKEY_MAXCHILD(:B2))
   5 - filter(SYS_XMLI_LOC_ISNODE("SYS_P5"."LOCATOR")=1)
   6 - access("SYS_P5"."RID"=:B1 AND "SYS_P5"."PATHID"=HEXTORAW('60E0')  AND "SYS_P5"."ORDER_KEY">:B2 AND
              "SYS_P5"."ORDER_KEY"<SYS_ORDERKEY_MAXCHILD(:B3))
       filter(SYS_ORDERKEY_DEPTH("SYS_P5"."ORDER_KEY")=SYS_ORDERKEY_DEPTH(:B1)+1)
  13 - filter("SYS_P10"."VALUE"='SBELL-2002100912333601PDT' AND "SYS_P10"."PATHID"=HEXTORAW('4F8C')  AND
              SYS_XMLI_LOC_ISNODE("SYS_P10"."LOCATOR")=1)
  14 - access(SUBSTRB("VALUE",1,1599)='SBELL-2002100912333601PDT')
  15 - filter(SYS_XMLI_LOC_ISNODE("SYS_P8"."LOCATOR")=1)
  16 - access("SYS_P10"."RID"="SYS_P8"."RID" AND "SYS_P8"."PATHID"=HEXTORAW('4E36')  AND
              "SYS_P8"."ORDER_KEY"<"SYS_P10"."ORDER_KEY")
       filter("SYS_P10"."ORDER_KEY"<SYS_ORDERKEY_MAXCHILD("SYS_P8"."ORDER_KEY") AND
              SYS_ORDERKEY_DEPTH("SYS_P8"."ORDER_KEY")+1=SYS_ORDERKEY_DEPTH("SYS_P10"."ORDER_KEY"))
  18 - access("PO_CLOB".ROWID="SYS_ALIAS_4"."RID" AND "SYS_ALIAS_4"."PATHID"=HEXTORAW('3748') )
  19 - filter(SYS_XMLI_LOC_ISNODE("SYS_ALIAS_4"."LOCATOR")=1)

Note
-----
   - dynamic sampling used for this statement (level=2)
```

The execution plan for the query in Example 6–29 shows, by referring to the path table, that `XMLIndex` is used. It also shows the use of Oracle internal SQL function `sys_orderkey_depth`—see "Guidelines for Using XMLIndex with an Unstructured Component" on page 6-32.

Example 6–30 shows an execution plan that indicates that the `XMLIndex` index created in Example 6–22 is picked up for a query that uses two `WHERE` clause predicates. With only the unstructured `XMLIndex` component, this query would involve a join of the path table to itself, because of the two different paths in the `WHERE` clause.

Example 6–30 Using a Structured XMLIndex Component for a Query with Two Predicates

```
EXPLAIN PLAN FOR
  SELECT XMLQuery('/PurchaseOrder/LineItems/LineItem' PASSING OBJECT_VALUE RETURNING CONTENT)
    FROM po_clob
    WHERE XMLExists('/PurchaseOrder/LineItems/LineItem
                     [ora:contains(Description, "Picnic") > 0]' PASSING OBJECT_VALUE)
```

```
      AND XMLEXists('/PurchaseOrder[User="SBELL"]' PASSING OBJECT_VALUE);

-----------------------------------------------------------------------------------------------------------
| Id  | Operation                    | Name                          | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                               |     1 |   189 |    22  (14)| 00:00:01 |
|   1 |  SORT GROUP BY               |                               |     1 |  3524 |            |          |
|*  2 |   TABLE ACCESS BY INDEX ROWID| PATH_TAB                      |     2 |  7048 |     3   (0)| 00:00:01 |
|*  3 |    INDEX RANGE SCAN          | SYS67840_PO_XMLINDE_PIKEY_IX  |     1 |       |     2   (0)| 00:00:01 |
|*  4 |  HASH JOIN SEMI              |                               |     1 |   189 |    22  (14)| 00:00:01 |
|   5 |   NESTED LOOPS               |                               |    13 |   637 |     4  (25)| 00:00:01 |
|   6 |    SORT UNIQUE               |                               |    13 |   351 |     3   (0)| 00:00:01 |
|*  7 |     TABLE ACCESS FULL        | PO_IDX_TAB                    |    13 |   351 |     3   (0)| 00:00:01 |
|*  8 |    INDEX UNIQUE SCAN         | SYS_C006004                   |     1 |    22 |     0   (0)| 00:00:01 |
|*  9 |   TABLE ACCESS FULL          | PO_INDEX_LINEITEM             |    13 |  1820 |    17   (6)| 00:00:01 |
-----------------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - filter(SYS_XMLI_LOC_ISNODE("SYS_P0"."LOCATOR")=1)
   3 - access("SYS_P0"."RID"=:B1 AND "SYS_P0"."PATHID"=HEXTORAW('3748') )
   4 - access("SYS_ALIAS_1"."SYS_NC_OID$"="SYS_ALIAS_3"."OID")
   7 - filter("SYS_ALIAS_2"."USERNAME"='SBELL')
   8 - access("SYS_ALIAS_1"."SYS_NC_OID$"="SYS_ALIAS_2"."OID")
   9 - filter(SYS_XMLCONTAINS("SYS_ALIAS_3"."DESCRIPTION",'Picnic')>0)

Note
-----
   - dynamic sampling used for this statement (level=2)
   - Unoptimized XML construct detected (enable XMLOptimizationCheck for more information)

30 rows selected.
```

The presence in Example 6–30 of the path table name, `path_tab`, indicates that the unstructured index component of the index is used. The presence of the index content table `po_idx_tab` indicates that the structured index component is used.

Example 6–31 shows an execution plan that indicates that the same `XMLIndex` index is also picked up for a query that uses multilevel `XMLTable` chaining. With only the unstructured `XMLIndex` component, this query too would involve a join of the path table to itself, because of the different paths in the two `XMLTable` function calls.

Example 6–31 Using a Structured XMLIndex Component for a Query with Multilevel Chaining

```
EXPLAIN PLAN FOR
  SELECT po.reference, li.*
    FROM po_clob p,
         XMLTable('/PurchaseOrder' PASSING p.OBJECT_VALUE
                  COLUMNS reference   VARCHAR2(30)  PATH 'Reference',
                          lineitem    XMLType       PATH 'LineItems/LineItem') po,
         XMLTable('/LineItem' PASSING po.lineitem
                  COLUMNS itemno      BINARY_DOUBLE PATH '@ItemNumber',
                          description VARCHAR2(256) PATH 'Description',
                          partno      VARCHAR2(14)  PATH 'Part/@Id',
                          quantity    BINARY_DOUBLE PATH 'Part/@Quantity',
                          unitprice   BINARY_DOUBLE PATH 'Part/@UnitPrice') li
    WHERE po.reference = 'SBELL-20021009123335280PDT';

-------------------------------------------------------------------------------------------------
| Id  | Operation                    | Name                    | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                         |    17 | 20366 |    11   (0)| 00:00:01 |
|   1 |  NESTED LOOPS                |                         |       |       |            |          |
```

```
|   2 |   NESTED LOOPS               |                        |    17 | 20366 |    11   (0)| 00:00:01 |
|   3 |    NESTED LOOPS              |                        |     1 |   539 |     3   (0)| 00:00:01 |
|*  4 |     TABLE ACCESS FULL        | PO_IDX_TAB             |     1 |   529 |     3   (0)| 00:00:01 |
|*  5 |     INDEX UNIQUE SCAN        | SYS_C006320            |     1 |    10 |     0   (0)| 00:00:01 |
|*  6 |    INDEX RANGE SCAN          | SYS69412_69421_PKY_IDX |    17 |       |     1   (0)| 00:00:01 |
|   7 |   TABLE ACCESS BY INDEX ROWID| PO_INDEX_LINEITEM      |    17 | 11203 |     8   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   4 - filter("SYS_ALIAS_8"."REFERENCE"='SBELL-20021009123335280PDT')
   5 - access("SYS_ALIAS_8"."OID"="P"."SYS_NC_OID$")
   6 - access("SYS_ALIAS_8"."KEY"="SYS_ALIAS_9"."PKEY")

Note
-----
   - dynamic sampling used for this statement

25 rows selected.
```

The execution plan shows direct access to the relational index content tables, `po_idx_tab` and `po_index_lineitem`. There is *no* access at all to the path table, `path_tab`.

> **See Also:** "Collecting Statistics on XMLIndex Objects for the Cost-Based Optimizer" on page 6-39

Turning Off Use of XMLIndex

You can turn off the use of `XMLIndex` in any of these ways:

- Use optimizer hint `/*+ NO_XML_QUERY_REWRITE */`
- Use optimizer hint `/*+ NO_XMLINDEX_REWRITE */`

Hints `NO_XML_QUERY_REWRITE` and `NO_XMLINDEX_REWRITE` turn off the use of *all* `XMLIndex` indexes. In addition to turning off use of `XMLIndex`, `NO_XML_QUERY_REWRITE` turns off all XQuery optimization (`XMLIndex` is part of XPath rewrite).

Example 6–32 shows the use of these optimizer hints.

Example 6–32 Turning Off XMLIndex using Optimizer Hints

```
SELECT /*+ NO_XMLINDEX_REWRITE */
  count(*) FROM po_clob WHERE XMLExists('$p/*' PASSING OBJECT_VALUE AS "p");

SELECT /*+ NO_XML_QUERY_REWRITE */
  count(*) FROM po_clob WHERE XMLExists('$p/*' PASSING OBJECT_VALUE AS "p");
```

> **Note:** The `NO_INDEX` optimizer hint does not apply to `XMLIndex`.

> **See Also:**
>
> - "XQuery Optional Features" on page 5-43 for information about XQuery pragmas `ora:xq_proc` and `ora:xq_qry`, which you can use for fine-grained control of XQuery optimization
> - "How Oracle XML DB Processes XMLType Methods and SQL Functions" on page 3-58 for information about streaming evaluation of binary XML data

XMLIndex Path Subsetting: Specifying the Paths You Want to Index

One of the advantages of an XMLIndex index with an unstructured component is that it is very general: you need not specify which XPath locations to index; you need no prior knowledge of the XPath expressions that will be queried. By default, an unstructured XMLIndex indexes all possible XPath locations in your XML data.

However, if you are aware of the XPath expressions that you are most likely to query, then you can narrow the focus of XMLIndex indexing and thus improve performance. Having fewer indexed nodes means less space is required for indexing, which improves index maintenance during DML operations. Having fewer indexed nodes improves DDL performance, and having a smaller path table improves query performance.

You narrow the focus of indexing by pruning the set of XPath expressions (paths) corresponding to XML fragments to be indexed, specifying a subset of all possible paths. You can do this in two alternative ways:

- Exclusion – Start with the default behavior of including all possible XPath expressions, and exclude some of them from indexing.
- Inclusion – Start with an empty set of XPath expressions to be used in indexing, and add paths to this inclusion set.

You can specify path subsetting either when you create an XMLIndex index using CREATE INDEX or when you modify it using ALTER INDEX. In both cases, you provide the subsetting information in the PATHS parameter of the statement's PARAMETERS clause. For exclusion, you use keyword EXCLUDE. For inclusion, you use keyword INCLUDE for ALTER INDEX and no keyword for CREATE INDEX (list the paths to include). You can also specify namespace mappings for the nodes targeted by the PATHS parameter.

For ALTER INDEX, keyword INCLUDE or EXCLUDE is followed by keyword ADD or REMOVE, to indicate whether the list of paths that follows the keyword is to be added or removed from the inclusion or exclusion list. For example, this statement adds path /PurchaseOrder/Reference to the list of paths to be excluded from indexing:

```
ALTER INDEX po_xmlindex_ix REBUILD
  PARAMETERS ('PATHS (EXCLUDE ADD (/PurchaseOrder/Reference))');
```

To alter an XMLIndex index so that it *includes all* possible paths, use keyword INDEX_ALL_PATHS. See "alter_index_paths_clause ::=" on page 6-42.

Note: If you create an XMLIndex index that has both structured and unstructured components, then, by default, any nodes indexed in the structured component are also indexed in the unstructured component; that is, they are *not* automatically *excluded* from the unstructured component. If you do not want unstructured XMLIndex indexing to apply to them, then you must explicitly use path subsetting to exclude them.

See Also: "PARAMETERS Clause for CREATE INDEX and ALTER INDEX" on page 6-41 for the syntax of the PARAMETERS clause

Examples of XMLIndex Path Subsetting

This section presents some examples of defining XMLIndex indexes on subsets of XPath expressions.

Example 6–33 XMLIndex Path Subsetting with CREATE INDEX

```
CREATE INDEX po_xmlindex_ix ON po_clob (OBJECT_VALUE) INDEXTYPE IS XDB.XMLINDEX
  PARAMETERS ('PATHS (INCLUDE (/PurchaseOrder/LineItems//*
                              /PurchaseOrder/Reference))');
```

This statement creates an index that indexes only top-level element `PurchaseOrder` and some of its children, as follows:

- All `LineItems` elements and their descendants
- All `Reference` elements

It does that by including the specified paths, starting with an empty set of paths to be used for the index.

Example 6–34 XMLIndex Path Subsetting with ALTER INDEX

```
ALTER INDEX po_xmlindex_ix REBUILD
  PARAMETERS ('PATHS (INCLUDE ADD (/PurchaseOrder/Requestor
                                  /PurchaseOrder/Actions/Action//*))');
```

This statement adds two more paths to those used for indexing. These paths index element `Requestor` and descendants of element `Action` (and their ancestors).

Example 6–35 XMLIndex Path Subsetting using a Namespace Prefix

If an XPath expression to be used for `XMLIndex` indexing uses namespace prefixes, you can use a `NAMESPACE MAPPING` clause to the `PATHS` list, to specify those prefixes. Here is an example:

```
CREATE INDEX po_xmlindex_ix ON po_clob (OBJECT_VALUE) INDEXTYPE IS XDB.XMLIndex
  PARAMETERS ('PATHS (INCLUDE (/PurchaseOrder/LineItems//*   /PurchaseOrder/ipo:Reference)
                   NAMESPACE MAPPING (xmlns="http://xmlns.oracle.com"
                                      xmlns:ipo="http://xmlns.oracle.com/ipo"))');
```

XMLIndex Path-Subsetting Rules

The following rules apply to `XMLIndex` path subsetting:

- The paths must reference only child and descendant axes, and they must test only element and attribute nodes or their names (possibly using wildcards). In particular, the paths must not involve predicates.
- You cannot specify both path exclusion and path inclusion; choose one or the other.
- If an index was created using path exclusion (inclusion), then you can modify it using only path exclusion (inclusion)—index modification must either further restrict or further extend the path subset. For example, you cannot create an index that includes certain paths and subsequently modify it to exclude certain paths.

Guidelines for Using XMLIndex with an Unstructured Component

The following are some guidelines for using `XMLIndex` with an unstructured component. These guidelines are applicable only when the two alternatives discussed return the same result set.

- Avoid prefixing `//` with ancestor elements. For example, use `//c`, *not* **`/a/b`**`//c`, provided these return the same result set.
- Avoid prefixing `/*` with ancestor elements. For example, use `/*/*/*`, *not* **`/a`**`/*/*`, provided these return the same result set.

- In a `WHERE` clause, use `XMLExists` rather than `XMLCast` of `XMLQuery`. This can allow optimization that, in effect, invokes a subquery against the path-table `VALUE` column. For example, use this:

```
SELECT count(*) FROM purchaseorder p
  WHERE
    XMLExists('$p/PurchaseOrder/LineItems/LineItem/Part[@Id="715515011020"]'
                  PASSING OBJECT_VALUE AS "p");
```

  Do not use this:

```
SELECT count(*) FROM purchaseorder p
  WHERE XMLCast(XMLQuery('$p/PurchaseOrder/LineItems/LineItem/Part/@Id'
                         PASSING OBJECT_VALUE AS "p" RETURNING CONTENT)
                AS VARCHAR2(14))
        = "715515011020";
```

- When possible, use `count(*)`, *not* `count(XMLCast(XMLQuery(...))`, in a `SELECT` clause. For example, if you know that a `LineItem` element in a purchase-order document has only one `Description` child, use this:

```
SELECT count(*) FROM po_clob, XMLTable('//LineItem' PASSING OBJECT_VALUE);
```

  Do not use this:

```
SELECT count(li.value)
 FROM po_clob p, XMLTable('//LineItem' PASSING p.OBJECT_VALUE
                          COLUMNS value VARCHAR2(30) PATH 'Description') li;
```

- Reduce the number of XPath expressions used in a query `FROM` list as much as possible. For example, use this:

```
SELECT li.description
  FROM po_clob p,
       XMLTable('PurchaseOrder/LineItems/LineItem' PASSING p.OBJECT_VALUE
                COLUMNS description VARCHAR2(256) PATH 'Description') li;
```

  Do not use this:

```
SELECT li.description
  FROM po_clob p,
       XMLTable('PurchaseOrder/LineItems' PASSING p.OBJECT_VALUE) ls,
       XMLTable('LineItems/LineItem'      PASSING ls.OBJECT_VALUE
                COLUMNS description VARCHAR2(256) PATH 'Description') li;
```

- If you use an XPath expression in a query to drill down inside a virtual table (created, for example, using SQL/XML function `XMLTable`), then create a secondary index on the order key of the path table using Oracle SQL function `sys_orderkey_depth`. Here is an example of such a query; the selection navigates to element `Description` inside virtual line-item table `li`.

```
SELECT li.description
  FROM po_clob p,
       XMLTable('PurchaseOrder/LineItems/LineItem' PASSING p.OBJECT_VALUE
                COLUMNS description VARCHAR2(256) PATH 'Description') li;
```

  Such queries are evaluated using function `sys_orderkey_depth`, which returns the depth of the order-key value. Because the order index uses two columns, the index needed is a *composite* index over columns `ORDER_KEY` and `RID`, as well as over function `sys_orderkey_depth` applied to the `ORDER_KEY` value. For example:

```
CREATE INDEX depth_ix ON my_path_table
  (RID, sys_orderkey_depth(ORDER_KEY), ORDER_KEY);
```

See Also: Example 6–29 on page 6-27 for an example that shows the use of `sys_orderkey_depth`

Guidelines for Using XMLIndex with a Structured Component

The following are some guidelines for using `XMLIndex` with a structured component.

- Use `XMLIndex` with a structured component to project and index XML data as relational columns. Do not use function-based indexes; they are deprecated for use with XML. See "Function-Based Indexes" on page 6-5.
- Ensure data type correspondence between a query and an `XMLIndex` index that has a structured component. See "Data Type Considerations for XMLIndex Structured Component" on page 6-11.
- If you create a relational view over `XMLType` data (for example, using SQL function `XMLTable`), then you consider also creating an `XMLIndex` index with a structured component that targets the same relational columns. See Chapter 19, "XMLType Views".
- Instead of using a single XQuery expression for both fragment extraction and value filtering (search), use SQL/XML function `XMLQuery` in the `SELECT` clause to extract fragments and `XMLExists` in the `WHERE` clause to filter values.

  This lets Oracle XML DB evaluate fragment extraction functionally or by using streaming evaluation. For value filtering, this lets Oracle XML DB pick up an `XMLIndex` index that has a relevant structured component.
- To order query results, use a SQL `ORDER BY` clause, together with SQL/XML function `XMLTable`. Avoid using the XQuery `order by` clause. This is particularly pertinent if you use an `XMLIndex` index with a structured component.

Using XMLIndex on Oracle XML DB Repository

A database administrator (DBA) can create an `XMLIndex` index on resources in Oracle XML DB Repository to improve querying of XML data or metadata (system-defined or user-defined). An `XMLIndex` index on the repository contains only an unstructured component, not a structured component.

Creating an XMLIndex Index on Repository Resources

Only user `SYS` can create an `XMLIndex` index on Oracle XML DB Repository. After creating such an index, restrict it to those resources that will actually be queried.

You cannot index resources that contain a `REF` (in their contents or metadata) to a row in a view or to a row in a table that is not hierarchy-enabled. An attempt to do so raises an error.

Follow this procedure as a database administrator to create an `XMLIndex` index on specific repository resources:

1. Create the `XMLIndex` index on the repository. This excludes repository paths `/Resource/ACL`, `/Resource/RefCount`, and `/Resource/RCList` from the index created. It creates the path table and all secondary indexes, but it does not index any resources.

   ```
   CALL DBMS_XDB_ADMIN.CreateRepositoryXMLIndex();
   ```

No resources are indexed at this point. They are indexed after step 2.

2. Specify the repository resources to index, by repository path. You use PL/SQL procedure `XMLIndexAddPath` in package `DBMS_XDB_ADMIN` to do this.

   ```
   DBMS_XDB_ADMIN.XMLIndexAddPath(<path to index>);
   ```

   The parameter to `XMLIndexAddPath` is a repository path expression that targets a resource (file or folder) to index. (Note: A repository path expression is not an XPath expression.)

To drop an `XMLIndex` index created on the repository, do this:

```
CALL DBMS_XDB_ADMIN.DropRepositoryXMLIndex();
```

See Also:

- Chapter 21, "Accessing Oracle XML DB Repository Data" for information on repository resources and resource paths
- *Oracle Database PL/SQL Packages and Types Reference* for information on the procedures in PL/SQL package `DBMS_XDB_ADMIN`

Removing Repository Resources from Indexing with XMLIndex

Follow this procedure as a database administrator to remove a resource from indexing:

1. Repeat this procedure for each ancestor of the resource, to first remove them from indexing. You cannot remove a resource from indexing until you have first removed all of its ancestors from indexing.
2. Specify the repository resource to be removed from indexing, by repository path. You use PL/SQL procedure `XMLIndexRemovePath` in package `DBMS_XDB_ADMIN` to do this.

   ```
   DBMS_XDB_ADMIN.XMLIndexRemovePath(<path to index>, <recursively?>);
   ```

   The first parameter to `XMLIndexRemovePath` is a repository path expression that targets a resource to remove from indexing. The second parameter is a Boolean value that, if `TRUE`, removes the targeted resource and all of its descendants from indexing; if `FALSE`, only the targeted resource is removed from indexing.

   See Also: *Oracle Database PL/SQL Packages and Types Reference* for information on PL/SQL procedure `DBMS_XDB_ADMIN.XMLIndexREmovePath`

Querying Repository Data and Metadata Indexed with XMLIndex

For an `XMLIndex` on repository data or metadata to be used when you query the repository, *all* of the targeted resources must be indexed. Use Oracle SQL functions `under_path` and `equals_path` to target resources by repository path when you query. If you do not use either of these functions, then `under_path('/')` is implied.

Example 6–36 presents a query of resource data, `PurchaseOrder` elements whose `Reference` element has value `TFOX-20021009123335520PDT`.

Example 6–36 Using XMLIndex When Querying Resource Data

```
SELECT ANY_PATH FROM RESOURCE_VIEW
  WHERE XMLExists('declare namespace r = "http://xmlns.oracle.com/xdb/XDBResource.xsd"; (: :)
                   /r:Resource/r:Contents/PurchaseOrder[Reference="TFOX-20021009123335520PDT"]'
                  PASSING RES)
```

```
    AND under_path (RES, '/home/OE/PurchaseOrders/2002/') = 1;

ANY_PATH
------------------------------------------------------------
/home/OE/PurchaseOrders/2002/Jan/TFOX-20021009123335520PDT.xml

1 row selected.
```

The `under_path` expression here limits the paths queried so that they include only data indexed using `XMLIndex`. The `XMLExists` expression targets XML nodes using an XPath expression.

You can query repository metadata similarly. To do that, simply target resource elements that represent metadata, either system-defined or user-defined. Example 6–37 shows a query of the `CreationDate` element in system metadata.

Example 6–37 Using XMLIndex When Querying Resource Metadata

```
SELECT ANY_PATH FROM RESOURCE_VIEW
  WHERE XMLExists('declare namespace r = "http://xmlns.oracle.com/xdb/XDBResource.xsd"; (: :)
                   /r:Resource/r:CreationDate/text()=xs:dateTime("2005-02-07T18:31:53.093179")'
                  PASSING RES)
   AND under_path (RES, '/home/OE/PurchaseOrders/2002/') = 1;

ANY_PATH
-----------------------------------------------------------------
/home/OE/PurchaseOrders/2002/Apr
/home/OE/PurchaseOrders/2002/Apr/AMCEWEN-20021009123336171PDT.xml
/home/OE/PurchaseOrders/2002/Apr/AMCEWEN-20021009123336271PDT.xml
/home/OE/PurchaseOrders/2002/Apr/EABEL-20021009123336251PDT.xml
/home/OE/PurchaseOrders/2002/Apr/PTUCKER-20021009123336191PDT.xml
/home/OE/PurchaseOrders/2002/Apr/PTUCKER-20021009123336291PDT.xml
. . .

144 rows selected.
```

See Also: "Querying Metadata and the Folder Hierarchy" on page 3-81

Dropping an XMLIndex Index Created on Repository Resources

Only user `SYS` can drop an `XMLIndex` index created on Oracle XML DB Repository. To drop an `XMLIndex` index created on the repository, do this:

```
CALL DBMS_XDB_ADMIN.DropRepositoryXMLIndex();
```

See Also: *Oracle Database PL/SQL Packages and Types Reference* for information on PL/SQL procedure `DBMS_XDB_ADMIN.DropRepositoryXMLIndex`

XMLIndex Partitioning and Parallelism

If you partition an `XMLType` table, or a table with an `XMLType` column using range or list partitioning, you can also create an `XMLIndex` index on the table. If you use the keyword `LOCAL` when you create the `XMLIndex` index, then the index and all of its storage tables are locally equipartitioned with respect to the base table.

If you do not use the keyword `LOCAL`, then you cannot create an `XMLIndex` index on a partitioned table. Also, if you hash-partition a table, then you cannot create an `XMLIndex` index on it.

You can use a PARALLEL clause (with optional degree) when creating or altering an XMLIndex index to ensure that index creation and maintenance are carried out in parallel. If the base table is partitioned or enabled for parallelism, then this can improve the performance for both DML operations (INSERT, UPDATE, DELETE) and index DDL operations (CREATE, ALTER, REBUILD).

Specifying parallelism for an index can also consume more storage, because storage parameters apply separately to each query server process. For example, an index created with an INITIAL value of 5M and a parallelism degree of 12 consumes at least 60M of storage during index creation.

The syntax for the parallelism clause for CREATE INDEX and ALTER INDEX is the same as for other domain indexes:

```
{ NOPARALLEL | PARALLEL [ integer ] }
```

Example 6–38 creates an XMLIndex index with a parallelism degree of 10. If the base table is partitioned, then this index is equipartitioned.

Example 6–38 Creating an XMLIndex Index in Parallel

```
CREATE INDEX po_xmlindex_ix ON sale_info (sale_po_clob) INDEXTYPE IS XDB.XMLIndex
  LOCAL PARALLEL 10;
```

In Example 6–38, the path table and the secondary indexes are created with the same parallelism degree as the XMLIndex index itself, 10, by inheritance. You can specify different parallelism degrees for these by using separate PARALLEL clauses. Example 6–39 demonstrates this. Again, because of keyword LOCAL, if the base table is partitioned, then this index is equipartitioned.

Example 6–39 Using Different PARALLEL Degrees for XMLIndex Internal Objects

```
CREATE INDEX po_xmlindex_ix ON sale_info (sale_po_clob) INDEXTYPE IS XDB.XMLIndex
  LOCAL PARAMETERS ('PATH TABLE po_path_table (PARALLEL 10)
                     PIKEY INDEX po_pikey_ix
                     VALUE INDEX po_value_ix (PARALLEL 5)') NOPARALLEL;
```

In Example 6–39, the XMLIndex index itself is created serially, because of NOPARALLEL. The secondary index po_pikey_ix is also populated serially, because no parallelism is specified explicitly for it; it inherits the parallelism of the XMLIndex index. The path table itself is created with a parallelism degree of 10, and the secondary index value column, po_value_ix, is populated with a degree of 5, due to their explicit parallelism specifications.

Any parallelism you specify for an XMLIndex index, its path table, or its secondary indexes is exploited during subsequent DML operations and queries.

Note that there are two places where you can specify parallelism for XMLIndex: within the PARAMETERS clause parenthetical expression and after it.

See Also:

- *Oracle Database SQL Language Reference* for information on the CREATE INDEX parallel clause
- "PARAMETERS Clause for CREATE INDEX and ALTER INDEX" on page 6-41 for the syntax of the PARAMETERS clause
- "Structured and Unstructured XMLIndex Components" on page 6-9

Asynchronous (Deferred) Maintenance of XMLIndex Indexes

This feature applies to an XMLIndex index that has *only* an *unstructured* component. If you specify asynchronous maintenance for an XMLIndex index that has a *structured* component, then an error is raised.

By default, XMLIndex indexing is updated (maintained) at each DML operation, so that it remains in sync with the base table. In some situations, you might not require this, and using possibly stale indexes might be acceptable. In that use case, you can decide to defer the cost of index maintenance, performing at commit time only or at some time when database load is reduced. This can improve DML performance. It can also improve index maintenance performance by enabling bulk loading of unsynchronized index rows when an index is synchronized.

Using a stale index has no effect, other than performance, on DML operations. It can have an effect on query results, however: If the index is not up-to-date at query time, then the query results might not be up-to-date either. Even if only one column of a base table is of data type XMLType, all queries on that table reflect the database data as of the last synchronization of the XMLIndex index on the XMLType column.

You can specify index maintenance deferment using the parameters clause of a CREATE INDEX or ALTER INDEX statement.

Be aware that even if you defer synchronization for an XMLIndex index, the following database operations automatically synchronize the index:

- Any DDL operation on the index – ALTER INDEX or creation of secondary indexes
- Any DDL operation on the base table – ALTER TABLE or creation of another index

Table 6–7 lists the synchronization options and the ASYNC clause syntax you use to specify them. The ASYNC clause is used in the PARAMETERS clause of a CREATE INDEX or ALTER INDEX statement for XMLIndex.

Table 6–7 Index Synchronization

When to Synchronize	ASYNC Clause Syntax
Always	ASYNC (SYNC ALWAYS) This is the default behavior. You can specify it explicitly, to cancel a previous ASYNC specification.
Upon commit	ASYNC (SYNC ON COMMIT)
Periodically	ASYNC (SYNC EVERY "*repeat_interval*") *repeat_interval* is the same as for the calendaring syntax of DBMS_SCHEDULER To use EVERY, you must have the CREATE JOB privilege.
Manually, on demand	ASYNC (SYNC MANUAL) You can manually synchronize the index using PL/SQL procedure DBMS_XMLINDEX.SyncIndex.

Optional ASYNC syntax parameter STALE is intended for possible future use; you need never specify it explicitly. It has value FALSE whenever ALWAYS is used; otherwise it has value TRUE. Specifying an explicit STALE value that contradicts this rule raises an error.

Example 6–40 creates an XMLIndex index that is synchronized every Monday at 3:00 pm, starting tomorrow.

Example 6–40 Specifying Deferred Synchronization for XMLIndex

```
CREATE INDEX po_xmlindex_ix ON po_clob (OBJECT_VALUE) INDEXTYPE IS XDB.XMLIndex
  PARAMETERS ('ASYNC (SYNC EVERY "FREQ=HOURLY; INTERVAL = 1")');
```

Example 6–41 manually synchronizes the index created in Example 6–40.

Example 6–41 Manually Synchronizing an XMLIndex Index using SYNCINDEX

```
EXEC DBMS_XMLINDEX.SyncIndex('OE', 'PO_XMLINDEX_IX', REINDEX => TRUE);
```

When `XMLIndex` index synchronization is deferred, all DML changes (inserts, updates, and deletions) made to the base table since the last index synchronization are recorded in a pending table, one row per DML operation. The name of this table is the value of column `PEND_TABLE_NAME` of static public views `USER_XML_INDEXES`, `ALL_XML_INDEXES`, and `DBA_XML_INDEXES`.

You can examine this table to determine when synchronization might be appropriate for a given `XMLIndex` index. The more rows there are in the pending table, the more the index is likely to be in need of synchronization.

If the pending table is large, then setting parameter `REINDEX` to `TRUE` when calling `SyncIndex`, as in Example 6–41, can improve performance. When `REINDEX` is `TRUE`, all of the secondary indexes are dropped and then re-created after the pending table data is bulk-loaded.

See Also:

- *Oracle Database PL/SQL Packages and Types Reference*, section "Calendaring Syntax", for the syntax of *`repeat_interval`*
- *Oracle Database PL/SQL Packages and Types Reference* for information on PL/SQL procedure `DBMS_XMLINDEX.SyncIndex`

Collecting Statistics on XMLIndex Objects for the Cost-Based Optimizer

The Oracle Database cost-based optimizer determines how to most cost-effectively evaluate a given query, including which indexes, if any, to use. For it to be able to do this accurately, you must collect statistics on various database objects.

For `XMLIndex`, you normally need to collect statistics on only the base table on which the `XMLIndex` index is defined (using, for example, procedure `DBMS_STATS.gather_table_stats`). This automatically collects statistics for the `XMLIndex` index itself, as well as the path table, its secondary indexes, and any structured component content tables and their secondary indexes.

If you delete statistics on the base table (using procedure `DBMS_STATS.delete_table_stats`), then statistics on the other objects are also deleted. Similarly, if you collect statistics on the `XMLIndex` index (using procedure `DBMS_STATS.gather_index_stats`), then statistics are also collected on the path table, its secondary indexes, and any structured component content tables and their secondary indexes.

Example 6–42 collects statistics on the base table `po_clob`. Statistics are automatically collected on the `XMLIndex` index, its path table, and the secondary path-table indexes.

Example 6–42 Automatic Collection of Statistics on XMLIndex Objects

```
CALL DBMS_STATS.gather_table_stats(USER, 'PO_CLOB', ESTIMATE_PERCENT => NULL);
```

See Also: "Data Dictionary Static Public Views Related to XMLIndex" on page 6-40 for information about database views that record statistics information for an XMLIndex index

Data Dictionary Static Public Views Related to XMLIndex

Information about the standard database indexes is available in static public views USER_INDEXES, ALL_INDEXES, and DBA_INDEXES. Similar information about XMLIndex indexes is available in static public views USER_XML_INDEXES, ALL_XML_INDEXES, and DBA_XML_INDEXES.

Table 6–8 describes the columns in each of these views.

Table 6–8 XMLIndex Static Public Views

Column Name	Type	Description
ASYNC	VARCHAR2	Asynchronous index updating specification. See "Asynchronous (Deferred) Maintenance of XMLIndex Indexes" on page 6-38.
EX_OR_INCLUDE	VARCHAR2	Path subsetting: ▪ FULLY_IX – The index uses no path subsetting. ▪ EXCLUDE – The index uses only exclusion subsetting. ▪ INCLUDE – The index uses only inclusion subsetting.
INDEX_NAME	VARCHAR2	Name of the XMLIndex index.
INDEX_OWNER	VARCHAR2	Owner of the index. Not available for USER_XML_INDEXES.
INDEX_TYPE	VARCHAR2	The types of components the index is composed of: STRUCTURED, UNSTRUCTURED, or STRUCTURED AND UNSTRUCTURED.
PARAMETERS	XMLType	Information from the PARAMETERS clause that was used to create the index. If an unstructured XMLIndex component is present, the PARAMETERS clause can include the set of XPath paths defining path-subsetting and the name of a scheduler job for synchronization. If a structured component is present, the PARAMETERS clause includes the name of the structure group and the table definitions provided by XMLTable, including the XQuery expressions that define the columns.
PATH_TABLE_NAME	VARCHAR2	Name of the XMLIndex path table.
PEND_TABLE_NAME	VARCHAR2	Name of the table that records base-table DML operations since the last index synchronization. See "Asynchronous (Deferred) Maintenance of XMLIndex Indexes" on page 6-38.
TABLE_NAME	VARCHAR2	Name of the base table on which the index is defined.
TABLE_OWNER	VARCHAR2	Owner of the base table on which the index is defined.

These views provide information about an XMLIndex index, but there is no single catalog view that provides information about the statistics gathered for an XMLIndex index. This statistics information is distributed among the following views:

- USER_INDEXES, ALL_INDEXES, DBA_INDEXES – Column LAST_ANALYZED provides the date when the XMLIndex index was last analyzed.

- USER_TAB_STATISTICS, ALL_TAB_STATISTICS, DBA_TAB_STATISTICS – Column TABLE_NAME provides information about the structured and unstructured components of an XMLIndex index. For information about the structured or unstructured component, query using the name of the path table or the XMLTable table as TABLE_NAME, respectively.
- USER_IND_STATISTICS, ALL_IND_STATISTICS, DBA_IND_STATISTICS – Column INDEX_NAME provides information about each of the secondary indexes for an XMLIndex index. for information about a given secondary index, query using the name of that secondary index as INDEX_NAME.

PARAMETERS Clause for CREATE INDEX and ALTER INDEX

This section describes the usage and syntax of the PARAMETERS clause for SQL statements CREATE INDEX and ALTER INDEX when used with XMLIndex.

See Also:

- *Oracle Database SQL Language Reference* for the syntax of *index_attributes*
- *Oracle Database SQL Language Reference* for the syntax of *segment_attributes_clause*
- *Oracle Database SQL Language Reference* for the syntax of *table_properties*
- *Oracle Database SQL Language Reference* for the syntax of *parallel_clause*
- *Oracle Database SQL Language Reference* for additional information about the syntax and semantics of CREATE INDEX
- *Oracle Database SQL Language Reference* for additional information about the syntax and semantics of ALTER INDEX
- *Oracle Database PL/SQL Packages and Types Reference*, section "Calendaring Syntax", for the syntax of *repeat_interval*

Using a Registered PARAMETERS Clause for XMLIndex

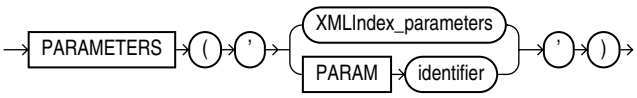
The string value used for the PARAMETERS clause of a CREATE INDEX or ALTER INDEX statement has a 1000-character limit. To get around this limitation, you can use PL/SQL procedures registerParameter and modifyParameter in package DBMS_XMLINDEX.

For each of these procedures, you provide a string of parameters (unlimited in length) and an identifier under which the string is registered. Then, in the index PARAMETERS clause, you provide the identifier preceded by the keyword PARAM, instead of a literal string.

The identifier must already have been registered before you can use it in a CREATE INDEX or ALTER INDEX statement.

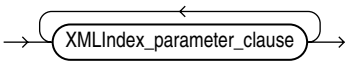
PARAMETERS Clause Syntax for CREATE INDEX and ALTER INDEX

XMLIndex_parameters_clause ::=



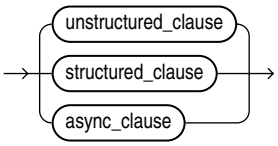
See Also: "Usage of XMLIndex_parameters_clause" on page 6-46

XMLIndex_parameters ::=

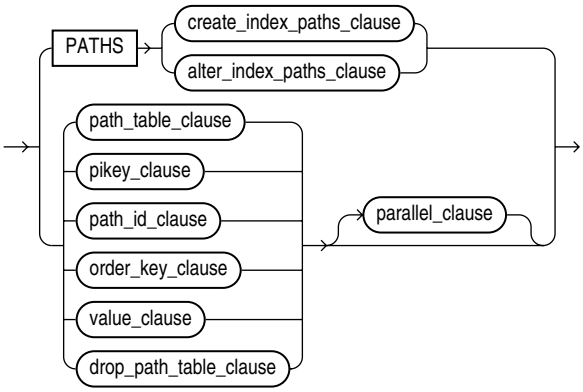


See Also: "Usage of XMLIndex_parameters" on page 6-46

XMLIndex_parameter_clause ::=

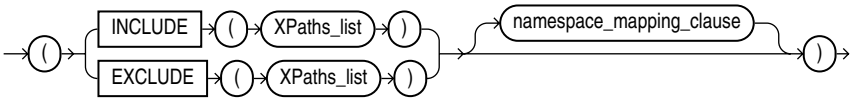


unstructured_clause ::=



****

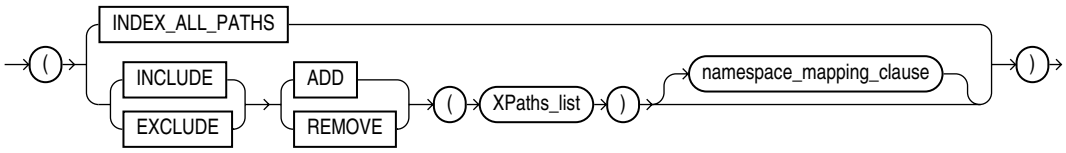
create_index_paths_clause ::=



See Also:

- "Usage of PATHS Clause" on page 6-46
- "Usage of create_index_paths_clause and alter_index_paths_clause" on page 6-46

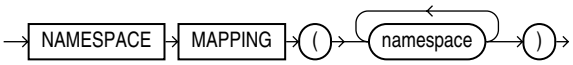
alter_index_paths_clause ::=



See Also:

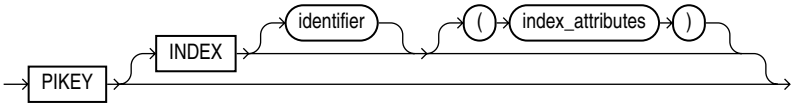
- "Usage of PATHS Clause" on page 6-46
- "Usage of create_index_paths_clause and alter_index_paths_clause" on page 6-46

namespace_mapping_clause ::=



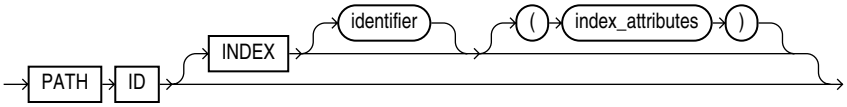
path_table_clause ::=

pikey_clause ::=



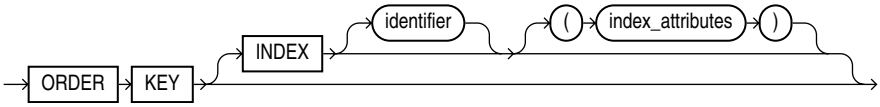
See Also: "Usage of pikey_clause, path_id_clause, and order_key_clause" on page 6-47

path_id_clause ::=



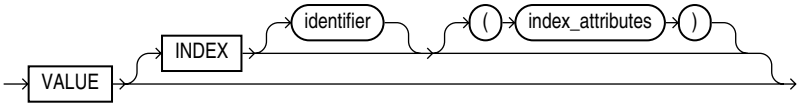
See Also: "Usage of pikey_clause, path_id_clause, and order_key_clause" on page 6-47

order_key_clause ::=



See Also: "Usage of pikey_clause, path_id_clause, and order_key_clause" on page 6-47

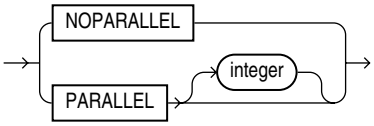
value_clause ::=



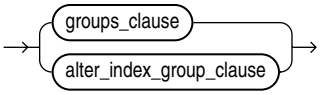
See Also: "Usage of value_clause" on page 6-47

drop_path_table_clause ::=

parallel_clause ::=

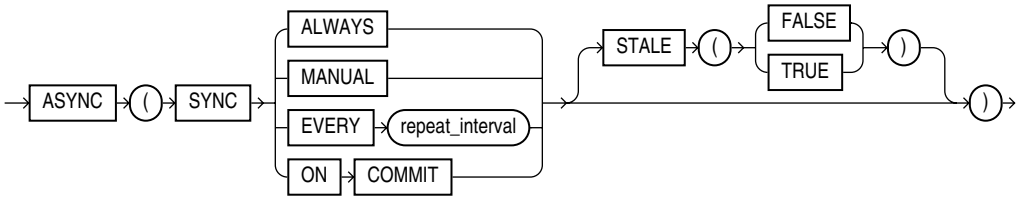


structured_clause ::=



See Also: "Usage of groups_clause and alter_index_group_clause" on page 6-48

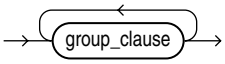
async_clause ::=



****

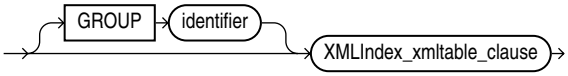
See Also: "Usage of async_clause" on page 6-47

groups_clause ::=



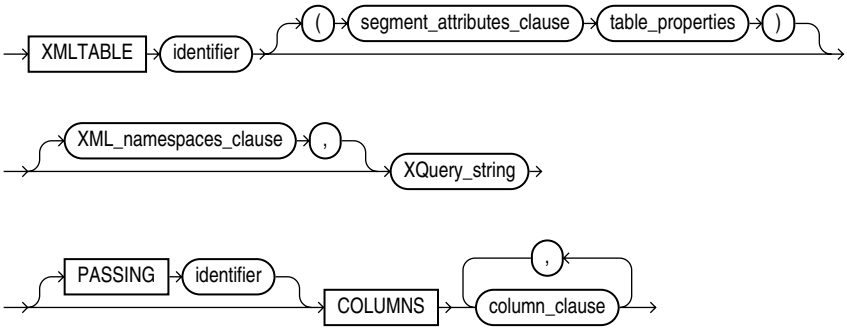
See Also: "Usage of groups_clause and alter_index_group_clause" on page 6-48

group_clause ::=



See Also: "Usage of groups_clause and alter_index_group_clause" on page 6-48

XMLIndex_xmltable_clause ::=

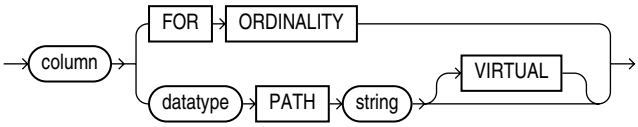


Syntax elements XML_namespaces_clause and XQuery_string are the same as for SQL/XML function `XMLTable`.

See Also:

- "Usage of XMLIndex_xmltable_clause" on page 6-48
- "XMLTABLE SQL/XML Function in Oracle XML DB" on page 5-7

column_clause ::=

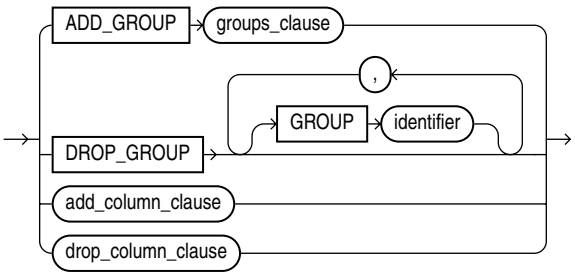


Syntax element column_clause is similar, but not identical, to XML_table_column in SQL/XML function `XMLTable`.

See Also:

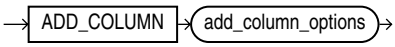
- "Usage of column_clause" on page 6-48
- "XMLTABLE SQL/XML Function in Oracle XML DB" on page 5-7

alter_index_group_clause ::=

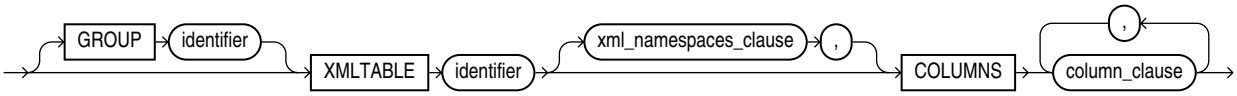


See Also: "Usage of groups_clause and alter_index_group_clause" on page 6-48

add_column_clause :==

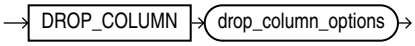


add_column_options :==

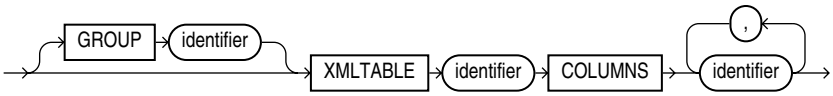


Syntax element XML_namespaces_clause is the same as for SQL/XML function `XMLTable`. See "XMLTABLE SQL/XML Function in Oracle XML DB" on page 5-7.

drop_column_clause :==



drop_column_options :==



Usage of XMLIndex_parameters_clause

When you create an `XMLIndex` index, if there is no *`XMLIndex_parameters_clause`*, then the new index has only an unstructured component. If there is an *`XMLIndex_parameters_clause`*, but the `PARAMETERS` argument is empty (`''`), then the result is the same: an index with only an unstructured component.

Usage of XMLIndex_parameters

The following considerations apply to using *`XMLIndex_parameters`*.

- There can be at most one *`XMLIndex_parameter_clause`* of each type in *`XMLIndex_parameters`*. For example, there can be at most one `PATHS` clause, at most one *`path_table_clause`*, and so on.
- If there is no *`structured_clause`* when you create an `XMLIndex` index, then the new index has only an unstructured component. If there is only a *`structured_clause`*, then the new index has only a structured component.

Usage of PATHS Clause

The following considerations apply to using the `PATHS` clause.

- There can be at most one `PATHS` clause in a `CREATE INDEX` statement. That is, there can be at most one occurrence of `PATHS` followed by *`create_index_paths_clause`*.
- Clause *`create_index_paths_clause`* is used only with `CREATE INDEX`; *`alter_index_paths_clause`* is used only with `ALTER INDEX`.

Usage of create_index_paths_clause and alter_index_paths_clause

The following considerations apply to using *`create_index_paths_clause`* and *`alter_index_paths_clause`*.

- The `INDEX_ALL_PATHS` keyword rebuilds the index to include all paths. This keyword is available only for *`alter_index_paths_clause`*, not *`create_index_paths_clause`*.
- An explicit list of paths to index can include wildcards and `//`.

- *XPaths_list* is a list of one or more XPath expressions, each of which includes only child axis, descendant axis, name test, and wildcard (*) constructs.
- If *XPaths_list* is omitted from *create_index_paths_clause*, all paths are indexed.
- For each unique namespace prefix that is used in an XPath expression in *XPaths_list*, a standard XML *namespace* declaration is needed, to provide the corresponding namespace information.
- You can change an index in ways that are not reflected directly in the syntax by dropping it and then creating it again as needed. For example, to change an index that was defined by including paths to one that is defined by excluding paths, drop it and then create it using EXCLUDE.

Usage of pikey_clause, path_id_clause, and order_key_clause

Syntactically, each of the clauses pikey_clause, path_id_clause, and order_key_clause is optional. A pikey index is created even if you do not specify a pikey_clause. To create a path id index or an order-key index, you must specify a path_id_clause or an order_key_clause, respectively.

Usage of value_clause

The following considerations apply to using *value_clause*.

- Column VALUE is created as VARCHAR2(4000).
- If clause *value_clause* consists only of the keyword VALUE, then the value index is created with the usual default attributes.
- If clause *path_id_clause* consists only of the keywords PATH ID, then the path-id index is created with the usual default attributes.
- If clause *order_key_clause* consists only of the keywords ORDER KEY, then the order-key index is created with the usual default attributes.

Usage of async_clause

The following considerations apply to using the ASYNC clause.

- Use this feature only with an XMLIndex index that has *only* an *unstructured* component. If you specify an ASYNC clause for an XMLIndex index that has a *structured* component, then an error is raised.
- ALWAYS means automatic synchronization occurs for each DML statement.
- MANUAL means no automatic synchronization occurs. You must manually synchronize the index using DBMS_XMLINDEX.SyncIndex.
- EVERY *repeat_interval* means automatically synchronize the index at interval *repeat_interval*. The syntax of *repeat_interval* is the same as that for PL/SQL package DBMS_SCHEDULER, and it must be enclosed in double-quotes (").To use EVERY you must have the CREATE JOB privilege.
- ON COMMIT means synchronize the index immediately after a commit operation. The commit does not return until the synchronization is complete. Since the synchronization is performed as a separate transaction, there can be a short period when the data is committed but index changes are not yet committed.
- STALE is optional. A value of TRUE means that query results might be stale; a value of FALSE means that query results are always up-to-date. The default value, and the only permitted explicitly specified value, is as follows.

- For ALWAYS, STALE is TRUE.
- For any other ASYNC option besides ALWAYS, STALE is FALSE.

Usage of groups_clause and alter_index_group_clause

Clause *groups_clause* is used only with CREATE INDEX; clause *alter_index_group_clause* is used only with ALTER INDEX.

Usage of XMLIndex_xmltable_clause

The following considerations apply to using *XMLIndex_xmltable_clause*.

- The *XQuery_string* expression in *XMLIndex_xmltable_clause* must not use the XQuery functions ora:view (deprecated), fn:doc, or fn:collection.
- Oracle XML DB raises an error if a given *XMLIndex_xmltable_clause* contains more than one *column_clause* of data type XMLType. To achieve the effect of defining two such virtual columns, you must instead add a separate *group_clause*.
- The PASSING clause in *XMLIndex_xmltable_clause* is optional. If not present, then an XMLType column is passed implicitly, as follows:
  - For the first *XMLIndex_xmltable_clause* in a parameters clause, the XMLType column being indexed is passed implicitly. (When indexing an XMLType table, pseudocolumn OBJECT_VALUE is passed.)
  - For each subsequent *XMLIndex_xmltable_clause*, the VIRTUAL XMLType column of the preceding *XMLIndex_xmltable_clause* is passed implicitly.

Usage of column_clause

When you use multilevel chaining of XMLTable in an XMLIndex index, the XMLTable table at one level corresponds to an XMLType column at the previous level. The syntax description shows keyword VIRTUAL as optional. In fact, it is used only for such an XMLType column, in which case it is *required*. It is an error to use it for a non-XMLType column. VIRTUAL specifies that the XMLType column itself is not materialized, meaning that its data is stored in the index only in the form of the relational columns specified by its corresponding XMLTable table.

Oracle Text Indexes on XML Data

You can create an Oracle Text index on an XMLType column. An Oracle Text CONTEXT index enables Oracle SQL function contains for full-text search over XML. With structured storage, XPath rewrite can often rewrite XPath function ora:contains to SQL function contains, so in those cases too an Oracle Text index can be employed.

See Also:

- Chapter 12, "Full-Text Search Over XML Data" for more information about using Oracle Text operations with Oracle XML DB
- Example 6–25, "Oracle Text CONTEXT Index on an XMLIndex Index Content Table" on page 6-24

Creating and Using Oracle Text Indexes

To create an Oracle Text index, use CREATE INDEX, specifying the INDEXTYPE as CTXSYS.CONTEXT, as illustrated in Example 6–43.

Example 6–43 Creating an Oracle Text Index

```
CREATE INDEX po_otext_ix ON po_clob (OBJECT_VALUE) INDEXTYPE IS CTXSYS.CONTEXT;
```

You can also perform Oracle Text operations such as `contains` and `score` on `XMLType` columns. Example 6–44 shows an Oracle Text search using Oracle SQL function `contains`.

Example 6–44 Searching XML Data using SQL Function CONTAINS

```
SELECT DISTINCT XMLCast(XMLQuery('$p/PurchaseOrder/ShippingInstructions/address'
                                 PASSING po.OBJECT_VALUE AS "p" RETURNING CONTENT)
                        AS VARCHAR2(256)) "Address"
  FROM po_clob po
  WHERE contains(po.OBJECT_VALUE,
                 '$(Fortieth) INPATH (PurchaseOrder/ShippingInstructions/address)') > 0;

Address
------------------------------
1200 East Forty Seventh Avenue
New York
NY
10024
USA

1 row selected.
```

The execution plan for this query shows two ways that the Oracle Text `CONTEXT` index is used:

- It references the index explicitly, as a domain index.
- It refers to Oracle SQL function `contains` in the predicate information.

```
PLAN_TABLE_OUTPUT
-----------------------------------------------------------------------------------------
Plan hash value: 274475732

-----------------------------------------------------------------------------------------
| Id  | Operation                    | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |             |     7 | 14098 |    10  (10)| 00:00:01 |
|   1 |  HASH UNIQUE                 |             |     7 | 14098 |    10  (10)| 00:00:01 |
|   2 |   TABLE ACCESS BY INDEX ROWID| PO_CLOB     |     7 | 14098 |     9   (0)| 00:00:01 |
|*  3 |    DOMAIN INDEX              | PO_OTEXT_IX |       |       |     4   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - access("CTXSYS"."CONTAINS"(SYS_MAKEXML('2B0A2483AB140B35E040578C8A173FEC',523
              3,"XMLDATA"),'$(Fortieth) INPATH (PurchaseOrder/ShippingInstructions/address)')>0)

20 rows selected.
```

Oracle Text Indexes Are Used Independently of Other Indexes

Oracle Text indexing is completely orthogonal to the other types of indexing described in this chapter. Whenever Oracle SQL function `contains` or XPath function `ora:contains` is used, an Oracle Text index can be used for full-text search.

Example 6–45 demonstrates this in the case where both an `XMLIndex` index and an Oracle Text index are defined on the same XML data. The query is the same as in Example 6–44. The Oracle Text index is created on the `VALUE` column of the `XMLIndex` path table of Example 6–11.

Example 6–45 Using an Oracle Text Index and an XMLIndex Index

```
CREATE INDEX po_otext_ix ON my_path_table (VALUE) INDEXTYPE IS CTXSYS.CONTEXT;

EXPLAIN PLAN FOR
  SELECT DISTINCT XMLCast(XMLQuery('$p/PurchaseOrder/ShippingInstructions/address'
                                   PASSING po.OBJECT_VALUE AS "p" RETURNING CONTENT)
                          AS VARCHAR2(256)) "Address"
    FROM po_clob po
    WHERE contains(po.OBJECT_VALUE,
                   '$(Fortieth) INPATH (PurchaseOrder/ShippingInstructions/address)') > 0;

-----------------------------------------------------------------------------------------------------------
| Id  | Operation                    | Name                          | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                               |     7 | 14098 |     3  (34)| 00:00:01 |
|   1 |  SORT GROUP BY               |                               |     1 |  3524 |            |          |
|*  2 |   TABLE ACCESS BY INDEX ROWID| MY_PATH_TABLE                 |     2 |  7048 |     3   (0)| 00:00:01 |
|*  3 |    INDEX RANGE SCAN          | SYS67616_PO_XMLINDE_PIKEY_IX  |     1 |       |     2   (0)| 00:00:01 |
|   4 |  HASH UNIQUE                 |                               |     7 | 14098 |     3  (34)| 00:00:01 |
|*  5 |   TABLE ACCESS FULL          | PO_CLOB                       |     7 | 14098 |     2   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - filter(SYS_XMLI_LOC_ISNODE("SYS_P0"."LOCATOR")=1)
   3 - access("SYS_P0"."RID"=:B1 AND "SYS_P0"."PATHID"=HEXTORAW('6F7F') )
   5 - filter("CTXSYS"."CONTAINS"(SYS_MAKEXML(0,"SYS_ALIAS_1"."XMLDATA"),'$(Fortieth) INPATH
              (PurchaseOrder/ShippingInstructions/address)')>0)

Note
-----
   - dynamic sampling used for this statement (level=2)
   - Unoptimized XML construct detected (enable XMLOptimizationCheck for more information)

24 rows selected.
```

The execution plan in Example 6–45 references both the `XMLIndex` index and the Oracle Text index, indicating that both are used.

- The `XMLIndex` index is indicated by its path table, `MY_PATH_TABLE`, and its order-key index, `SYS78942_PO_XMLINDE_ORDKEY_IX`.
- The Oracle Text index is indicated by the reference to Oracle SQL function `contains` in the predicate information.

7

XML Schema Storage and Query: Basic

The XML Schema Recommendation was created by the World Wide Web Consortium (W3C) to describe the content and structure of XML documents in XML. It includes the full capabilities of Document Type Definitions (DTDs) so that existing DTDs can be converted to XML Schema. XML schemas have additional capabilities compared to DTDs.

This chapter provides basic information about using XML Schema with Oracle XML DB. It explains how to do the following:

- Register, update, and delete an XML schema
- Create storage structures for XML schema-based data
- Map XML Schema data types to SQL data types and binary XML encoding types

This chapter contains these topics:

- Overview of XML Schema and Oracle XML DB
- Using Oracle XML DB with XML Schema
- Managing XML Schemas with DBMS_XMLSCHEMA
- XMLType Methods Related to XML Schema
- Local and Global XML Schemas
- DOM Fidelity
- XML Translations
- Creating XMLType Tables and Columns Based on XML Schemas
- Oracle XML Schema Annotations
- Querying a Registered XML Schema to Obtain Annotations
- Mapping XML Schema Data Types to Oracle XML DB Storage
- Mapping XML Schema Data Types to SQL Data Types
- Mapping XML Schema Data Types To Binary XML Encoding Types

See Also:

- Chapter 9, "XML Schema Storage and Query: Advanced" for more advanced information about using XML Schema with Oracle XML DB
- Chapter 8, "XPath Rewrite for Structured Storage" for information about the optimization of XPath expressions in Oracle XML DB
- `http://www.w3.org/TR/xmlschema-0/` for an introduction to XML Schema

Overview of XML Schema and Oracle XML DB

XML Schema is a schema definition language written in XML. It can be used to describe the structure and semantics of conforming instance documents. For example, the following XML schema definition, `purchaseOrder.xsd`, describes the structure and other properties of purchase-order XML documents.

This manual refers to an XML schema instance definition as an **XML schema**.

Example 7–1 shows an XML schema that declares a `complexType` called `purchaseOrderType` and a global element `PurchaseOrder` of this type. This is the same schema as Example 3–9, "Purchase-Order XML Schema, purchaseOrder.xsd", with the exception of the lines in **`bold`** here, which are additional. For brevity, part of the schema is replaced here by an ellipsis (**`...`**).

Example 7–1 XML Schema Instance purchaseOrder.xsd

```
<xs:schema
  targetNamespace="http://xmlns.oracle.com/xdb/documentation/purchaseOrder"
  xmlns:po="http://xmlns.oracle.com/xdb/documentation/purchaseOrder"
  xmlns:xs="http://www.w3.org/2001/XMLSchema" version="1.0">
  <xs:element name="PurchaseOrder" type="po:PurchaseOrderType"/>
   <xs:complexType name="PurchaseOrderType">
    <xs:sequence>
        <xs:element name="Reference" type="po:ReferenceType"/>
        <xs:element name="Actions" type="po:ActionsType"/>
        <xs:element name="Reject" type="po:RejectionType" minOccurs="0"/>
        <xs:element name="Requestor" type="po:RequestorType"/>
        <xs:element name="User" type="po:UserType"/>
        <xs:element name="CostCenter" type="po:CostCenterType"/>
        <xs:element name="ShippingInstructions"
                    type="po:ShippingInstructionsType"/>
        <xs:element name="SpecialInstructions"
                    type="po:SpecialInstructionsType"/>
        <xs:element name="LineItems" type="po:LineItemsType"/>
        <xs:element name="Notes" type="po:NotesType"/>
      </xs:sequence>
    </xs:complexType>
    <xs:complexType name="LineItemsType">
      <xs:sequence>
        <xs:element name="LineItem" type="po:LineItemType"
                    maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>

    ...
```

```
    <xs:simpleType name="DescriptionType">
      <xs:restriction base="xs:string">
        <xs:minLength value="1"/>
        <xs:maxLength value="256"/>
      </xs:restriction>
    </xs:simpleType>
    <xs:simpleType name="NotesType">
      <xs:restriction base="xs:string">
        <xs:minLength value="1"/>
        <xs:maxLength value="32767"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:schema>
```

Example 7–2 shows an XML document that conforms to XML schema `purchaseOrder.xsd`:

Example 7–2 purchaseOrder.xml: Document That Conforms to purchaseOrder.xsd

```
<po:PurchaseOrder xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:po="http://xmlns.oracle.com/xdb/documentation/purchaseOrder"
  xsi:schemaLocation=
    "http://xmlns.oracle.com/xdb/documentation/purchaseOrder
     http://xmlns.oracle.com/xdb/documentation/purchaseOrder.xsd">
  <Reference>SBELL-2002100912333601PDT</Reference>
  <Actions>
    <Action>
      <User>SVOLLMAN</User>
    </Action>
  </Actions>
  <Reject/>
  <Requestor>Sarah J. Bell</Requestor>
  <User>SBELL</User>
  <CostCenter>S30</CostCenter>
  <ShippingInstructions>
    <name>Sarah J. Bell</name>
    <address>400 Oracle Parkway
             Redwood Shores
             CA
             94065
             USA
    </address>
    <telephone>650 506 7400</telephone>
  </ShippingInstructions>
  <SpecialInstructions>Air Mail</SpecialInstructions>
  <LineItems>
    <LineItem ItemNumber="1">
      <Description>A Night to Remember</Description>
      <Part Id="715515009058" UnitPrice="39.95" Quantity="2"/>
    </LineItem>
    <LineItem ItemNumber="2">
      <Description>The Unbearable Lightness Of Being</Description>
      <Part Id="37429140222" UnitPrice="29.95" Quantity="2"/>
    </LineItem>
    <LineItem ItemNumber="3">
      <Description>Sisters</Description>
      <Part Id="715515011020" UnitPrice="29.95" Quantity="4"/>
    </LineItem>
  </LineItems>
  <Notes>Section 1.10.32 of &quot;de Finibus Bonorum et Malorum&quot;,
```

```
         written by Cicero in 45 BC

&quot;Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusantium
 doloremque laudantium, totam rem aperiam, eaque ips
...
tiae consequatur, vel illum qui dolorem eum fugiat quo voluptas nulla
 pariatur?&quot;

1914 translation by H. Rackham

&quot;But I must explain to you how all this mistaken idea of denouncing
 pleasure and praising pain was born and I will give you a c
...
o avoids a pain that produces no resultant pleasure?&quot;

Section 1.10.33 of &quot;de Finibus Bonorum et Malorum&quot;, written by Cicero
in 45 BC

&quot;At vero eos et accusamus et iusto odio dignissimos ducimus qui blanditiis
 praesentium voluptatum deleniti atque corrupti quos
...
 delectus, ut aut reiciendis voluptatibus maiores alias
consequatur aut perferendis doloribus asperiores repellat.&quot;

1914 translation by H. Rackham

&quot;On the other hand, we denounce with righteous indignation and dislike men
 who are so beguiled and demoralized by the charms of
...
secure other greater pleasures, or else he endures pains to avoid worse
 pains.&quot;
  </Notes>
</po:PurchaseOrder>
```

Note: The URL used is a name that uniquely identifies the registered XML schema within the database: `http://xmlns.oracle.com/xdb/documentation/purchaseOrder.xsd`. This need not point to a location where the XML schema document is located. The target namespace of the XML schema is another URL, different from the XML schema location URL, which specifies an abstract namespace within which elements and types get declared.

An XML schema can optionally specify the target namespace URL. If this attribute is omitted, the XML schema has no target namespace. The target namespace is commonly the same as the URL of the XML schema.

An XML instance document must specify the namespace of the root element (same as the target namespace of the XML schema) and the location (URL) of the XML schema that defines this root element. The location is specified with attribute `xsi:schemaLocation`. When the XML schema has no target namespace, use attribute `xsi:noNamespaceSchemaLocation` to specify the schema URL.

Using Oracle XML DB with XML Schema

Oracle XML DB exploits the strong typing and other powerful properties of XML Schema to process XML database data safely and efficiently.

> **Note:** With XML data that is stored using binary XML storage, you can use a DTD to obtain the XML entities defined there. However, the structural and type information in the DTD is not used by Oracle XML DB. The entities are the only information used.

Oracle XML DB uses annotated XML schemas as metadata. The standard XML Schema definitions are used, along with several Oracle namespace attributes. These attributes determine how XML instance documents get mapped to the database. Because these attributes are in a different namespace from the XML Schema namespace, such annotated XML schemas are legal XML Schema documents.

> **See Also:** `http://www.w3.org/2001/XMLSchema`

When using Oracle XML DB with XML Schema, you must first *register* the XML schema. You can then use the XML schema URLs while creating `XMLType` tables, columns, and views. The XML schema URL identifies the XML schema in the database. It is associated with parameter `SCHEMAURL` of PL/SQL procedure `DBMS_XMLSCHEMA.registerSchema`.

Oracle XML DB provides XML Schema support for the following tasks:

- Registering W3C-compliant XML schemas, both local and global.
- Validating your XML documents against registered XML schema definitions.
- Generating XML schemas from SQL object types.
- Referencing an XML schema owned by another user.
- Explicitly referencing a global XML schema when a local XML schema exists with the same name.
- Generating a database mapping from your XML schemas during XML schema registration. This includes generating SQL object types, collection types, and default tables, and capturing the mapping information using XML schema attributes.
- Specifying a particular SQL data type mapping when there are multiple allowed mappings.
- Creating `XMLType` tables, views, and columns based on registered XML schemas.
- Manipulating and querying XML schema-based `XMLType` tables.
- Automatically inserting data into default tables when XML schema-based documents are inserted into Oracle XML DB Repository using protocols (FTP, HTTP(S)/WebDAV) and languages besides SQL.

> **See Also:** Chapter 3, "Using Oracle XML DB"

Why XML Schema?

`XMLType` is an abstract data type that facilitates storing XML data in database columns and tables. XML Schema offers you additional storage and access options for XML

data. You can use XML schemas to define which XML elements and attributes, which kinds of element nesting, and which data types can be used.

XML Schema lets you verify that your XML data conforms to its intended definition: the data is validated against the XML schemas that define its proper structure. This definition includes data types, numbers of allowed item occurrences, and allowed lengths of items. When storing XML Schema-based documents in Oracle XML DB using protocols such as FTP or HTTP(S), the XML schema information can improve the efficiency of document insertion. When XML instances must be handled without any prior information about them, XML schemas can be useful in predicting optimum storage, fidelity, and access.

You can take advantage of XML Schema, including its strong typing, for XML data that is unstructured, semi-structured, or structured by storing it as binary XML.

If your XML data is highly structured, then consider storing it object-relationally. In that case, XML Schema is used to efficiently map XML (Schema) data types to SQL data types and object-relational tables and columns.

DTD Support in Oracle XML DB

A **DTD** is a set of rules that define the allowable structure of an XML document. DTDs are text files that derive their format from SGML and can be associated with an XML document either by using the `DOCTYPE` element or by using an external file through a `DOCTYPE` reference. In addition to supporting XML Schema, which provides a structured mapping to object-relational storage or binary XML storage, Oracle XML DB also supports DTD specifications in XML instance documents. Though DTDs are not used to derive the mapping, XML processors can still access and interpret the DTDs.

Inline DTD Definitions

When an XML instance document has an inline DTD definition, it is used during document parsing. Any DTD validations and entity declaration handling is done at this point. However, once parsed, the entity references are replaced with actual values and the original entity reference is lost.

External DTD Definitions

Oracle XML DB also supports external DTD definitions if they are stored in Oracle XML DB Repository. Applications needing to process an XML document containing an external DTD definition such as `/public/flights.dtd` must first ensure that the DTD document is stored in Oracle XML DB at path `/public/flights.dtd`.

> **See Also:** Chapter 21, "Accessing Oracle XML DB Repository Data"

Managing XML Schemas with DBMS_XMLSCHEMA

Before an XML schema can be used by Oracle XML DB, it must be registered with Oracle Database. You register an XML schema using the PL/SQL package `DBMS_XMLSCHEMA`.

> **See Also:** *Oracle Database PL/SQL Packages and Types Reference*

Some of the main `DBMS_XMLSCHEMA` procedures are these:

- `registerSchema` – Register an XML schema with Oracle Database

- `deleteSchema` – Delete a previously registered XML schema.
- `copyEvolve` – Update a registered XML schema. See Chapter 10, "XML Schema Evolution".

Registering an XML Schema with Oracle XML DB

The main parameters to procedure `DBMS_XMLSCHEMA.registerSchema` are these:

- `SCHEMAURL` – the XML schema URL. This is a unique identifier for the XML schema within Oracle XML DB. It is conventionally in the form of a URL, but this is not a requirement. The XML schema URL is used with Oracle XML DB to identify instance documents, by making the schema location hint identical to the XML schema URL. Oracle XML DB never tries to access a Web server identified by the specified URL.

  Note: You cannot register an XML schema using the same `SCHEMAURL` as any system-defined XML schema.

- `SCHEMADOC` – The XML schema source document. This is a `VARCHAR`, `CLOB`, `BLOB`, `BFILE`, `XMLType`, or `URIType` value.
- `CSID` – The character-set ID of the source-document encoding, when `schemaDoc` is a `BFILE` or `BLOB` value.
- `OPTIONS` – Options that specify how the XML schema should be registered. The most important option is `REGISTER_BINARYXML`, which indicates that the XML schema is used for binary XML storage. Another option is `REGISTER_NT_AS_IOT`, which forces OCTs to be stored as index-organized tables (IOTs).

  Note: If you specify option `REGISTER_BINARYXML`, then you must also set parameter `GENTYPES` to `FALSE`.

  See Also: *Oracle Database PL/SQL Packages and Types Reference*

The code in Example 7–3 registers the XML schema at URL `http://xmlns.oracle.com/xdb/documentation/purchaseOrder.xsd`. This example shows how to register an XML schema using the `BFILE` mechanism to read the source document from a file on the local file system of the database server.

Example 7–3 Registering an XML Schema using DBMS_XMLSCHEMA.REGISTERSCHEMA

```
BEGIN
  DBMS_XMLSCHEMA.registerSchema(
    SCHEMAURL => 'http://xmlns.oracle.com/xdb/documentation/purchaseOrder.xsd',
    SCHEMADOC => bfilename('XMLDIR','purchaseOrder.xsd'),
    CSID      => nls_charset_id('AL32UTF8'));
END;
/
```

Delete and Reload Documents Before Registering Their XML Schema

When you register an XML schema, keep in mind that the act of registering a schema has *no effect* on the status of any instance documents already loaded into Oracle XML DB Repository that reference the XML schema. Because the XML schema was not yet registered, such instance documents were non-schema-based when they were loaded. *They remain non-schema-based after the schema is registered*.

You must *delete* such instance documents, and *reload* them after registering the schema, in order to obtain schema-based documents.

Storage and Access Infrastructure

As part of registering an XML schema, Oracle XML DB also performs several tasks that facilitate storing, accessing, and manipulating XML instances that conform to the XML schema. These steps include:

- Mapping XML Schema data types to Oracle XML DB storage. When XML schema-based data is stored, its storage data types are derived from the XML Schema data types using a default mapping and, optionally, using mapping information that you specify using XML schema annotations. For binary XML storage, XML Schema types are mapped to binary XML encoding types. For object-relational storage, XML schema registration creates the appropriate SQL object types for the structured storage of conforming documents.
- Creating default tables. XML schema registration generates default `XMLType` tables for all global elements. You can use XML-schema annotations to control the names of the tables, and to provide column-level and table-level storage clauses and constraints for use during table creation.

  See Also:

  - "Mapping XML Schema Data Types to Oracle XML DB Storage" on page 7-44
  - "Default Tables Created During XML Schema Registration" on page 7-10
  - "Oracle XML Schema Annotations" on page 7-34

After XML schema registration, documents that reference the XML schema using the XML Schema instance mechanism can be processed automatically by Oracle XML DB. For XML data that is stored object-relationally, `XMLType` tables and columns can be created that are constrained to the global elements defined by the XML schema.

See Also: Chapter 3, "Using Oracle XML DB"

Atomic Nature of XML Schema Registration

Like all DDL operations, XML schema registration is non-transactional. However, registration is *atomic*, in this sense:

- If registration succeeds, then the operation is auto-committed.
- If registration fails, then the database is rolled back to the state before registration began.

Because XML schema registration potentially involves creating object types and tables, error recovery involves dropping any types and tables thus created. The entire XML schema registration process is guaranteed to be atomic: either it succeeds or the database is restored to its state before the start of registration.

Managing and Storing XML Schemas

XML schema documents are themselves stored in Oracle XML DB as `XMLType` instances. XML schema-related `XMLType` types and tables are created as part of the Oracle XML DB installation script, `catxdbs.sql`.

The XML schema for Oracle XML DB XML schemas is called the **root XML Schema**, `XDBSchema.xsd`. The root XML schema describes any valid XML schema that can be registered with Oracle XML DB. You can access `XDBSchema.xsd` at Oracle XML DB Repository location `/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/XDBSchema.xsd`.

> **See Also:** Chapter 34, "Administering Oracle XML DB"

Debugging XML Schema Registration for XML Data Stored Object-Relationally

For XML data stored object-relationally, you can monitor the object types and tables created during XML schema registration by setting the following event before invoking PL/SQL procedure `DBMS_XMLSCHEMA.registerSchema`:

```
ALTER SESSION SET EVENTS = '31098 TRACE NAME CONTEXT FOREVER'
```

Setting this event causes the generation of a log of all of the `CREATE TYPE` and `CREATE TABLE` statements. This log is written to the user session trace file, typically found in `ORACLE_BASE/diag/rdbms/ORACLE_SID/ORACLE_SID/udump`. This script can be a useful aid in diagnosing problems during XML schema registration.

> **See Also:** "Using XML Schema with Oracle XML DB" on page 3-18

SQL Object Types Created During XML Schema Registration, for Structured Storage

If parameter `GENTYPES` is `TRUE` when an XML schema is registered for use with XML data stored object-relationally, then Oracle XML DB creates the appropriate SQL object types that enable structured storage of conforming XML documents. By default, all SQL object types are created in the database schema of the user who registers the XML schema. If annotation `xdb:defaultSchema` is used, then Oracle XML DB attempts to create the object type using the specified database schema. The current user must have the necessary privileges to create these object types.

Example 7–4 shows the SQL object types that are created automatically when XML schema `purchaseOrder.xsd` is registered with Oracle XML DB.

Example 7–4 Creating SQL Object Types to Store XMLType Tables

```
DESCRIBE "PurchaseOrderType1668_T"

"PurchaseOrderType1668_T" is NOT FINAL
Name                 Null?  Type
-------------------- ------ -------------------------------
SYS_XDBPD$                  XDB.XDB$RAW_LIST_T
Reference                   VARCHAR2(30 CHAR)
Actions                     ActionsType1661_T
Reject                      RejectionType1660_T
Requestor                   VARCHAR2(128 CHAR)
User                        VARCHAR2(10 CHAR)
CostCenter                  VARCHAR2(4 CHAR)
ShippingInstructions        ShippingInstructionsTyp1659_T
SpecialInstructions         VARCHAR2(2048 CHAR)
LineItems                   LineItemsType1666_T
```

```
Notes                       VARCHAR2(4000 CHAR)

DESCRIBE "LineItemsType1666_T"

"LineItemsType1666_T" is NOT FINAL
Name                 Null? Type
-------------------- ----- -------------------------------
SYS_XDBPD$                 XDB.XDB$RAW_LIST_T
LineItem                   LineItem1667_COLL

DESCRIBE "LineItem1667_COLL"

"LineItem1667_COLL" VARRAY(2147483647) OF LineItemType1665_T
"LineItemType1665_T" is NOT FINAL
Name                 Null? Type
-------------------- ----- -------------------------------
SYS_XDBPD$                 XDB.XDB$RAW_LIST_T
ItemNumber                 NUMBER(38)
Description                VARCHAR2(256 CHAR)
Part                       PartType1664_T
```

Note: By default, the names of the SQL object types and attributes are system-generated. This is the case in Example 7–4. If the XML schema does not contain attribute `SQLName`, then the SQL name is derived from the XML name. You can use XML schema annotations to provide user-defined names (see "Oracle XML Schema Annotations" for details).

Default Tables Created During XML Schema Registration

As part of XML schema registration for XML data, you can create default tables. Default tables are most useful when documents conforming to the XML schema are inserted through APIs and protocols such as FTP and HTTP(S) that do not provide any table specification. In such cases, the XML instance is inserted into the default table.

Example 7–5 Default Table for Global Element PurchaseOrder

```
DESCRIBE "purchaseorder1669_tab"

Name                         Null? Type
---------------------------- ----- -----------------------
TABLE of
  SYS.XMLTYPE(
    XMLSchema "http://xmlns.oracle.com/xdb/documentation/purchaseOrder.xsd"
    Element "PurchaseOrder")
  STORAGE OBJECT-RELATIONAL TYPE "PurchaseOrderType1668_T"
```

If you provide a value for attribute `xdb:defaultTable`, then the `XMLType` table is created with that name. Otherwise it is created with an internally generated name.

Any text specified using attributes `xdb:tableProps` and `xdb:columnProps` is appended to the generated `CREATE TABLE` statement.

Generated Names are Case Sensitive

The names of any SQL tables, object, and attributes generated by XML schema registration are *case sensitive*. For instance, in Example 7–3 on page 7-7, a table named

PurchaseOrder1669_TAB is created automatically during registration of the XML schema. Because this table name was derived from the element name, PurchaseOrder, the table name is also mixed case. You must therefore refer to this table in SQL code by using a quoted identifier: "PurchaseOrder1669_TAB". Failure to do so results in an object-not-found error, such as ORA-00942: table or view does not exist.

Database Objects That Depend on Registered XML Schemas

The following database objects are dependent on registered XML schemas:

- Tables or views that have an XMLType column that conforms to an element in an XML schema.
- Other XML schemas that include or import a given XML schema as part of their definition.
- Cursors that reference an XML schema. This includes references within functions of package DBMS_XMLGEN. Such cursors are purely transient objects.

Listing All Registered XML Schemas

Example 7–6 shows how to use PL/SQL procedure DBMS_XMLSCHEMA.registerSchema to obtain a list of all XML schemas registered with Oracle XML DB. You can also examine views USER_XML_SCHEMAS, ALL_XML_SCHEMAS, USER_XML_TABLES, and ALL_XML_TABLES.

Example 7–6 Data Dictionary Table for Registered Schemas

```
DESCRIBE DBA_XML_SCHEMAS

Name         Null? Type
------------ ----- -----------------------
OWNER              VARCHAR2(30)
SCHEMA_URL         VARCHAR2(700)
LOCAL              VARCHAR2(3)
SCHEMA             XMLTYPE(XMLSchema "http://xmlns.oracle.com/xdb/XDBSchema.xsd"
                           Element "schema")
INT_OBJNAME        VARCHAR2(4000)
QUAL_SCHEMA_URL    VARCHAR2(767)
HIER_TYPE          VARCHAR2(11)
BINARY             VARCHAR2(3)
SCHEMA_ID          RAW(16)
HIDDEN             VARCHAR2(3)

SELECT OWNER, LOCAL, SCHEMA_URL FROM DBA_XML_SCHEMAS;

OWNER   LOC   SCHEMA_URL
-----   ---   ---------------------
XDB     NO    http://xmlns.oracle.com/xdb/XDBSchema.xsd
XDB     NO    http://xmlns.oracle.com/xdb/XDBResource.xsd
XDB     NO    http://xmlns.oracle.com/xdb/acl.xsd
XDB     NO    http://xmlns.oracle.com/xdb/dav.xsd
XDB     NO    http://xmlns.oracle.com/xdb/XDBStandard.xsd
XDB     NO    http://xmlns.oracle.com/xdb/log/xdblog.xsd
XDB     NO    http://xmlns.oracle.com/xdb/log/ftplog.xsd
XDB     NO    http://xmlns.oracle.com/xdb/log/httplog.xsd
XDB     NO    http://www.w3.org/2001/xml.xsd
XDB     NO    http://xmlns.oracle.com/xdb/XDBFolderListing.xsd
XDB     NO    http://xmlns.oracle.com/xdb/stats.xsd
```

```
XDB     NO    http://xmlns.oracle.com/xdb/xdbconfig.xsd
SCOTT   YES   http://xmlns.oracle.com/xdb/documentation/purchaseOrder.xsd

13 rows selected.

DESCRIBE DBA_XML_TABLES

Name         Null? Type
------------ ----- -----------------------
OWNER              VARCHAR2(30)
TABLE_NAME         VARCHAR2(30)
XMLSCHEMA          VARCHAR2(700)
SCHEMA_OWNER       VARCHAR2(30)
ELEMENT_NAME       VARCHAR2(2000)
STORAGE_TYPE       VARCHAR2(17)
ANYSCHEMA          VARCHAR2(3)
NONSCHEMA          VARCHAR2(3)

SELECT TABLE_NAME FROM DBA_XML_TABLES
  WHERE XMLSCHEMA = 'http://xmlns.oracle.com/xdb/documentation/purchaseOrder.xsd';

TABLE_NAME
---------------------
PurchaseOrder1669_TAB

1 row selected.
```

Deleting an XML Schema

You can delete a registered XML schema by using procedure `DBMS_XMLSCHEMA.`**`deleteSchema`**. This does the following, by default:

1. Checks that the current user has the appropriate privileges to delete the resource corresponding to the XML schema within Oracle XML DB Repository. You can control which users can delete which XML schemas, by setting the appropriate ACLs on the XML schema resources.
2. Checks whether there are any tables dependent on the XML schema that is to be deleted. If so, raises an error and cancels the deletion. This check is not performed if option `delete_invalidate` or `delete_cascade_force` is used. In that case, no error is raised.
3. Removes the XML schema document from the Oracle XML DB Repository (folder `/sys/schemas`).
4. Removes the XML schema document from `DBA_XML_SCHEMAS`, unless it was registered for use with binary XML instances and neither `delete_invalidate` nor `delete_cascade_force` is used.
5. Drops the default table, if either `delete_cascade` or `delete_cascade_force` is used. Raises an error if `delete_cascade_force` is specified and there are instances in other tables that are also dependent on the XML schema.

DBMS_XMLSCHEMA.DELETESCHEMA Options

The following values are available for option `DELETE_OPTION` of procedure `DBMS_XMLSCHEMA.deleteSchema`:

- `DELETE_RESTRICT` – Raise an error and cancel deletion if dependencies are detected. This is the default behavior.

- DELETE_INVALIDATE – Do not raise an error if dependencies are detected. Instead, mark each of the dependencies as being invalid. If the XML schema was registered for use with binary XML, do not remove it from DBA_XMLSCHEMAS.
- DELETE_CASCADE – Drop all types and default tables that were generated during XML schema registration. Raise an error if there are instances that depend upon the XML schema that are stored in tables other than the default table. However, do not raise an error for any such instances that are stored in XMLType columns that were created using ANY_SCHEMA. If the XML schema was registered for use with binary XML, do not remove it from DBA_XMLSCHEMAS.
- DELETE_CASCADE_FORCE – Drop all types and default tables that were generated during XML schema registration. Do not raise an error if there are instances that depend upon the XML schema that are stored in tables other than the default table. Instead, mark each of the dependencies as being invalid. Remove the XML schema from DBA_XMLSCHEMAS.

See Also: *Oracle Database PL/SQL Packages and Types Reference*

Example 7–7 illustrates the use of DELETE_CASCADE_FORCE.

Example 7–7 Deleting an XML Schema with DBMS_XMLSCHEMA.DELETESCHEMA

```
BEGIN
  DBMS_XMLSCHEMA.deleteSchema(
    SCHEMAURL => 'http://xmlns.oracle.com/xdb/documentation/purchaseOrder.xsd',
    DELETE_OPTION => DBMS_XMLSCHEMA.DELETE_CASCADE_FORCE);
END;
/
```

If an XML schema was registered for use with binary XML, it is not removed from DBA_XMLSCHEMAS when you delete it using option DELETE_RESTRICT (the default value) or DELETE_CASCADE. As a consequence, although you can no longer use the XML schema to encode new XML instance documents, any existing documents in Oracle XML DB that reference the XML schema can still be *decoded* using it.

This remains the case, until you remove the XML schema from DBA_XMLSCHEMAS using DBMS_XMLSCHEMA.**purgeSchema**. Oracle recommends that, in general, you use delete_restrict or delete_cascade. Instead of using DELETE_CASCADE_FORCE, call DBMS_XMLSCHEMA.purgeSchema when you are sure you no longer need the XML schema.

Procedure purgeSchema removes the XML schema completely from Oracle XML DB. In particular, it removes it from DBA_XMLSCHEMAS. Before you use DBMS_XMLSCHEMA.purgeSchema, be sure that you have transformed all existing XML documents that reference the XML schema to be purged, so they reference a different XML schema or no XML schema. Otherwise, it will be impossible to decode them after the purge.

XMLType Methods Related to XML Schema

Table 7–1 lists some of the XMLType methods that are useful for working with XML schemas.

Table 7–1 XMLType Methods Related to XML Schema

XMLType Method	Description
`isSchemaBased()`	Returns `TRUE` if the `XMLType` instance is based on an XML schema, `FALSE` otherwise.
`getSchemaURL()`	The XML schema URL for an `XMLType` instance.

`schemaValidate()`
`isSchemaValid()`
`isSchemaValidated()`
`setSchemaValidated()`
	Validate an `XMLType` instance against a registered XML schema.
	See Chapter 11, "Transforming and Validating XMLType Data".

Local and Global XML Schemas

XML schemas can be registered as local or global:

- A local xml schema is, by default, visible only to its owner.
- A global xml schema is, by default, visible and usable by all database users.

When you register an XML schema, PL/SQL package `DBMS_XMLSCHEMA` adds a corresponding resource to Oracle XML DB Repository. The XML schema URL determines the path name of the XML schema resource in the repository (and it is associated with parameter `SCHEMAURL` of PL/SQL procedure `DBMS_XMLSCHEMA.registerSchema`).

Note: In Oracle Enterprise Manager, local and global registered XML schemas are referred to as **private** and **public**, respectively.

Local XML Schema

By default, an XML schema belongs to you after you register it with Oracle XML DB. A reference to the XML schema document is stored in Oracle XML DB Repository. Such XML schemas are referred to as **local**. By default, they are usable only by you, the owner. In Oracle XML DB, local XML schema resources are created under folder `/sys/schemas/`*`username`*. The rest of the repository path name is derived from the schema URL.

Example 7–8 Registering a Local XML Schema

```
BEGIN
  DBMS_XMLSCHEMA.registerSchema(
    SCHEMAURL => 'http://xmlns.oracle.com/xdb/documentation/purchaseOrder.xsd',
    SCHEMADOC => bfilename('XMLDIR','purchaseOrder.xsd'),
    LOCAL     => TRUE,
    GENTYPES  => TRUE,
    GENTABLES => FALSE,
    CSID      => nls_charset_id('AL32UTF8'));
END;
/
```

If this local XML schema is registered by user `QUINE`, it is given this path name:

```
/sys/schemas/QUINE/xmlns.oracle.com/xdb/documentation/purchaseOrder.xsd
```

Database users need appropriate permissions and Access Control Lists (ACLs) to create a resource with this path name, in order to register the XML schema as a local XML schema.

See Also: Chapter 27, "Repository Access Control"

Note: Typically, only the owner of the XML schema can use it to define `XMLType` tables, columns, or views, validate documents, and so on. However, Oracle XML DB supports fully qualified XML schema URLs. For example: `http://xmlns.oracle.com/xdb/schemas/QUINE/xmlns.oracle.com/xdb/documentation/purchaseOrder.xsd`. Privileged users can use such an extended URL to specify XML schemas belonging to other users.

Global XML Schema

In contrast to local schemas, a privileged user can register an XML schema as global by specifying an argument in the `DBMS_XMLSCHEMA` registration function. **Global XML schemas** are visible to *all* users. They are stored under folder `/sys/schemas/PUBLIC/` in Oracle XML DB Repository.

Note: Access to folder `/sys/schemas/PUBLIC` is controlled by access control lists (ACLs). By default, this folder is writable only by a database administrator. You need write privileges on this folder to register global XML schemas. Role `XDBADMIN` provides write access to this folder, assuming that it is protected by the default ACLs. See Chapter 27, "Repository Access Control".

You can register a local schema with the same URL as an existing global schema. A local schema always shadows (hides) any global schema with the same name (URL). Example 7–9 illustrates registration of a global schema.

Example 7–9 Registering a Global XML Schema

```
GRANT XDBADMIN TO QUINE;

Grant succeeded.

CONNECT quine
Enter password: password

Connected.

BEGIN
  DBMS_XMLSCHEMA.registerSchema(
    SCHEMAURL => 'http://xmlns.oracle.com/xdb/documentation/purchaseOrder.xsd',
    SCHEMADOC => bfilename('XMLDIR','purchaseOrder.xsd'),
    LOCAL     => FALSE,
    GENTYPES  => TRUE,
    GENTABLES => FALSE,
    CSID      => nls_charset_id('AL32UTF8'));
END;
/
```

If this global XML schema is registered by user `QUINE`, it is given this path name:

```
/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/documentation/purchaseOrder.xsd
```

Database users need appropriate permissions (ACL access) to create this resource in order to register the XML schema as global.

DOM Fidelity

Document Object Model (DOM) fidelity is the concept of retaining the structure of a retrieved XML document, compared to the original XML document, for DOM traversals. DOM fidelity is needed to ensure the accuracy and integrity of XML documents stored in Oracle XML DB.

See Also:

- "Overriding the SQLType Value in an XML Schema When Declaring Attributes" on page 7-47
- "Overriding the SQLType Value in an XML Schema when Declaring Elements" on page 7-48

What is DOM Fidelity?

DOM fidelity means that all information in an XML document is preserved, except whitespace that is insignificant. With DOM fidelity, XML data retrieved from the database has the same information as before it was inserted into the database, with the single exception of insignificant whitespace. The term "DOM fidelity" is used because this kind of fidelity is particularly important for DOM traversals.

With binary XML storage of XML data, all of the significant information is encoded in the binary XML format, ensuring DOM fidelity. With structured storage of XML data, the elements and attributes declared in an XML schema are mapped to separate attributes in the corresponding SQL object types. However, the following information in XML instance documents is not stored in these object attributes:

- Namespace declarations
- Comments
- Prefix information

Instead, Oracle XML DB uses a separate mechanism to keep track of this information: it is recorded as instance-level metadata.

SYS_XDBPD$ and DOM Fidelity for Structured Storage

In order to provide DOM fidelity for XML data stored object-relationally, Oracle XML DB maintains instance-level metadata. This metadata is tracked at the type level using the system-defined binary object attribute `SYS_XDBPD$`. This object attribute is referred to as the **positional descriptor**, or **PD** for short. The PD is intended for Oracle XML DB *internal use only*. You should never directly access or manipulate column PD.

The positional descriptor stores all information that cannot be stored in any of the other object attributes. PD information is used to ensure the DOM fidelity of all XML documents stored in Oracle XML DB. Examples of PD information include: ordering information, comments, processing instructions, and namespace prefixes.

If DOM fidelity is not required, you can suppress the use of `SYS_XDBPD$` by setting attribute `xdb:maintainDOM` to `false` in the XML schema, at the type level.

Note: For clarity, object attribute SYS_XDBPD$ is omitted in many examples in this book. However, it is always present as a positional descriptor (PD) column in all SQL object types that are generated by the XML schema registration process.

In general, Oracle recommends that you do not suppress the PD attribute, because the extra information, such as comments and processing instructions, could be lost if there is no PD column.

XML Translations

You can store XML documents in Oracle XML DB Repository as XMLType instances. You can use any storage model for these instances: structured (object-relational storage), unstructured (CLOB), or binary XML. These documents sometimes contain strings that must be translated into various (natural) languages. You typically store both the original strings and their translations in the repository. You can retrieve and operate on these strings, depending on your language settings.

Note: XML schemas stored object-relationally are not translatable.

Changing an XML Schema and XML Instance Documents for Translation

This section describes the changes that are required to be made to an XML schema and an associated XML instance document to make the document translatable.

Indicating Translatable Elements in an XML Schema

Attribute xdb:translate must be specified in the XML schema for each element that is to be translated. The following restrictions apply to attribute xdb:translate.

1. Attribute xdb:translate can be specified only on complexType elements that have simpleContent. Here, simpleContent must be an extension or a restriction of type string. However, if a complexType element has the xdb:translate flag set, then none of its descendants can have this flag set.
2. Attribute xdb:translate can be set only on a single-valued element, which has exactly one translation. For such an element, the value of maxoccurs must be 0 or 1. If you want to set this attribute on a multiple-valued element, the element must have an ID attribute, which uniquely identifies the element.

During XML schema registration, PL/SQL procedure DBMS_XMLSCHEMA.registerSchema checks whether the XML schema satisfies these restrictions.

Indicating Translation Language Attributes in an XML Instance Document

The following translation language attributes are supported:

- xml:lang: For an instance document associated with an XML schema that supports translations, you must specify the translation language. You can do this by annotating each translation with attribute xml:lang. The allowed values of the xml:lang attribute are the language identifiers identified by IETF RFC 3066.
- xdb:srclang: For multiple-valued elements, that is, elements that can have multiple translations, only one translation can be used as the source language translation. That translation is specified by attribute xdb:srclang. This is the default translation, which is returned when the session language is not specified.

Making XML Documents Translatable

This section uses the translation-specifying XML schema attributes to make elements in a sample document translatable. Example 7–10 shows an XML schema that defines documents that contain a title string that needs to be translatable.

Example 7–10 XML Schema Defining Documents with a Title To Be Translated

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:xdbsc="http://xmlns.oracle.com/xdb/security.xsd"
        xmlns:xdb="http://xmlns.oracle.com/xdb.xsd"
        targetNamespace="http://xmlns.oracle.com/xdb/security.xsd"
        elementFormDefault="qualified" version="1.0">
  <annotation>
    <documentation>
This XML schema describes the structure of Security Class documents.
    </documentation>
  </annotation>
  <element name="securityClass" xdb:defaultTable="">
    <complexType>
      <sequence>
        <element name="name" type="string"/>
        <element name="title" minOccurs="0" maxOccurs="unbounded"/>
        <element name="inherits-from" type="QName" minOccurs="0" maxOccurs="unbounded"/>
        <element name="privlist" minOccurs="0" maxOccurs="unbounded">
          <complexType>
            <choice minOccurs="0" maxOccurs="unbounded">
              <element ref="xdbsc:privilege"/>
                <element ref="xdbsc:aggregatePrivilege"/>
            </choice>
          </complexType>
        </element>
        <!-- this "any" contains all application specific information
             for a security class in general e.g. reason for creation  -->
        <any namespace="##other" minOccurs="0" maxOccurs="unbounded"/>
      </sequence>
      <attribute name="targetNamespace" type="anyURI" use="required"/>
      <!-- all privileges in this security class are under this target namespace -->
    </complexType>
  </element>
  <element name="aggregatePrivilege">
    <complexType>
      <sequence>
        <element name="title" minOccurs="0" maxOccurs="unbounded"/>
        <sequence maxOccurs="unbounded">
          <element name="privilegeRef">
            <complexType>
              <attribute name="name" type="QName" use="required"/>
            </complexType>
          </element>
          <any namespace="##other" minOccurs="0" maxOccurs="unbounded"/>
        </sequence>
        <!-- this "any" contains all application specific information
             an aggregate privilege e.g. translations  -->
        <any namespace="##other" minOccurs="0" maxOccurs="unbounded"/>
      </sequence>
      <attribute name="name" type="string" use="required"/>
    </complexType>
  </element>
  <element name="privilege">
```

```
    <complexType>
      <sequence minOccurs="0">
        <element name="title" minOccurs="0" maxOccurs="unbounded"/>
        <sequence minOccurs="0" maxOccurs="unbounded">
          <element name="columnRef">
            <complexType>
              <attribute name="schema" type="string" use="required"/>
              <attribute name="table" type="string" use="required"/>
              <attribute name="column" type="string" use="required"/>
            </complexType>
          </element>
          <any namespace="##other" minOccurs="0" maxOccurs="unbounded"/>
        </sequence>
        <!-- this "any" contains all application specific information
             for a privilege e.g. translations  -->
        <any namespace="##other" minOccurs="0" maxOccurs="unbounded"/>
      </sequence>
      <attribute name="name" type="string" use="required"/>
    </complexType>
  </element>
</schema>
```

Example 7–11 shows a document that is associated with the XML schema of Example 7–10.

Example 7–11 Untranslated Instance Document

```
<securityClass xmlns="http://xmlns.oracle.com/xdb/security.xsd"
               xmlns:is="xmlns.oracle.com/iStore"
               xmlns:oa="xmlns.oracle.com/OracleApps"
               targetNamespace="xmlns.oracle.com/example">
  <name>
securityClassExample
  </name>
  <title>
Security Class Example
  </title>
  <inherits-from>is:iStorePurchaseOrder</inherits-from>
  <privlist>
  <privilege name="privilege1"/>
  <aggregatePrivilege name="iStorePOApprover">
    <title>
iStore Purchase Order Approver
    </title>
    <privilegeRef name="is:privilege1"/>
    <privilegeRef name="oa:submitPO"/>
    <privilegeRef name="oa:privilege3"/>
  </aggregatePrivilege>
  <privilege name="privilege2">
    <title>
secondary privilege
    </title>
    <columnRef schema="APPS" table="PurchaseOrder" column="POId"/>
    <columnRef schema="APPS" table="PurchaseOrder" column="Amount"/>
  </privilege>
  </privlist>
</securityClass>
```

To make the top-level title translatable, set `xdb:translate` to `true`. This is a single-valued element (`xdb:maxOccurs` is `1`). Example 7–12 shows the new XML schema, where attribute `xdb:translate` is `true`.

Example 7–12 XML Schema with Attribute xdb:translate for a Single-Valued Element

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:xdbsc="http://xmlns.oracle.com/xdb/security.xsd"
        xmlns:xdb="http://xmlns.oracle.com/xdb.xsd"
        targetNamespace="http://xmlns.oracle.com/xdb/security.xsd"
        elementFormDefault="qualified" version="1.0">
<xs:import namespace="http://www.w3.org/XML/1998/namespace"
        schemaLocation="http://www.w3.org/2001/xml.xsd"/>
<xs:import namespace-"http://xmlns.oracle.com/xdb"
        schemaLocation="http://xmlns.oracle.com/xdb/xmltr.xsd"/>
  <annotation>
    <documentation>
This XML schema describes the structure of Security Class documents.
    </documentation>
  </annotation>
  <element name="securityClass" xdb:defaultTable="">
    <complexType>
      <sequence>
        <element name="name" type="string"/>
        <element ref="titleref" minOccurs="0" maxOccurs="unbounded"
                 xdb:maxOccurs="1" xdb:translate="true"/>
        <element name="inherits-from" type="QName" minOccurs="0" maxOccurs="unbounded"/>
        <element name="privlist" minOccurs="0" maxOccurs="unbounded" xdb:maxOccurs="1">
          <complexType>
            <choice minOccurs="0" maxOccurs="unbounded">
              <element ref="xdbsc:privilege"/>
              <element ref="xdbsc:aggregatePrivilege"/>
            </choice>
          </complexType>
        </element>
        <!-- this "any" contains all application specific information
             for a security class in general e.g. reason for creation  -->
        <any namespace="##other" minOccurs="0" maxOccurs="unbounded"/>
      </sequence>
      <attribute name="targetNamespace" type="anyURI" use="required"/>
      <!-- all privileges in this security class are under this target namespace -->
    </complexType>
  </element>
  <element name="aggregatePrivilege">
    <complexType>
      <sequence>
        <element name="title" minOccurs="0" maxOccurs="unbounded"/>
        <sequence maxOccurs="unbounded">
          <element name="privilegeRef">
            <complexType>
              <attribute name="name" type="QName" use="required"/>
            </complexType>
          </element>
          <any namespace="##other" minOccurs="0" maxOccurs="unbounded"/>
        </sequence>
        <!-- this "any" contains all application specific information
             an aggregate privilege e.g. translations  -->
        <any namespace="##other" minOccurs="0" maxOccurs="unbounded"/>
      </sequence>
      <attribute name="name" type="string" use="required"/>
```

```
    </complexType>
  </element>
  <element name="privilege">
    <complexType>
      <sequence minOccurs="0">
        <element name="title" minOccurs="0" maxOccurs="unbounded"/>
        <sequence minOccurs="0" maxOccurs="unbounded">
          <element name="columnRef">
            <complexType>
              <attribute name="schema" type="string" use="required"/>
              <attribute name="table" type="string" use="required"/>
              <attribute name="column" type="string" use="required"/>
            </complexType>
          </element>
          <any namespace="##other" minOccurs="0" maxOccurs="unbounded"/>
        </sequence>
        <!-- this "any" contains all application specific information
             for a privilege e.g. translations  -->
        <any namespace="##other" minOccurs="0" maxOccurs="unbounded"/>
      </sequence>
      <attribute name="name" type="string" use="required"/>
    </complexType>
  </element>
  <element name="titleref">
    <complexType>
      <simpleContent>
        <extension base="xs:string">
          <attribute ref="xml:lang"/>
          <attribute ref="xdb:srclang"/>
        </extension>
      </simpleContent>
    </complexType>
  </element>
</schema>
```

Example 7–13 shows an instance document after translation of the title text.

Example 7–13 Translated Document

```
<securityClass xmlns="http://xmlns.oracle.com/xdb/security.xsd"
               xmlns:is="xmlns.oracle.com/iStore"
               xmlns:oa="xmlns.oracle.com/OracleApps">
  <name>
securityClassExample
  </name>
  <title xml:lang="en" xdb:srclang="true">
Security Class Example
  </title>
  <title xml:lang="es">
Security Class Example - Spanish
  </title>
  <title xml:lang="fr">
Security Class Example - French
  </title>
  <inherits-from>is:iStorePurchaseOrder</inherits-from>
  <privlist>
    <privilege name="privilege1"/>
    <aggregatePrivilege name="iStorePOApprover">
      <title>
iStore Purchase Order Approver
```

```
      </title>
      <privilegeRef name="is:privilege1"/>
      <privilegeRef name="oa:submitPO"/>
      <privilegeRef name="oa:privilege3"/>
    </aggregatePrivilege>
    <privilege name="privilege2">
      <title>
secondary privilege
      </title>
      <columnRef schema="APPS" table="PurchaseOrder" column="POId"/>
      <columnRef schema="APPS" table="PurchaseOrder" column="Amount"/>
    </privilege>
  </privlist>
</securityClass>
```

To make the title translatable in the case of a multi-valued element, you would set `xdb:maxOccurs` to `unbounded`. However, `xdb:translate` cannot be set to `true` for a multiple-valued element, unless there is an identifier attribute that uniquely identifies each element. Example 7–14 shows an XML schema that uses an identifier attribute `id` for the `title` element.

Example 7–14 XML Schema with Attribute xdb:translate for a Multi-Valued Element

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:xdbsc="http://xmlns.oracle.com/xdb/security.xsd"
        xmlns:xdb="http://xmlns.oracle.com/xdb.xsd"
        targetNamespace="http://xmlns.oracle.com/xdb/security.xsd"
        elementFormDefault="qualified" version="1.0">
<xs:import namespace="http://www.w3.org/XML/1998/namespace"
        schemaLocation="http://www.w3.org/2001/xml.xsd"/>
<xs:import namespace-"http://xmlns.oracle.com/xdb"
        schemaLocation="http://xmlns.oracle.com/xdb/xmltr.xsd"/>
  <annotation>
    <documentation>
This XML schema describes the structure of Security Class documents.
    </documentation>
  </annotation>
  <element name="securityClass" xdb:defaultTable="">
    <complexType>
      <sequence>
        <element name="name" type="string"/>
        <element name="title" minOccurs="0" maxOccurs="unbounded" xdb:maxOccurs="1"/>
        <element name="inherits-from" type="QName" minOccurs="0" maxOccurs="unbounded"/>
        <element name="privlist" minOccurs="0" maxOccurs="unbounded" xdb:maxOccurs="1">
          <complexType>
            <choice minOccurs="0" maxOccurs="unbounded">
              <element ref="xdbsc:privilege"/>
              <element ref="xdbsc:aggregatePrivilege"/>
            </choice>
          </complexType>
        </element>
        <!-- this "any" contains all application specific information
             for a security class in general e.g. reason for creation  -->
        <any namespace="##other" minOccurs="0" maxOccurs="unbounded"/>
      </sequence>
      <attribute name="targetNamespace" type="anyURI" use="required"/>
      <!-- all privileges in this security class are under this target namespace -->
    </complexType>
  </element>
  <element name="aggregatePrivilege">
```

```
    <complexType>
      <sequence>
        <element name="titleref" minOccurs="0" maxOccurs="unbounded"
                 xdb:maxoccurs="unbounded" xdb:translate="true"/>
        <sequence maxOccurs="unbounded">
          <element name="privilegeRef">
            <complexType>
              <attribute name="name" type="QName" use="required"/>
            </complexType>
          </element>
          <any namespace="##other" minOccurs="0" maxOccurs="unbounded"/>
        </sequence>
        <!-- this "any" contains all application specific information
             an aggregate privilege e.g. translations  -->
        <any namespace="##other" minOccurs="0" maxOccurs="unbounded"/>
      </sequence>
      <attribute name="name" type="string" use="required"/>
    </complexType>
  </element>
  <element name="privilege">
    <complexType>
      <sequence minOccurs="0">
        <element name="titleref" minOccurs="0" maxOccurs="unbounded"
                 xdb:maxoccurs="unbounded" xdb:translate="true"/>
        <sequence minOccurs="0" maxOccurs="unbounded">
          <element name="columnRef">
            <complexType>
              <attribute name="schema" type="string" use="required"/>
              <attribute name="table" type="string" use="required"/>
              <attribute name="column" type="string" use="required"/>
            </complexType>
          </element>
          <any namespace="##other" minOccurs="0" maxOccurs="unbounded"/>
        </sequence>
        <!-- this "any" contains all application specific information
             for a privilege e.g. translations  -->
        <any namespace="##other" minOccurs="0" maxOccurs="unbounded"/>
      </sequence>
      <attribute name="name" type="string" use="required"/>
    </complexType>
  </element>
  <element name="titleref">
    <complexType>
      <simpleContent>
        <extension base="xs:string">
          <attribute ref="xml:lang"/>
          <attribute ref="xdb:srclang"/>
          <attribute name="id" type="integer"/>
        </extension>
      </simpleContent>
    </complexType>
  </element>
</schema>
```

Example 7–15 shows a document associated with the XML schema in Example 7–14.

Example 7–15 Translated Document for an XML Schema with Multiple-Valued Elements

```
<securityClass xmlns="http://xmlns.oracle.com/xdb/security.xsd"
               xmlns:is="xmlns.oracle.com/iStore"
```

```
               xmlns:oa="xmlns.oracle.com/OracleApps">
  <name>
securityClassExample
  </name>
  <title>
Security Class Example
  </title>
  <inherits-from>is:iStorePurchaseOrder</inherits-from>
  <privlist>
    <privilege name="privilege1"/>
    <aggregatePrivilege name="iStorePOApprover">
      <title>
iStore Purchase Order Approver
      </title>
      <privilegeRef name="is:privilege1"/>
      <privilegeRef name="oa:submitPO"/>
      <privilegeRef name="oa:privilege3"/>
    </aggregatePrivilege>
    <privilege name="privilege2">
      <title id="2" xml:lang="en" xdb:srclang="true">
secondary privilege - english
      </title>
      <title id="1" xml:lang="fr">
primary privilege - french
      </title>
      <title id="1" xml:lang="en" xdb:srclang="true">
primary privilege - english
      </title>
      <columnRef schema="APPS" table="PurchaseOrder" column="POId"/>
      <columnRef schema="APPS" table="PurchaseOrder" column="Amount"/>
    </privilege>
  </privlist>
</securityClass>
```

Operations on Translated Documents

You can perform the following operations on translated documents:

- Insert: You can insert a document into Oracle XML DB, if it conforms to an XML schema that supports translations. For the document that contains translations, you can either provide the language information or use the session language translation.
  - When the document does not contain the language information and the `xml:lang` attribute is not set, the session language is used for translation. Example 7–16 describes a document with a session language of Japanese. Attribute `xml:lang` is set to `session language`, and attribute `xdb:srclang` is set to `true`.

Example 7–16 Inserting a Document with No Language Information

```
<securityClass xmlns="http://xmlns.oracle.com/xdb/security.xsd"
               xmlns:is="xmlns.oracle.com/iStore"
               xmlns:oa="xmlns.oracle.com/OracleApps"
               targetNamespace="xmlns.oracle.com/example">
  <name>
securityClassExample
  </name>
  <title>
Security Class Example
```

```
  </title>
  <inherits-from>is:iStorePurchaseOrder</inherits-from>
</securityClass>
```

Example 7–17 shows the document after it is inserted into Oracle XML DB Repository.

Example 7–17 Document After Insertion into the Repository

```
<securityClass xmlns="http://xmlns.oracle.com/xdb/security.xsd"
               xmlns:is="xmlns.oracle.com/iStore"
               xmlns:oa="xmlns.oracle.com/OracleApps"
               targetNamespace="xmlns.oracle.com/example">
  <name>
securityClassExample
  </name>
  <title xml:lang="jp" xdb:srclang="true">
Security Class Example
  </title>
  <inherits-from>is:iStorePurchaseOrder</inherits-from>
</securityClass>
```

– When you provide the language information, you set attribute `xml:lang` by either explicitly marking a translation as `xdb:srclang=true` or using the session language translation in attribute `xdb:srclang`. If you do neither, then an arbitrary translation is picked, for which `xdb:srclang` is set to `true`.

  Example 7–18 describes a document with a session language of Japanese.

Example 7–18 Inserting a Document with Language Information

```
<securityClass xmlns="http://xmlns.oracle.com/xdb/security.xsd"
               xmlns:is="xmlns.oracle.com/iStore"
               xmlns:oa="xmlns.oracle.com/OracleApps"
               targetNamespace="xmlns.oracle.com/example">
  <name>
securityClassExample
  </name>
  <title xml:lang="en">
Security Class Example
  </title>
  <title xml:lang="fr">
Security Class Example - FR
  </title>
  <inherits-from>is:iStorePurchaseOrder</inherits-from>
</securityClass>
```

Example 7–19 shows the document after it is inserted into Oracle XML DB Repository.

Example 7–19 Document After Insertion

```
<securityClass xmlns="http://xmlns.oracle.com/xdb/security.xsd"
               xmlns:is="xmlns.oracle.com/iStore"
               xmlns:oa="xmlns.oracle.com/OracleApps"
               targetNamespace="xmlns.oracle.com/example">
  <name>
securityClassExample
  </name>
```

```
  <title xml:lang="en" xdb:srclang="true">
Security Class Example
  </title>
  <title xml:lang="fr">
Security Class Example - FR
  </title>
  <inherits-from>is:iStorePurchaseOrder</inherits-from>
</securityClass>
```

- Query: If you query nodes that involve translated elements, the query displays the translation's default behavior. In order to specify that the translation's default behavior should be applied to the query result, you need to use the Oracle XPath function `ora:translate`. This is the syntax of the function:

  ```
  Nodeset ora:translate(Nodeset parent, String childname, String childnsp)
  ```

  Parameter *`parent`* is the parent node under which you want to search for the translated nodes; *`childname`* is the name of the child node; and *`childnsp`* is the namespace URL of the child node.

  Function `ora:translate` returns a positive integer when the name of the parent node matches the name of the specified child node, and the `xml:lang` value is same as the session language or the language for which `xdb:srclang` is `true`.

  When SQL functions such as `XMLQuery` are applied to translated documents, they return the session language translation, if present, or the source language translation, otherwise. For example, this query returns the session language translation:

  ```
  SELECT XMLQuery('$x/ora:translate(securityClass, "title")'
                  PASSING x.OBJECT_VALUE AS "x" RETURNING CONTENT)
    FROM some_table x;
  ```

  This is the output of that query:

  ```
  <title xml:lang="fr">
  Security Class Example - FR
  </title>
  ```

  To obtain the result in a particular language, specify it in the XPath expression.

  ```
  SELECT XMLQuery('$x/securityClass/title[@xml:lang="en"]'
                  PASSING x.OBJECT_VALUE AS "x" RETURNING CONTENT)
    FROM some_table x;
  ```

  This is the output of that query:

  ```
  <title xml:lang="en" xdb:srclang="true">
  Security Class Example
  </title>
  ```

  Because you can store translated documents only as text (`CLOB`) or binary XML, only functional evaluation and queries with a function-based index, an `XMLIndex` index, or a `CONTEXT` index are possible. For `XMLIndex` index and `CONTEXT` index queries, if the document has a session language translation, then that is returned, otherwise the source language translation is returned. However, for queries with a function-based index, you need to create an index with an explicit `xml:lang` predicate for every language for which you want to use the index.

  When you retrieve the complete document using SQL functions such as `XMLSerialize` and `XDBURIType`, only the translations that match the session

language translations are returned. For protocols, you can set your language preferences, and the document is returned in that language only.

The following PL/SQL procedures and functions support XML translations:

- `DBMS_XMLTRANSLATIONS.translateXML`: Translate a document to the specified language. If the specified language translation is present, it is returned, otherwise, the source language translation is returned.

  For example, if you write `translateXML(doc, 'fr')` to specify French as the translation language for the Example 7–19, it returns the following code and ignores all other translations:

```
securityClass xmlns="http://xmlns.oracle.com/xdb/security.xsd"
              xmlns:is="xmlns.oracle.com/iStore"
              xmlns:oa="xmlns.oracle.com/OracleApps"
              targetNamespace="xmlns.oracle.com/example">
    <name>
        securityClassExample
    </name>
    <title xml:lang="fr">
        Security Class Example - FR
    </title>
    <inherits-from>is:iStorePurchaseOrder</inherits-from>
</securityClass>
```

- `DBMS_XMLTRANSLATIONS.enableTranslation`, `DBMS_XMLTRANSLATIONS.disableTranslation`: Enable or disable translations at the session level. Queries work on the base document if the translation is disabled and on the translated document if it is enabled.
- `DBMS_XMLTRANSLATIONS.getBaseDocument`: Returns the entire document, with all of the translations.

- Update: You can use Oracle SQL function `updateXML` to update the translated nodes. However, an error is raised if you try to update a translated node without specifying the translation language. The following PL/SQL procedures support update operations on translated documents:
  - `DBMS_XMLTRANSLATIONS.updateTranslation`: This function updates the translation at a specified `xpath` in a particular language. If the translation in a particular language is not present, then it is inserted.
  - `DBMS_XMLTRANSLATIONS.setSourceLang`: This procedure sets the source language at a specified xpath to the specified language.

Creating XMLType Tables and Columns Based on XML Schemas

Using Oracle XML DB, you can create `XMLType` tables and columns that are constrained to a global element defined by a registered XML schema. After an `XMLType` column has been constrained to a particular element and a particular XML schema, it can only contain documents that are compliant with the schema definition of that element. You constrain an `XMLType` table column to a particular element and XML schema by adding appropriate `XMLSCHEMA` and `ELEMENT` clauses to the `CREATE TABLE` operation.

Figure 7–1 through Figure 7–4 show the syntax for creating an `XMLType` table.

See Also: *Oracle Database SQL Language Reference* for the complete description of `CREATE TABLE`, including syntax elements such as object_properties.

Note: To create an `XMLType` table in a different database schema from your own, you must have not only privilege `CREATE ANY TABLE` but also privilege `CREATE ANY INDEX`. This is because a unique index is created on column `OBJECT_ID` when you create the table. Column `OBJECT_ID` stores a system-generated object identifier.

Figure 7–1 Creating an XMLType Table – CREATE TABLE

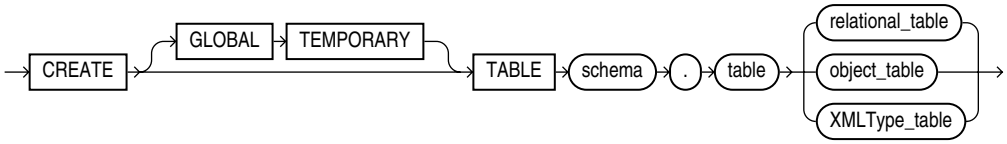


Figure 7–2 Creating an XMLType Table – XMLType_table

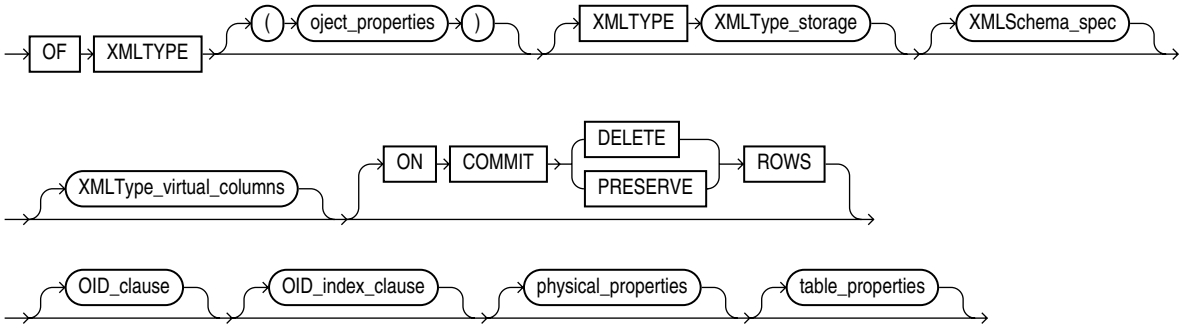


Figure 7–3 Creating an XMLType Table – table_properties

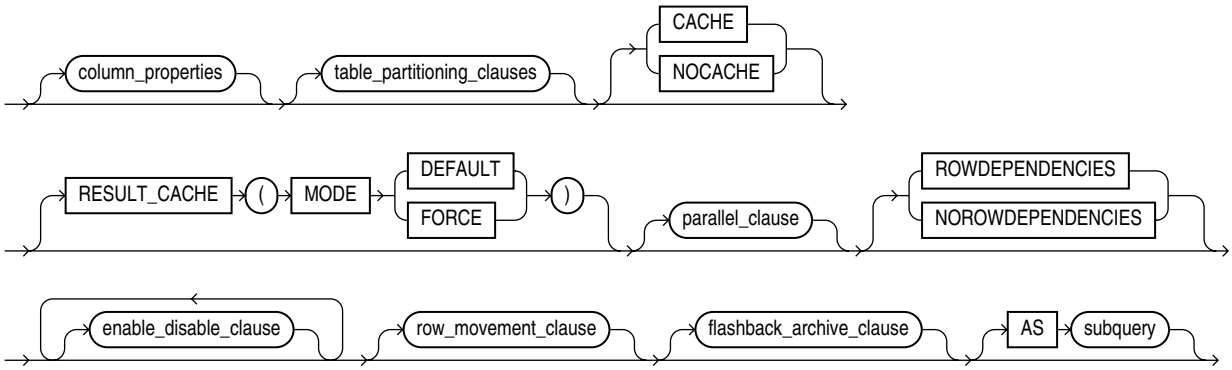
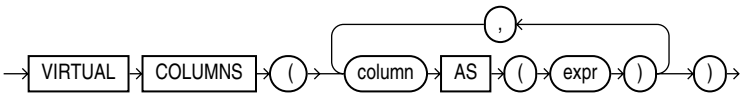


Figure 7–4 Creating an XMLType Table – XMLType_virtual_columns



> **Note:** For XML data, virtual columns are used primarily for partitioning or defining SQL constraints. If your need is to project out specific XML data in order to access it relationally, then consider using SQL/XML function `XMLTable` or `XMLIndex` with a structured component. See also:
>
> - "Partitioning or Constraining Binary XML Data using Virtual Columns" on page 3-3
> - "XMLTABLE SQL/XML Function in Oracle XML DB" on page 5-7
> - "XMLIndex Structured Component" on page 6-10

A subset of the XPointer notation can also be used to provide a single URL that contains the XML schema location and element name. See also Chapter 4, "XMLType Operations".

Example 7–20 shows two `CREATE TABLE` statements. The first creates `XMLType` table `purchaseorder_as_table`. The second creates relational table `purchaseorder_as_column`, which has `XMLType` column `xml_document`. In each table, the `XMLType` instance is constrained to the `PurchaseOrder` element that is defined by the XML schema registered with URL `http://xmlns.oracle.com/xdb/documentation/purchaseOrder.xsd`.

Example 7–20 Creating XML Schema-Based XMLType Tables and Columns

```
CREATE TABLE purchaseorder_as_table OF XMLType
  XMLSCHEMA "http://xmlns.oracle.com/xdb/documentation/purchaseOrder.xsd"
  ELEMENT "PurchaseOrder";

CREATE TABLE purchaseorder_as_column (id NUMBER, xml_document XMLType)
  XMLTYPE COLUMN xml_document
  ELEMENT
    "http://xmlns.oracle.com/xdb/documentation/purchaseOrder.xsd#PurchaseOrder";
```

There are two ways to specify `XMLSchema` and `Element`:

- as separate clauses, `XMLSchema` and `Element`
- using only the `Element` clause with an XPointer notation

The data associated with an `XMLType` table or column that is constrained to an XML schema can be stored in different ways:

- Decomposed and stored object-relationally (structured storage)
- Stored as text, using a single `CLOB` column (unstructured storage)
- Stored as binary XML, using a single binary-XML column (binary XML storage)

Specifying XMLType Storage Options for XML Schema-Based Data

You can specify storage options to use when you manually create a table that stores XML instance documents that reference an XML schema. To specify a particular `XMLType` storage model, use a `STORE AS` clause in the `CREATE TABLE` statement. Otherwise, the storage model specified during registration of the XML schema is used. If no storage model was specified during registration, then object-relational storage is used.

This section describes what you need to know about specifying storage options for XML schema-based data. You can also specify storage options for tables that are created automatically, by using XML schema annotations.

See Also: "Oracle XML Schema Annotations" on page 7-34

Binary XML Storage of XML Schema-Based Data

If you specify `STORE AS BINARY_XML`, then binary XML storage is used. If you specify an XML schema that the XML documents must conform to, then you can use that XML schema only to create `XMLType` tables and columns that are stored as binary XML. You *cannot* use the same XML schema to create `XMLType` tables and columns that are stored object-relationally or as `CLOB` instances.

The converse is also true: If you use a storage model other than binary XML for the registered XML schema, then you can use only that XML schema to create `XMLType` tables and columns that are *not* stored object-relationally or as `CLOB` instances.

Binary XML storage offers a great deal of flexibility for XML data, especially concerning the use of XML schemas. Binary XML encodes XML data differently, depending upon whether or not an XML schema is used for the encoding, and it can encode the same data differently using different XML schemas.

When an XML schema is taken into account for encoding binary XML data, the XML Schema data types are mapped to encoded types for storage. Alternatively, you can encode XML data as non-schema-based binary XML, whether or not the data references an XML schema. In that case, any referenced XML schema is ignored, and there is no encoding of XML Schema data types.

When you create an `XMLType` table or column and you use binary XML storage, you can specify how to encode the column or table to make use of XML schemas. Choose from among these possibilities:

- Encode the column or table data as *non-schema-based* binary XML. The XML data stored in the column can nevertheless conform to an XML schema, but it need not. Any referenced XML schema is ignored for encoding purposes, and documents are not automatically validated when they are inserted or updated.

  You can nevertheless explicitly validate an XML schema-based document that is encoded as non-schema-based binary XML. This represents an important use case: situations where you do not want to tie documents too closely to a particular XML schema, because you might change it or delete it.

- Encode the column or table data to conform to a *single XML schema*. All rows (documents) must conform to the same XML schema. You can nevertheless specify, as an option, that non-schema-based documents can also be stored in the same column.

- Encode the column or table data to conform to whatever XML schema it references Each row (document) can reference *any XML schema*, and that XML schema is used to encode that particular XML document. In this case also, you can specify, as an option, that non-schema-based documents can also be stored in the same column.

  You can use multiple *versions* of the same XML schema in this way. Store documents that conform to different versions. Each is encoded according to the XML schema that it references.

You can specify that any XML schema can be used for encoding by using option `ALLOW ANYSCHEMA` when you create the table.

Note:

- If you use option `ALLOW ANYSCHEMA`, then any XML schema referenced by your instance documents is used *only for validation*. It is *not* used at query time. Queries of your data treat it as if it were non XML schema-based data.
- Oracle recommends that you do *not* use option `ALLOW ANYSCHEMA` if you anticipate using copy-based XML schema evolution (see "Using Copy-Based Schema Evolution" on page 10-2). If you use this option, it is impossible to determine which rows (documents) might conform to the XML schema that is evolved. Conforming rows are not transformed during copy-based evolution, and afterward they are not decodable.

You can specify, for tables and columns that use XML schema-based encodings, that they can accept also non-schema-based documents by using option `ALLOW NONSCHEMA`. In the absence of keyword `XMLSCHEMA`, encoding is for non-schema-based documents. In the absence of the keywords `ALLOW NONSCHEMA` but the presence of keyword `XMLSCHEMA`, encoding is for the single XML schema specified. In the absence of the keywords `ALLOW NONSCHEMA` but the presence of the keywords `ALLOW ANYSCHEMA`, encoding is for any XML schema that is referenced.

An error is raised if you try to insert an XML document into an `XMLType` table or column that does not correspond to the document.

The various possibilities are summarized in Table 7–2.

Table 7–2 CREATE TABLE Encoding Options for Binary XML

Storage Options	Encoding Effect
`STORE AS BINARY XML`	Encodes all documents using the non-schema-based encoding.
`STORE AS BINARY XML XMLSCHEMA ...`	Encodes all documents using an encoding based on the referenced XML schema. Trying to insert or update a document that does not conform to the XML schema raises an error.
`STORE AS BINARY XML XMLSCHEMA ... ALLOW NONSCHEMA`	Encodes all XML schema-based documents using an encoding based on the referenced XML schema. Encodes all non-schema-based documents using the non-schema-based encoding. Trying to insert or update an XML schema-based document that does not conform to the referenced XML schema raises an error.
`STORE AS BINARY XML ALLOW ANYSCHEMA`	Encodes all XML schema-based documents using an encoding based on the XML schema referenced by the document. Trying to insert or update a document that does not reference a registered XML schema or that does not conform to the XML schema it references raises an error.
`STORE AS BINARY XML ALLOW ANYSCHEMA ALLOW NONSCHEMA`	Encodes all XML schema-based documents using an encoding based on the XML schema referenced by the document. Encodes all non-schema-based documents using the non-schema-based encoding. Trying to insert or update an XML schema-based document that does not conform to the registered XML schema it references raises an error.

> **Note:** If you use `CREATE TABLE` with `ALLOW NONSCHEMA` but not `ALLOW ANYSCHEMA`, then all documents, even XML schema-based documents, are encoded using the non-schema-based encoding. If you later use `ALTER TABLE` with `ALLOW ANYSCHEMA` on the same table, this has no effect on the encoding of documents that were stored prior to the `ALTER TABLE` operation—all such documents continue to be encoded using the non-schema-based encoding, regardless of whether they reference an XML schema. Only XML schema-based documents that you insert in the table after the `ALTER TABLE` operation are encoded using XML schema-based encodings.

Unstructured Storage of XML Schema-Based Data

You use `STORE AS CLOB` during table creation to specify unstructured storage. In this case, an entire XML document is stored in a single `CLOB` column.

Example 7–21 shows how to create an `XMLType` table and a table with an `XMLType` column, where the contents of the `XMLType` are constrained to a global element defined by a registered XML schema, and the contents of the `XMLType` are stored using a single `CLOB` column.

Example 7–21 Specifying CLOB Storage for Schema-Based XMLType Tables and Columns

```
CREATE TABLE purchaseorder_as_table OF XMLType
  XMLTYPE STORE AS CLOB
  XMLSCHEMA "http://xmlns.oracle.com/xdb/documentation/purchaseOrder.xsd"
  ELEMENT "PurchaseOrder";

CREATE TABLE purchaseorder_as_column (id NUMBER, xml_document XMLType)
  XMLTYPE COLUMN xml_document
  STORE AS CLOB
  XMLSCHEMA "http://xmlns.oracle.com/xdb/documentation/purchaseOrder.xsd"
  ELEMENT "PurchaseOrder";
```

You can add LOB storage parameters to the `STORE AS CLOB` clause.

Structured Storage of XML Schema-Based Data

With structured storage, collections are mapped into SQL varray values. An XML **collection** is any element that has `maxOccurs` > 1, allowing it to appear multiple times. By default, the entire contents of such a varray is stored as a set of rows in an ordered collection table (OCT).

Example 7–22 illustrates specifying additional storage options. The `LineItem` collection varray is stored as a LOB, not as a table. `USERS` is the tablespace used for storing element `Notes`. The table is compressed for online transaction processing (OLTP).

Example 7–22 Specifying Structured Storage Options for XMLType Tables and Columns

```
CREATE TABLE purchaseorder_as_table
  OF XMLType (UNIQUE ("XMLDATA"."Reference"),
              FOREIGN KEY ("XMLDATA"."User") REFERENCES hr.employees (email))
ELEMENT
  "http://xmlns.oracle.com/xdb/documentation/purchaseOrder.xsd#PurchaseOrder"
  VARRAY "XMLDATA"."LineItems"."LineItem" STORE AS LOB lineitem_lob
  LOB ("XMLDATA"."Notes")
```

```
    STORE AS (TABLESPACE USERS ENABLE STORAGE IN ROW
              STORAGE(INITIAL 4K NEXT 32K))
    COMPRESS FOR OLTP;

CREATE TABLE purchaseorder_as_column (
  id NUMBER,
  xml_document XMLType,
  UNIQUE (xml_document."XMLDATA"."Reference"),
  FOREIGN KEY (xml_document."XMLDATA"."User") REFERENCES hr.employees (email))
  XMLTYPE COLUMN xml_document
  XMLSCHEMA "http://xmlns.oracle.com/xdb/documentation/purchaseOrder.xsd"
  ELEMENT "PurchaseOrder"
  VARRAY xml_document."XMLDATA"."LineItems"."LineItem" STORE AS LOB lineitem_lob
  LOB (xml_document."XMLDATA"."Notes")
    STORE AS (TABLESPACE USERS ENABLE STORAGE IN ROW
              STORAGE(INITIAL 4K NEXT 32K))
    COMPRESS FOR OLTP;
```

Note: In releases prior to Oracle Database 11*g*R2, the default behavior for `CREATE TABLE` was to store a collection using a varray stored as a LOB, not a varray stored as a table.

Note: When compression is specified for a parent `XMLType` table or column, all descendant `XMLType` ordered collection tables (OCTs) are similarly compressed.

See Also:

- "Oracle XML Schema Annotations" on page 7-34 for information about specifying storage options by using XML schema annotations
- *Oracle Database SQL Language Reference* for information about compression for OLTP

As a convenience, if you need to specify that *all* varrays in an `XMLType` table or column are to be stored as LOBs, or all are to be stored as tables, then you can use the syntax clause `STORE ALL VARRAYS AS`, followed by `LOBS` or `TABLES`, respectively. This is a convenient alternative to using multiple `VARRAY...STORE AS` clauses, one for each collection. Example 7–23 illustrates this.

Example 7–23 Using STORE ALL VARRAYS AS

```
CREATE TABLE purchaseorder_as_table OF XMLType (UNIQUE ("XMLDATA"."Reference"),
  FOREIGN KEY ("XMLDATA"."User") REFERENCES hr.employees (email))
  ELEMENT
    "http://xmlns.oracle.com/xdb/documentation/purchaseOrder.xsd#PurchaseOrder"
  STORE ALL VARRAYS AS LOBS;
```

The storage method specified using `STORE ALL VARRAYS AS` overrides any storage method specified using `xdb:storeVarrayAsTable` in the corresponding XML schema.

See Also:

- "Controlling How Collections Are Stored for Object-Relational XMLType Storage" on page 3-19 for information about collection storage using default tables
- *Oracle Database SQL Language Reference* for information about using `STORE ALL VARRAYS AS LOBS`

Specifying Relational Constraints on XMLType Tables and Columns

When you store XML data using structured storage, typical relational constraints can be specified for elements and attributes that occur only once in an XML document. Example 7–22 shows how to use object-relational notation to define a unique constraint and a foreign key constraint when creating the table.

It is not possible to define constraints for `XMLType` tables and columns that make use of unstructured storage.

See Also:

- "Partitioning or Constraining Binary XML Data using Virtual Columns" on page 3-3 for how to define constraints on XML data stored as binary XML
- "Adding Unique Constraints to the Parent Element of an Attribute" on page 9-3

Oracle XML Schema Annotations

You can annotate XML schemas to influence the objects and tables that are generated by the XML schema registration process. You do this by adding Oracle-specific attributes to `complexType`, `element`, and `attribute` definitions that are declared by the XML schema.

Most XML attributes used by Oracle XML DB belong to the namespace `http://xmlns.oracle.com/xdb`. XML attributes used for encoding XML data as binary XML belong to the namespace `http://xmlns.oracle.com/2004/CSX`. To simplify the process of annotating an XML schema, Oracle recommends that you declare namespace prefixes in the root element of the XML schema.

Common Uses of XML Schema Annotations

Common reasons for wanting to annotate an XML schema include the following:

- To ensure that the names of the tables, objects, and object attributes created by PL/SQL procedure `DBMS_XMLSCHEMA.registerSchema` for structured storage of XML data are easy to recognize and compliant with any application-naming standards. Set parameter `GENTYPES` or `GENTABLES` to `TRUE` for this.
- To map between the XML schema and existing objects and tables within the database. Set parameter `GENTYPES` or `GENTABLES` to `FALSE` for this.
- To prevent the generation of mixed-case names that require the use of quoted identifiers when working directly with SQL.
- To allow XPath rewrite for structured storage in the case of document-correlated recursive XPath queries. This applies to certain applications of SQL/XML access and query functions whose XQuery-expression argument targets recursive XML data.

The most commonly used XML schema annotations are the following:

- `xdb:defaultTable` – Name of the default table generated for each global element when parameter `GENTABLES` is `FALSE`. Setting this to the empty string, **`""`**, prevents a default table from being generated for the element in question.
- `xdb:SQLName` – Name of the SQL object attribute that corresponds to each element or attribute defined in the XML schema.
- `xdb:SQLType` – For `complexType` definitions, the corresponding object type. For `simpleType` definitions, `SQLType` is used to override the default mapping between XML schema data types and SQL data types. A common use of `SQLType` is to define when unbounded strings should be stored as `CLOB` values, rather than as `VARCHAR(4000) CHAR` values (the default). Note: You cannot use data type `NCHAR`, `NVARCHAR`, or `NCLOB` as the value of a `SQLType` annotation.
- `xdb:SQLCollType` – Used to specify the varray type that manages a collection of elements.
- `xdb:maintainDOM` – Used to determine whether or not DOM fidelity should be maintained for a given `complexType` definition
- `xdb:storeVarrayAsTable` – Specified in the root element of the XML schema. Used to force all collections to be stored as ordered collection tables (OCTs). An OCT is created for each element that is specified with `maxOccurs > 1`. The OCTs are created with system-generated names. The default value of `storeVarrayAsTable` is `true`.

You need not specify values for any of these attributes. Oracle XML DB provides appropriate values by default during the XML schema registration process. However, if you are using structured storage, then Oracle recommends that you specify the names of at least the top-level SQL types, so that you can reference them later.

XML Schema Annotation Example

Example 7–24 shows a partial listing of the XML schema in Example 7–1, modified to include some of the most important Oracle XML DB annotations.

Example 7–24 Using Common Schema Annotations

```
<xs:schema
  targetNamespace="http://xmlns.oracle.com/xdb/documentation/purchaseOrder"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xdb="http://xmlns.oracle.com/xdb"
  xmlns:po="http://xmlns.oracle.com/xdb/documentation/purchaseOrder"
  version="1.0"
  xdb:storeVarrayAsTable="true">
  <xs:element name="PurchaseOrder" type="po:PurchaseOrderType"
              xdb:defaultTable="PURCHASEORDER"/>
  <xs:complexType name="PurchaseOrderType" xdb:SQLType="PURCHASEORDER_T">
    <xs:sequence>
      <xs:element name="Reference" type="po:ReferenceType" minOccurs="1"
                  xdb:SQLName="REFERENCE"/>
      <xs:element name="Actions" type="po:ActionsType"
                  xdb:SQLName="ACTION_COLLECTION"/>
      <xs:element name="Reject" type="po:RejectionType" minOccurs="0"/>
      <xs:element name="Requestor" type="po:RequestorType"/>
      <xs:element name="User" type="po:UserType" minOccurs="1"
                  xdb:SQLName="EMAIL"/>
      <xs:element name="CostCenter" type="po:CostCenterType"/>
      <xs:element name="ShippingInstructions"
```

```
                  type="po:ShippingInstructionsType"/>
      <xs:element name="SpecialInstructions" type="po:SpecialInstructionsType"/>
      <xs:element name="LineItems" type="po:LineItemsType"
                  xdb:SQLName="LINEITEM_COLLECTION"/>
      <xs:element name="Notes" type="po:NotesType" xdb:SQLType="CLOB"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="LineItemsType" xdb:SQLType="LINEITEMS_T">
    <xs:sequence>
      <xs:element name="LineItem" type="po:LineItemType" maxOccurs="unbounded"
                  xdb:SQLCollType="LINEITEM_V" xdb:SQLName="LINEITEM_VARRAY"/>
   </xs:sequence>
  </xs:complexType>
  <xs:complexType name="LineItemType" xdb:SQLType="LINEITEM_T">
    <xs:sequence>
      <xs:element name="Description" type="po:DescriptionType"/>
      <xs:element name="Part" type="po:PartType"/>
    </xs:sequence>
    <xs:attribute name="ItemNumber" type="xs:integer"/>
  </xs:complexType>
  <xs:complexType name="PartType" xdb:SQLType="PART_T" xdb:maintainDOM="false">
    <xs:attribute name="Id">
      <xs:simpleType>
        <xs:restriction base="xs:string">
          <xs:minLength value="10"/>
          <xs:maxLength value="14"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:attribute>
    <xs:attribute name="Quantity" type="po:moneyType"/>
    <xs:attribute name="UnitPrice" type="po:quantityType"/>
  </xs:complexType>
</xs:schema>
```

The `schema` element includes the declaration of the `xdb` namespace. It also includes the annotation `xdb:storeVarrayAsTable = "true"` (which is the default value). This causes all collections within the XML schema to be managed using ordered collection tables (OCTs).

The definition of global element `PurchaseOrder` includes a `defaultTable` annotation that specifies that the name of the default table associated with this element is `purchaseorder`.

The definition of global complex type `PurchaseOrderType` includes a `SQLType` annotation that specifies that the generated SQL object type is named `purchaseorder_t`. Within the definition of this type, the following annotations are used:

- The definition of element `Reference` includes a `SQLName` annotation that specifies that the SQL attribute corresponding to XML element `Reference` is named `reference`.
- The definition of element `Actions` includes a `SQLName` annotation that specifies that the SQL attribute corresponding to XML element `Actions` is named `action_collection`.
- The definition of element `USER` includes a `SQLName` annotation that specifies that the SQL attribute corresponding to XML element `User` is named `email`.

- The definition of element `LineItems` includes a `SQLName` annotation that specifies that the SQL attribute corresponding to XML element `LineItems` is named `lineitem_collection`.
- The definition of element `Notes` includes a `SQLType` annotation that specifies that the data type of the SQL attribute corresponding to XML element `Notes` is `CLOB`.

The definition of global complex type `LineItemsType` includes a `SQLType` annotation that specifies that the generated SQL object type is named `lineitems_t`. Within the definition of this type, the following annotations are used:

- The definition of element `LineItem` includes a `SQLName` annotation that specifies that the data type of the SQL attribute corresponding to XML element `LineItems` is named `lineitem_varray`, and a `SQLCollName` annotation that specifies that the SQL object type that manages the collection is named `lineitem_v`.

The definition of global complex type `LineItemType` includes a `SQLType` annotation that specifies that generated SQL object type is named `lineitem_t`.

The definition of complex type `PartType` includes a `SQLType` annotation that specifies that the SQL object type is named `part_t`. It also includes the annotation `xdb:maintainDOM = "false"`, specifying that there is no need for Oracle XML DB to maintain DOM fidelity for elements based on this data type.

Example 7–25 shows some of the tables and objects that are created when the annotated XML schema is registered.

Example 7–25 Registering an Annotated XML Schema

```
BEGIN
  DBMS_XMLSCHEMA.registerSchema(
    SCHEMAURL => 'http://xmlns.oracle.com/xdb/documentation/purchaseOrder.xsd',
    SCHEMADOC => bfilename('XMLDIR', 'purchaseOrder.Annotated.xsd'),
    LOCAL     => TRUE,
    GENTYPES  => TRUE,
    GENTABLES => TRUE,
    CSID      => nls_charset_id('AL32UTF8'));
END;
/

SELECT table_name, xmlschema, element_name FROM USER_XML_TABLES;

TABLE_NAME     XMLSCHEMA                              ELEMENT_NAME
-------------  -----------------------------------    -------------
PURCHASEORDER  http://xmlns.oracle.com/xdb/documen    PurchaseOrder
               tation/purchaseOrder.xsd

1 row selected.

DESCRIBE purchaseorder

Name                            Null? Type
-----------------------------   ----- -----------------
TABLE of SYS.XMLTYPE(XMLSchema
 "http://xmlns.oracle.com/xdb/documentation/purchaseOrder.xsd"
 ELEMENT "PurchaseOrder") STORAGE Object-relational TYPE "PURCHASEORDER_T"

DESCRIBE purchaseorder_t

PURCHASEORDER_T is NOT FINAL
Name                 Null? Type
```

```
-------------------- ----- --------------------------
SYS_XDBPD$                 XDB.XDB$RAW_LIST_T
REFERENCE                  VARCHAR2(30 CHAR)
ACTION_COLLECTION          ACTIONS_T
REJECT                     REJECTION_T
REQUESTOR                  VARCHAR2(128 CHAR)
EMAIL                      VARCHAR2(10 CHAR)
COSTCENTER                 VARCHAR2(4 CHAR)
SHIPPINGINSTRUCTIONS       SHIPPING_INSTRUCTIONS_T
SPECIALINSTRUCTIONS        VARCHAR2(2048 CHAR)
LINEITEM_COLLECTION        LINEITEMS_T
Notes                      CLOB

DESCRIBE lineitems_t
LINEITEMS_T is NOT FINAL
Name                 Null? Type
-------------------- ----- --------------------------
SYS_XDBPD$                 XDB.XDB$RAW_LIST_T
LINEITEM_VARRAY            LINEITEM_V

DESCRIBE lineitem_v

LINEITEM_V VARRAY(2147483647) OF LINEITEM_T
LINEITEM_T is NOT FINAL
Name                 Null? Type
-------------------- ----- --------------------------
SYS_XDBPD$                 XDB.XDB$RAW_LIST_T
ITEMNUMBER                 NUMBER(38)
DESCRIPTION                VARCHAR2(256 CHAR)
PART                       PART_T

DESCRIBE part_t

PART_T is NOT FINAL
Name                 Null? Type
-------------------- ----- --------------------------
ID                         VARCHAR2(14 CHAR)
QUANTITY                   NUMBER(12,2)
UNITPRICE                  NUMBER(8,4)

SELECT table_name, parent_table_column FROM USER_NESTED_TABLES
  WHERE parent_table_name = 'purchaseorder';

TABLE_NAME                       PARENT_TABLE_COLUMN
----------                       -----------------------
SYS_NTNOHV+tfSTRaDTA9FETvBJw==   "XMLDATA"."LINEITEM_COLLECTION"."LINEITEM_VARRAY"
SYS_NTV4bNVqQ1S4WdCIvBK5qjZA==   "XMLDATA"."ACTION_COLLECTION"."ACTION_VARRAY"

2 rows selected.
```

The following are results of this XML schema registration:

- A table called `purchaseorder` was created.
- Types called `purchaseorder_t`, `lineitems_t`, `lineitem_v`, `lineitem_t`, and `part_t` were created. The attributes defined by these types are named according to supplied the `SQLName` annotations.
- The `Notes` attribute defined by `purchaseorder_t` is of data type `CLOB`.
- Type `part_t` does not include a positional descriptor (PD) attribute.

- Ordered collection tables (OCTs) were created to manage the collections of `LineItem` and `Action` elements.

Available Oracle XML DB XML Schema Annotations

Table 7–3, Table 7–4, and Table 7–5 list Oracle XML DB annotations that you can specify in element and attribute declarations. All annotations except those that have the prefix `csx` are applicable to XML schemas registered for structured storage. This includes the portions of hybrid storage that are stored object-relationally.

The following annotations apply to XML schemas that are registered for unstructured storage:

- `xdb:defaultTable`
- `xdb:defaultTableSchema`

The following annotations apply to XML schemas that are registered for binary XML storage:

- `xdb:defaultTable`
- `xdb:defaultTableSchema`
- `xdb:tableProps`
- `csx:encodingType`

Table 7–3 Annotations in Elements

Attribute	Values	Default	Description
`xdb:columnProps`	Any column storage clause	`NULL`	Specifies the `COLUMN` storage clause that is inserted into the default `CREATE TABLE` statement. It is useful mainly for elements that get mapped to SQL tables, namely top-level element declarations and out-of-line element declarations.
`xdb:defaultTable`	Any table name	Based on element name	Specifies the name of the SQL table into which XML instances of this XML schema are stored. This is most useful in cases where the XML data is inserted from APIs and protocols, such as FTP and HTTP(S), where the table name is not specified. Applicable to structured storage and binary XML storage.
`xdb:defaultTableSchema`	Any SQL user name	User registering XML schema	Name of the database user (database schema) who owns the type specified by `xdb:defaultTable`. Applicable to structured storage and binary XML storage.
`csx:encodingType`	Any binary XML encoding type[1]	Based on XML Schema data type	Specifies the encoding type to be used to encode the node value of this element or attribute. This can be used only within elements `xsd:attribute`, `xsd:simpleType`, and `xsd:element` (for elements based on `simpleType` or `simpleContent` only). Applicable only to binary XML storage.
`xdb:maintainDOM`	`true` | `false`	`true`	If `true`, then instances of this element are stored so that they retain DOM fidelity on output. This implies that all comments, processing instructions, namespace declarations, and so on are retained, in addition to the ordering of elements. If `false`, then the output is not guaranteed to have the same DOM action as the input.

Table 7–3 (Cont.) Annotations in Elements

Attribute	Values	Default	Description
`xdb:maintainOrder`	`true` | `false`	`true`	If `true` (generally recommended, and the default value), then the collection is mapped to a varray (stored in a LOB or an ordered collection table). If `false`, then the collection is mapped to an unordered table, and document order is *not* preserved.
`xdb:maxOccurs`	Any positive integer	1	Specifies the maximum number of times an element can appear. If the value is `unbounded`, then there is no limit to the maximum number of occurrences.
`xdb:SQLCollSchema`	Any SQL user name	User registering XML schema	Name of the database user (database schema) who owns the type specified by `xdb:SQLCollType`.
`xdb:SQLCollType`	Any SQL collection type	Name generated from element name	Name of the SQL collection type that corresponds to this XML element. The XML element must be specified with `maxOccurs` > 1.
`xdb:SQLInline`	`true` | `false`	`true`	If `true`, then this element is stored inline as an embedded object attribute (or as a collection, if `maxOccurs` > 1). If `false`, then a `REF` value is stored (or a collection of `REF` values, if `maxOccurs` > 1). This attribute is forced to `false` in certain situations, such as cyclic references, where SQL does not support inlining.
`xdb:SQLName`	Any SQL identifier	Element name	Name of the attribute within the SQL object that maps to this XML element.
`xdb:SQLSchema`	Any SQL user name	User registering XML schema	Name of the database user (database schema) who owns the type specified by `SQLType`.
`xdb:SQLType`	Any SQL data type[2], *except* `NCHAR`, `NVARCHAR`, and `NCLOB`	Name generated from element name	Name of the SQL type corresponding to this XML element declaration.
`xdb:srclang`	`true` | `false`	true	If `true`, then the given language translation is used as the default translation.
`xdb:tableProps`	Any table storage clause	`NULL`	Specifies the `TABLE` storage clause that is appended to the default `CREATE TABLE` statement. This is meaningful mainly for global and out-of-line elements. Applicable to structured storage and binary XML storage.
`xdb:translate`	`true` | `false`	`true`	If `true`, then instances of this element are translated. The `maxOccurs` attribute must be <=1 for this element to be set to `true`.

[1] See "Mapping XML Schema Data Types To Binary XML Encoding Types" on page 7-53.
[2] See "Mapping XML Schema Data Types to SQL Data Types" on page 7-45.

See Also: "Structured Storage of XML Schema-Based Data" on page 7-32 for information about specifying storage options when manually creating `XMLType` tables for object-relational storage

Table 7–4 Annotations in Elements Declaring Global complexType Elements

Attribute	Values	Default	Description
csx:encodingType	Any binary XML encoding type[1]	Based on XML Schema data type	The encoding type to be used to encode the node value of this element or attribute. This can be used only within elements xsd:attribute, xsd:simpleType, and xsd:element (for elements based on simpleType or simpleContent only).
xdb:maintainDOM	true | false	true	If true, then instances of this element are stored so that they retain DOM fidelity on output. This implies that all comments, processing instructions, namespace declarations, and so on are retained, in addition to the ordering of elements. If false, then the output is not guaranteed to have the same DOM action as the input.
xdb:SQLSchema	Any SQL user name	User registering XML schema	Name of the database user (database schema) who owns the type specified by SQLType.
xdb:SQLType	Any SQL data type[2] *except* NCHAR, NVARCHAR, and NCLOB	Name generated from element name	Name of the SQL type that corresponds to this XML element declaration.

[1] See "Mapping XML Schema Data Types To Binary XML Encoding Types" on page 7-53.
[2] See "Mapping XML Schema Data Types to SQL Data Types" on page 7-45.

Table 7–5 Annotations in XML Schema Declarations

Attribute	Values	Default	Description
xdb:mapUnboundedStringToLob	true | false	false	If true, then unbounded strings are mapped to CLOB instances by default. Similarly, unbounded binary data gets mapped to a BLOB value, by default. If false, then unbounded strings are mapped to VARCHAR2(4000) values, and unbounded binary components are mapped to RAW(2000) values.
xdb:storeVarrayAsTable	true | false	true	If true, then the varray is stored as a table (OCT). If false, then the varray is stored in a LOB.

See Also: "Changing an XML Schema and XML Instance Documents for Translation" on page 17 for more information on xdb:maxOccurs, xdb:translate, and xdb:srclang.

XML Schema Annotation Guidelines for Structured Storage

For XMLType data stored object-relationally (structured storage), careful planning is called for, to optimize performance. Similar considerations are in order as for ordinary relational data: the entity-relationship model, indexing, data types, table partitions, and so on.

To enable XPath rewrite and achieve optimal performance, you implement many such design choices using XML schema annotations. This section provides annotation guidelines to optimize the use of `XMLType` data stored object-relationally.

See Also:

- Table 7–3, " Annotations in Elements" on page 7-39
- Chapter 8, "XPath Rewrite for Structured Storage"

Avoid Creation of Unnecessary Tables for Unused Top-Level Elements

By default, XML schema registration creates a top-level table for each top-level element defined in the XML schema. Some such elements are used as top-level elements in XML instances that conform to the XML schema. Others might not. It is common, for example, for elements in an XML schema to be top-level in order to be used as a `REF` target.

Whenever a top-level element in an XML schema is never used at the top level in any corresponding XML instance, you can avoid the creation of the associated unnecessary tables by adding annotation `xdb:defaultTable = ""` to the element in the XML schema. An empty value for this attribute prevents default-table creation.

Provide Your Own Names for Default Tables

For tuning purposes, you examine execution plan output for queries you are interested in. This output refers to the tables that underlie `XMLType` data stored object-relationally. By default, these tables have system-generated names. Oracle recommends that you provide your own table names, especially for tables that you are sure to be interested in. You do that using annotation `xdb:defaultTable`.

See Also: "Default Tables Created During XML Schema Registration" on page 7-10

Turn Off DOM Fidelity If Not Needed

By default, XML schema registration generates tables that store XML data in such a way that DOM fidelity is maintained. It is often the case that for data-centric XML data DOM fidelity is not needed. You can improve the performance of storage, queries, and data modification by instead using object-relational tables that do not maintain DOM fidelity. You use the annotation `xdb:maintainDOM = "false"` to do that.

See Also: "DOM Fidelity" on page 7-16

Use Unordered Collection Elements When Order Doesn't Matter

If the order among collection elements is not important, then use annotation `xdb:maintainOrder = "false"`. This can allow more optimization in XPath rewrite and can generally lead to more efficient query execution.

Annotate Time-Related Elements with a Timestamp Data Type

If your application needs to work with time-zone indicators, then annotate any XML schema elements of type `xs:time` and `xs:dateTime` with `xdb:SQLType = "TIMESTAMP WITH TIME ZONE"`. This ensures that values containing time-zone indicators can be stored, retrieved, and compared.

Add Table and Column Properties

If a table or column underlying XML data needs additional properties, such as partition, tablespace, or compression clauses, then use annotation `xdb:tableProps` or `xdb:columnProps` to provide them. This lets users add primary keys or constraints. For example, to achieve table compression for online transaction processing (OLTP), you would add `COMPRESS FOR OLTP` using a `tableProps` attribute.

> **See Also:** Example 7–22 on page 7-32 for an example of specifying OLTP compression when creating `XMLType` tables and columns manually

Store Large Collections Out of Line

When the total number of elements and attributes defined by a `complexType` reaches 1000, it is not possible to create a single table that can manage the SQL objects that are generated when an instance of that type is stored in the database. If you have large collections, then you might run up against this limit of 1000 columns for a table.

You can use annotations `xdb:defaultTable` and `xdb:SQLInline` to specify that such collection elements be stored out of line. That means that their data is stored in a separate table—only a reference to a row in that table is stored in the main collection table. Use `xdb:defaultTable` to name the out-of -line table. Annotate each element of a potentially large collection with `xdb:SQLInline = "false"`, to store it out of line.

> **See Also:**
>
> - "Working with Large XML Schemas" on page 3-26
> - "Setting Annotation Attribute SQLInline to false for Out-Of-Line Storage" on page 9-4

Querying a Registered XML Schema to Obtain Annotations

The registered version of an XML schema contains a full set of Oracle XML DB annotations. As shown in Example 7–8 and Example 7–9, the location of the registered XML schema depends on whether it is local or global.

A registered XML schema can be queried for the annotations that were supplied by the user or added by the schema registration process. Example 7–26 shows the set of global `complexType` definitions declared by an XML schema for structured storage of XML data, and the corresponding SQL object types and DOM fidelity values.

Example 7–26 Querying Metadata from a Registered XML Schema

```
SELECT ct.xmlschema_type_name, ct.sql_type_name, ct.dom_fidelity
  FROM RESOURCE_VIEW,
       XMLTable(
         XMLNAMESPACES(
           'http://xmlns.oracle.com/xdb/XDBResource.xsd' AS "r",
           'http://xmlns.oracle.com/xdb/documentation/purchaseOrder' AS "po",
           'http://www.w3.org/2001/XMLSchema' AS "xs",
           'http://xmlns.oracle.com/xdb' AS "xdb"),
         '/r:Resource/r:Contents/xs:schema/xs:complexType' PASSING RES
         COLUMNS
           xmlschema_type_name VARCHAR2(30) PATH '@name',
           sql_type_name       VARCHAR2(30) PATH '@xdb:SQLType',
           dom_fidelity        VARCHAR2(6)  PATH '@xdb:maintainDOM') ct
```

```
  WHERE
    equals_path(
      RES,
      '/sys/schemas/SCOTT/xmlns.oracle.com/xdb/documentation/purchaseOrder.xsd')
    =1;

XMLSCHEMA_TYPE_NAME        SQL_TYPE_NAME            DOM_FIDELITY
-------------------------  -----------------------  ------------
PurchaseOrderType          PURCHASEORDER_T          true
LineItemsType              LINEITEMS_T              true
LineItemType               LINEITEM_T               true
PartType                   PART_T                   true
ActionsType                ACTIONS_T                true
RejectionType              REJECTION_T              true
ShippingInstructionsType   SHIPPING_INSTRUCTIONS_T  true

7 rows selected.
```

Mapping XML Schema Data Types to Oracle XML DB Storage

XML data that conforms to an XML schema is typed using XML Schema data types. When this XML data is stored in Oracle XML DB, its storage data types are derived from the XML Schema data types using a default mapping and, optionally, using mapping information that you specify using XML schema annotations.

Whenever you do not specify a data type to use for storage, Oracle XML DB uses the default mapping to annotate the XML schema appropriately, during registration. In this way, the registered XML schema has a complete set of data-type annotations.

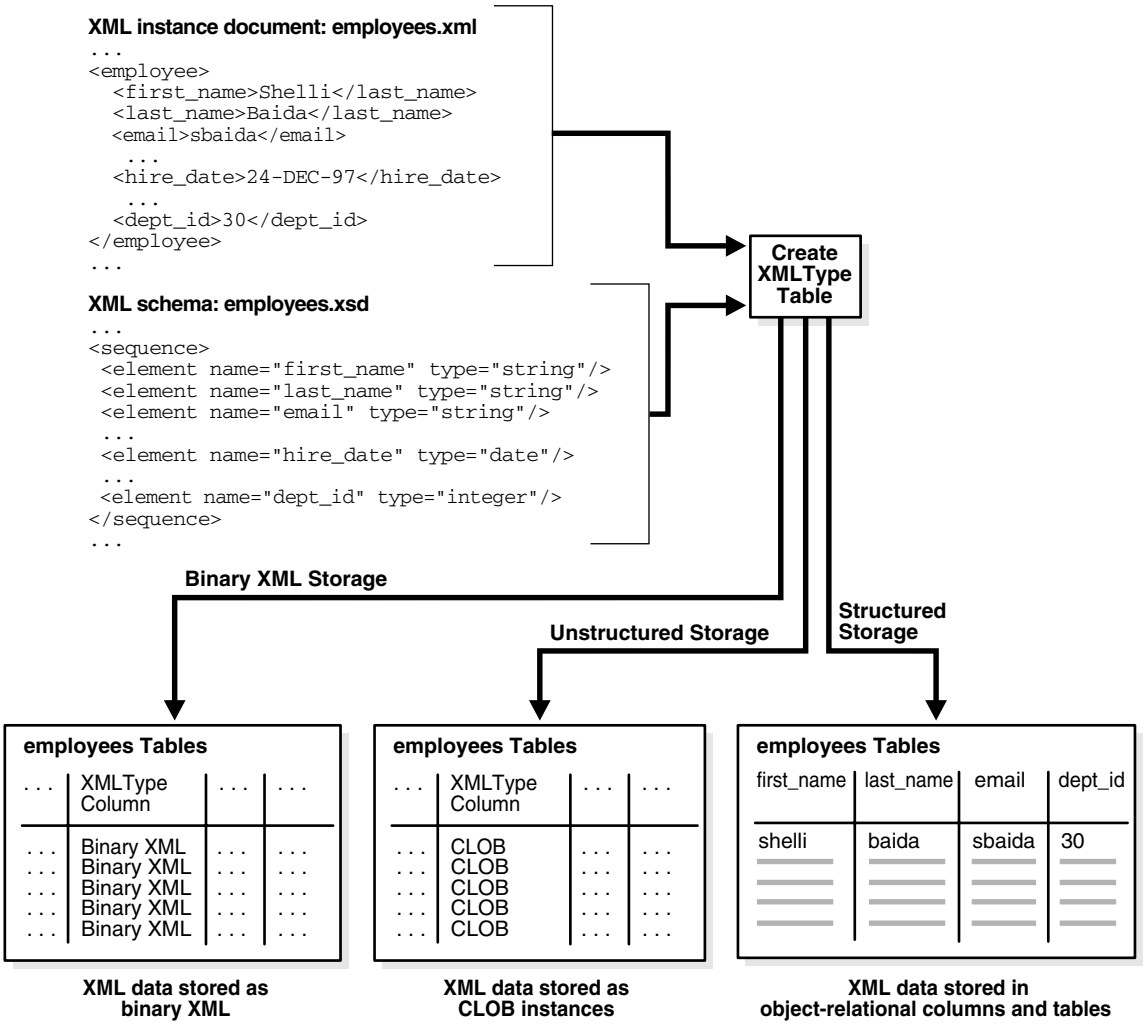
- For unstructured storage, the data-type mapping is trivial: all of the XML data is stored together as a single `CLOB`.
- For structured storage, XML Schema data types are mapped to SQL data types.
- For binary XML storage, XML Schema data types are mapped to Oracle XML DB binary XML encoding types.

See Also:

- "Mapping XML Schema Data Types to SQL Data Types" on page 7-45
- "Mapping XML Schema Data Types To Binary XML Encoding Types" on page 7-53

Figure 7–5 shows how Oracle XML DB creates XML schema-based `XMLType` tables using an XML document and a mapping specified in an XML schema. Depending on the storage method specified in the XML schema, an XML instance document is stored either as a binary XML or `CLOB` value in a single `XMLType` column, or using multiple object-relational columns.

Figure 7–5 How Oracle XML DB Maps XML Schema-Based XMLType Tables



Mapping XML Schema Data Types to SQL Data Types

This section describes how to use PL/SQL package `DBMS_XMLSCHEMA` to map data types for XML Schema attributes and elements to SQL data types.

Note: Do *not* directly access the SQL data types that are mapped from XML Schema data types during XML schema registration. These SQL types are part of the implementation of Oracle XML DB. They are not exposed for your use.

Oracle reserves the right to change the implementation at any time, including in a product patch. Such a change by Oracle will have no effect on applications that abide by the XML abstraction, but it might impact applications that directly access these data types.

Example of Mapping XML Schema Data Types to SQL

Example 7–27 shows a simple example of mapping XML Schema data types to SQL data types. It uses attribute `SQLType` to specify the data-type mapping. It also uses

attribute `SQLName` to specify the object attributes to use for various XML elements and attributes.

Example 7–27 Mapping XML Schema Data Types to SQL Data Types using Attribute SQLType

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:xdb="http://xmlns.oracle.com/xdb"
           version="1.0"
           xdb:storeVarrayAsTable="true">
  <xs:element name="PurchaseOrder" type="PurchaseOrderType" xdb:defaultTable="PURCHASEORDER"/>
  <xs:complexType name="PurchaseOrderType" xdb:SQLType="PURCHASEORDER_T">
    <xs:sequence>
      <xs:element name="Reference" type="ReferenceType" minOccurs="1" xdb:SQLName="REFERENCE"/>
      <xs:element name="Actions" type="ActionsType" xdb:SQLName="ACTIONS"/>
      <xs:element name="Reject" type="RejectionType" minOccurs="0" xdb:SQLName="REJECTION"/>
      <xs:element name="Requestor" type="RequestorType" xdb:SQLName="REQUESTOR"/>
      <xs:element name="User" type="UserType" minOccurs="1" xdb:SQLName="USERID"/>
      <xs:element name="CostCenter" type="CostCenterType" xdb:SQLName="COST_CENTER"/>
      <xs:element name="ShippingInstructions" type="ShippingInstructionsType"
                  xdb:SQLName="SHIPPING_INSTRUCTIONS"/>
      <xs:element name="SpecialInstructions" type="SpecialInstructionsType"
                  xdb:SQLName="SPECIAL_INSTRUCTIONS"/>
      <xs:element name="LineItems" type="LineItemsType" xdb:SQLName="LINEITEMS"/>
      <xs:element name="Notes" type="po:NotesType" xdb:SQLType="CLOB"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="LineItemsType" xdb:SQLType="LINEITEMS_T">
    <xs:sequence>
      <xs:element name="LineItem" type="LineItemType" maxOccurs="unbounded"
                  xdb:SQLName="LINEITEM" xdb:SQLCollType="LINEITEM_V"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="LineItemType" xdb:SQLType="LINEITEM_T">
    <xs:sequence>
      <xs:element name="Description" type="DescriptionType"
                  xdb:SQLName="DESCRIPTION"/>
      <xs:element name="Part" type="PartType" xdb:SQLName="PART"/>
    </xs:sequence>
    <xs:attribute name="ItemNumber" type="xs:integer" xdb:SQLName="ITEMNUMBER"
                  xdb:SQLType="NUMBER"/>
  </xs:complexType>
  <xs:complexType name="PartType" xdb:SQLType="PART_T">
    <xs:attribute name="Id" xdb:SQLName="PART_NUMBER" xdb:SQLType="VARCHAR2">
      <xs:simpleType>
        <xs:restriction base="xs:string">
          <xs:minLength value="10"/>
          <xs:maxLength value="14"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:attribute>
    <xs:attribute name="Quantity" type="moneyType" xdb:SQLName="QUANTITY"/>
    <xs:attribute name="UnitPrice" type="quantityType" xdb:SQLName="UNITPRICE"/>
  </xs:complexType>

  ...

  <xs:complexType name="ActionsType" xdb:SQLType="ACTIONS_T">
    <xs:sequence>
      <xs:element name="Action" maxOccurs="4" xdb:SQLName="ACTION" xdb:SQLCollType="ACTION_V">
        <xs:complexType xdb:SQLType="ACTION_T">
          <xs:sequence>
            <xs:element name="User" type="UserType" xdb:SQLName="ACTIONED_BY"/>
            <xs:element name="Date" type="DateType" minOccurs="0" xdb:SQLName="DATE_ACTIONED"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
```

```
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="RejectionType" xdb:SQLType="REJECTION_T">
    <xs:all>
      <xs:element name="User" type="UserType" minOccurs="0" xdb:SQLName="REJECTED_BY"/>
      <xs:element name="Date" type="DateType" minOccurs="0" xdb:SQLName="DATE_REJECTED"/>
      <xs:element name="Comments" type="CommentsType" minOccurs="0" xdb:SQLName="REASON_REJECTED"/>
    </xs:all>
  </xs:complexType>
  <xs:complexType name="ShippingInstructionsType" xdb:SQLType="SHIPPING_INSTRUCTIONS_T">
    <xs:sequence>
      <xs:element name="name" type="NameType" minOccurs="0" xdb:SQLName="SHIP_TO_NAME"/>
      <xs:element name="address" type="AddressType" minOccurs="0" xdb:SQLName="SHIP_TO_ADDRESS"/>
      <xs:element name="telephone" type="TelephoneType" minOccurs="0" xdb:SQLName="SHIP_TO_PHONE"/>
    </xs:sequence>
  </xs:complexType>
  ...
</xs:schema>
```

Mapping XML Schema Attribute Data Types to SQL

An attribute declaration can specify its XML Schema data type in terms of one of the following:

- Primitive type
- Global `simpleType`, declared within this XML schema or in an external XML schema
- Reference to global attribute (`ref=". ."`), declared within this XML schema or in an external XML schema
- Local `simpleType`

In all cases, the SQL data type, its associated information (length, precision), and the memory mapping information are derived from the `simpleType` on which the attribute is based.

Overriding the SQLType Value in an XML Schema When Declaring Attributes

You can explicitly specify a `SQLType` value in the input XML schema document. In this case, the data type you specify is used for schema validation. This allows for the following specific forms of overrides:

- If the default SQL data type is `STRING`, you can override it with `CHAR`, `VARCHAR`, or `CLOB`.
- If the default SQL data type is `RAW`, you can override it with `RAW` or `BLOB`.

Mapping XML Schema Element Data Types to SQL

An element declaration can specify its XML Schema data type in terms of one of the following:

- Any of the ways for specifying type for an attribute declaration. See "Mapping XML Schema Attribute Data Types to SQL" on page 7-47.
- Global `complexType`, specified within this XML schema document or in an external XML schema.
- Reference to a global element (`ref=". . ."`), which could itself be within this XML schema document or in an external XML schema.
- Local `complexType`.

Overriding the SQLType Value in an XML Schema when Declaring Elements

An element based on a `complexType` is, by default, mapped to a SQL object type that contains object attributes corresponding to each of the sub-elements and attributes. You can override this mapping by explicitly specifying a value for attribute `SQLType` in the input XML schema. The following values for `SQLType` are permitted here:

- `VARCHAR2`
- `RAW`
- `CLOB`
- `BLOB`

These represent storage of the XML data in a text form in the database.

For example, to override the `SQLType` from `VARCHAR2` to `CLOB`, declare the `xdb` namespace using `xmlns:xdb="http://xmlns.oracle.com/xdb"`, and then use `xdb:SQLType = "CLOB"`.

The following special cases are handled:

- If a cycle is detected when processing the `complexType` values that are used to declare elements and the elements declared within the `complexType`, the `SQLInline` attribute is forced to be `false`, and the correct SQL mapping is set to `REF XMLType`.
- If `maxOccurs > 1`, a varray type might be created.
  - If `SQLInline = "true"`, then a varray type is created whose element type is the SQL data type previously determined. Cardinality of the varray is based on the value of attribute `maxOccurs`. Either you specify the name of the varray type using attribute `SQLCollType`, or it is derived from the element name.
  - If `SQLInline = "false"`, then the SQL data type is set to `XDB.XDB$XMLTYPE_REF_LIST_T`. This is a predefined data type that represents an array of `REF` values pointing to `XMLType` instances.
- If the element is a global element, or if `SQLInline = "false"`, then the system creates a default table. Either you specify the name of the default table, or it is derived from the element name.

> **See Also:** Chapter 9, "XML Schema Storage and Query: Advanced" for more information about mapping `simpleType` values and `complexType` values to SQL.

Mapping simpleType to SQL

This section describes how XML schema definitions map XML Schema `simpleType` to SQL object types. Figure 7–6 shows an example of this.

Figure 7–6 Mapping simpleType: XML Strings to SQL VARCHAR2 or CLOB

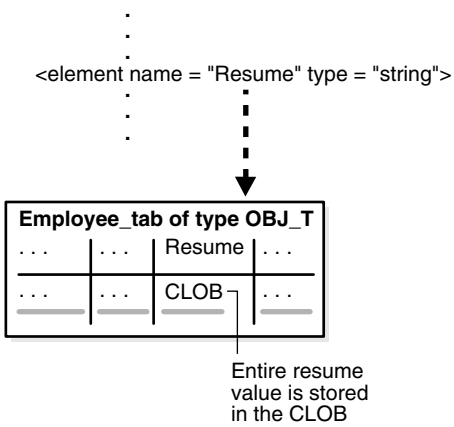


Table 7–6 through Table 7–9 present the default mapping of XML Schema `simpleType` to SQL, as specified in the XML Schema definition. For example:

- A XML Schema *primitive* type is mapped to the closest SQL data type. For example, `DECIMAL`, `POSITIVEINTEGER`, and `FLOAT` are all mapped to SQL `NUMBER`.
- An XML Schema *enumeration* type is mapped to a SQL object type with a single `RAW(n)` object attribute. The value of *n* is determined by the number of possible values in the enumeration declaration.
- An XML Schema *list* or a *union* type is mapped to a SQL string (`VARCHAR2` or `CLOB`) data type.

Table 7–6 Mapping XML Schema String Data Types to SQL

XML Schema String Type	Length or MaxLength Facet	Default SQL Data Type	Compatible SQL Data Type
`string`	n	`VARCHAR2(n)` if n < 4000, else `VARCHAR2(4000)`	`CHAR`, `CLOB`
`string`	-	`VARCHAR2(4000)` if `mapUnboundedStringToLob = "false"`, `CLOB`	`CHAR`, `CLOB`

Table 7–7 Mapping XML Schema Binary Data Types (hexBinary/base64Binary) to SQL

XML Schema Binary Type	Length or MaxLength Facet	Default SQL Data Type	Compatible SQL Data Type
`hexBinary`, `base64Binary`	n	`RAW(n)` if n < 2000, else `RAW(2000)`	`RAW`, `BLOB`
`hexBinary`, `base64Binary`	-	`RAW(2000)` if `mapUnboundedStringToLob = "false"`, `BLOB`	`RAW`, `BLOB`

Table 7–8 Default Mapping of Numeric XML Schema Primitive Types to SQL

XML Schema Simple Type	Default SQL Data Type	totalDigits (m), fractionDigits(n) Specified	Compatible SQL Data Types
`float`	`NUMBER`	`NUMBER(m+n,n)`	`FLOAT`, `DOUBLE`, `BINARY_FLOAT`
`double`	`NUMBER`	`NUMBER(m+n,n)`	`FLOAT`, `DOUBLE`, `BINARY_DOUBLE`
`decimal`	`NUMBER`	`NUMBER(m+n,n)`	`FLOAT`, `DOUBLE`

Table 7–8 (Cont.) Default Mapping of Numeric XML Schema Primitive Types to SQL

XML Schema Simple Type	Default SQL Data Type	totalDigits (m), fractionDigits(n) Specified	Compatible SQL Data Types
integer	NUMBER	NUMBER(m+n,n)	NUMBER
nonNegativeInteger	NUMBER	NUMBER(m+n,n)	NUMBER
positiveInteger	NUMBER	NUMBER(m+n,n)	NUMBER
nonPositiveInteger	NUMBER	NUMBER(m+n,n)	NUMBER
negativeInteger	NUMBER	NUMBER(m+n,n)	NUMBER
long	NUMBER(20)	NUMBER(m+n,n)	NUMBER
unsignedLong	NUMBER(20)	NUMBER(m+n,n)	NUMBER
int	NUMBER(10)	NUMBER(m+n,n)	NUMBER
unsignedInt	NUMBER(10)	NUMBER(m+n,n)	NUMBER
short	NUMBER(5)	NUMBER(m+n,n)	NUMBER
unsignedShort	NUMBER(5)	NUMBER(m+n,n)	NUMBER
byte	NUMBER(3)	NUMBER(m+n,n)	NUMBER
unsignedByte	NUMBER(3)	NUMBER(m+n,n)	NUMBER

Table 7–9 Mapping XML Schema Date and Time Data Types to SQL

XML Schema Date or Time Type	Default SQL Data Type	Compatible SQL Data Types
dateTime	TIMESTAMP	TIMESTAMP WITH TIME ZONE, DATE
time	TIMESTAMP	TIMESTAMP WITH TIME ZONE, DATE
date	DATE	TIMESTAMP WITH TIME ZONE
gDay	DATE	TIMESTAMP WITH TIME ZONE
gMonth	DATE	TIMESTAMP WITH TIME ZONE
gYear	DATE	TIMESTAMP WITH TIME ZONE
gYearMonth	DATE	TIMESTAMP WITH TIME ZONE
gMonthDay	DATE	TIMESTAMP WITH TIME ZONE
duration	VARCHAR2(4000)	none

Table 7–10 Default Mapping of Other XML Schema Primitive and Derived Data Types to SQL

XML Schema Primitive or Derived Type	Default SQL Data Type	Compatible SQL Data Types
boolean	RAW(1)	VARCHAR2
language(string)	VARCHAR2(4000)	CLOB, CHAR
NMTOKEN(string)	VARCHAR2(4000)	CLOB, CHAR
NMTOKENS(string)	VARCHAR2(4000)	CLOB, CHAR
Name(string)	VARCHAR2(4000)	CLOB, CHAR
NCName(string)	VARCHAR2(4000)	CLOB, CHAR
ID	VARCHAR2(4000)	CLOB, CHAR
IDREF	VARCHAR2(4000)	CLOB, CHAR

Table 7–10 (Cont.) Default Mapping of Other XML Schema Primitive and Derived Data Types to SQL

XML Schema Primitive or Derived Type	Default SQL Data Type	Compatible SQL Data Types
IDREFS	VARCHAR2(4000)	CLOB, CHAR
ENTITY	VARCHAR2(4000)	CLOB, CHAR
ENTITIES	VARCHAR2(4000)	CLOB, CHAR
NOTATION	VARCHAR2(4000)	CLOB, CHAR
anyURI	VARCHAR2(4000)	CLOB, CHAR
anyType	VARCHAR2(4000)	CLOB, CHAR
anySimpleType	VARCHAR2(4000)	CLOB, CHAR
QName	XDB.XDB$QNAME	none
normalizedString	VARCHAR2(4000)	none
token	VARCHAR2(4000)	none

NCHAR, NVARCHAR, and NCLOB SQLType Values are Not Supported

Oracle XML DB does *not* support NCHAR, NVARCHAR, and NCLOB as values for attribute SQLType: you cannot specify that an XML element or attribute is to be of type NCHAR, NVARCHAR, or NCLOB. Also, if you provide your own data type, do not use any of these data types.

See Also: Appendix B, "Oracle XML DB Restrictions"

simpleType: Mapping XML Strings to SQL VARCHAR2 Versus CLOB

If an XML schema specifies an XML Schema data type to be a string with a maxLength less than 4000, then it is mapped to a VARCHAR2 object attribute of the specified length. However, if maxLength is not specified in the XML schema, then it can only be mapped to a LOB. This is sub-optimal when most of the string values are small and only a small fraction of them are large enough to need a LOB.

See Also: Table 7–6, " Mapping XML Schema String Data Types to SQL"

Working with Time Zones

The following XML Schema data types allow for an optional time-zone indicator as part of their literal values.

- xsd:dateTime
- xsd:time
- xsd:date
- xsd:gYear
- xsd:gMonth
- xsd:gDay
- xsd:gYearMonth
- xsd:gMonthDay

By default, XML schema registration maps `xsd:dateTime` and `xsd:time` to SQL data type `TIMESTAMP` and all the other data types to SQL data type `DATE`. SQL data types `TIMESTAMP` and `DATE` do not permit a time-zone indicator.

If your application needs to work with time-zone indicators, then use attribute `SQLType` to specify the SQL data type as `TIMESTAMP WITH TIME ZONE`. This ensures that values containing time-zone indicators can be stored and retrieved correctly. For example:

```
<element name="dob" type="xsd:dateTime"
         xdb:SQLType="TIMESTAMP WITH TIME ZONE"/>
<attribute name="endofquarter" type="xsd:gMonthDay"
           xdb:SQLType="TIMESTAMP WITH TIME ZONE"/>
```

Using Trailing Z to Indicate UTC Time Zone XML Schema lets the time-zone component be specified as `Z`, to indicate UTC time zone. When a value with a trailing `Z` is stored in a SQL `TIMESTAMP WITH TIME ZONE` column, the time zone is actually stored as `+00:00`. Thus, the retrieved value contains the trailing `+00:00`, not the original `Z`. For example, if the value in the input XML document is `1973-02-12T13:44:32`**`Z`**, the output is `1973-02-12T13:44:32`**`.000000+00:00`**.

Mapping complexType to SQL

Using XML Schema, a `complexType` is mapped to a SQL object type as follows:

- XML attributes declared within the `complexType` are mapped to SQL object attributes. The `simpleType` defining an XML attribute determines the SQL data type of the corresponding object attribute.
- XML elements declared within the `complexType` are also mapped to SQL object attributes. The `simpleType` or `complexType` defining an XML element determines the SQL data type of the corresponding object attribute.

If the XML element is declared with attribute `maxOccurs` > 1, then it is mapped to a SQL collection (object) attribute. The collection could be a varray value (the default, recommended) or an unordered table (if you set attribute `xdb:maintainOrder` to `false`). The default storage of a varray value is an ordered collections table (OCT). You can choose LOB storage instead, by setting attribute `xdb:storeVarrayAsTable` to `false`.

Specifying Attributes in a complexType XML Schema Declaration

When you have an element based on a global `complexType`, both the `SQLType` and `SQLSchema` attributes must be specified for the `complexType` declaration. In addition you can optionally include the same `SQLType` and `SQLSchema` attributes within the element declaration.

If you do not specify attribute `SQLType` for the global `complexType`, Oracle XML DB creates a `SQLType` attribute with an internally generated name. The elements that reference this global type cannot then have a different value for `SQLType`. The following code is acceptable:

```
<xs:complexType name="LineItemsType" xdb:SQLType="LINEITEMS_T">
  <xs:sequence>
    <xs:element name="LineItem" type="LineItemType" maxOccurs="unbounded"
                xdb:SQLName="LINEITEM" xdb:SQLCollType="LINEITEM_V"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="LineItemType" xdb:SQLType="LINEITEM_T">
  <xs:sequence>
```

```
    <xs:element name="Description" type="DescriptionType"
                xdb:SQLName="DESCRIPTION"/>
    <xs:element name="Part" type="PartType" xdb:SQLName="PART"/>
  </xs:sequence>
  <xs:attribute name="ItemNumber" type="xs:integer" xdb:SQLName="ITEMNUMBER"
                xdb:SQLType="NUMBER"/>
</xs:complexType>
<xs:complexType name="PartType" xdb:SQLType="PART_T">
  <xs:attribute name="Id" xdb:SQLName="PART_NUMBER" xdb:SQLType="VARCHAR2">
    <xs:simpleType>
      <xs:restriction base="xs:string">
        <xs:minLength value="10"/>
        <xs:maxLength value="14"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:attribute>
  <xs:attribute name="Quantity" type="moneyType" xdb:SQLName="QUANTITY"/>
  <xs:attribute name="UnitPrice" type="quantityType" xdb:SQLName="UNITPRICE"/>
</xs:complexType>
```

Mapping XML Schema Data Types To Binary XML Encoding Types

This section describes how to use PL/SQL package `DBMS_XMLSCHEMA` to map the data types used for XML Schema attributes and elements to the encoding types used for binary XML data.

If you register an XML schema using option `REGISTER_BINARYXML`, then XML instance documents that reference that XML schema are stored as binary XML, and their XML Schema data types are encoded in the binary XML data. Each XML Schema data type is mapped to a default binary XML encoding type, but you can override this default mapping by using XML attribute `csx:encodingType`.

Different XML schemas specify possibly different binary XML encodings. If you used option `ALLOW ANYSCHEMA` when you created the `XMLType` table or column, then XML documents that reference different XML schemas, and that therefore might have different binary XML encodings, can be stored in the same `XMLType` column or table.

> **Note:** The binary XML encoding type has no effect on the data types of values used in query results.

Table 7–11 describes the available binary XML encoding types.Table 7–12 describes how they map to XML Schema data types by default, and which binary XML encoding types you can use to override the default mapping.

Table 7–11 Binary XML Encoding Types

Binary XML Encoding Type	Description
`string`	UTF-8 character data
`binary`	binary data
`boolean`	1 for `true`, 0 for `false` (one byte) This is always decoded to XML Schema as `true` or `false`, never as `1` or `0`.
`int`	signed, twos-complement, big-endian integer (1, 2, 4, or 8 bytes)
`unsigned-int`	twos-complement, big-endian integer (1, 2, 4, or 8 bytes)

Table 7–11 (Cont.) Binary XML Encoding Types

Binary XML Encoding Type	Description
`float`	IEEE-754 floating-point number (4 or 8 bytes)
`oranum`	Oracle internal number representation (maximum of 22 bytes)
	Numbers from 1.0×10^{-130} to 1.0×10^{126}. Maximum precision: 38 digits. Maximum scale: -84 to 127.
`oradate`	Oracle internal format for Gregorian dates (7 bytes)
	No fractional seconds. No time-zone information.
`orats`	Oracle internal format for Gregorian dates with time zone information (timestamp)
	1–9 digits for fractional seconds. An XML Schema time-zone component specified originally as `Z` and stored using `orats` is decoded to XML Schema as `+00:00`.
`epoch`	number of seconds since January 1, 1970 (4 or 8 bytes, signed binary)
	No fractional seconds. No time-zone information.
`epochtz`	same as `epoch`, but followed by number of minutes from GMT (4 or 8 bytes, signed binary, followed by 2-byte signed number)
	An XML Schema time-zone component specified originally as `Z` and stored using `epochtz` is decoded to XML Schema as `+00:00`.
`enum`	unsigned integers, starting with zero, one for each possible enumeration value, in the order of definition in the XML schema
`qname`	token ID followed by prefix ID (4 or 8 bytes, followed by 2 bytes)

Table 7–12 Mapping XML Schema Data Types to Binary XML Encoding Types

XML Schema Type	Binary XML Encoding Type	Compatible Encoding Types
`string`	`string`	none
`hexBinary`	`binary`	`string`
`base64binary`	`binary`	`string`
`float`	`float`	`string`
`double`	`float`	`string`
`decimal`	`oranum`	`string`
`integer`	`oranum`	`string`
`nonNegativeInteger`	`oranum`	`string`
`positiveInteger`	`oranum`	`string`
`nonPositiveInteger`	`oranum`	`string`
`negativeInteger`	`oranum`	`string`
`long`	`int`	`oranum`, `string`
`unsignedLong`	`unsigned-int`	`oranum`, `string`
`int`	`int`	`oranum`, `string`
`unsignedInt`	`unsigned-int`	`oranum`, `string`
`short`	`int`	`oranum`, `string`
`unsignedShort`	`unsigned-int`	`oranum`, `string`

Table 7–12 (Cont.) Mapping XML Schema Data Types to Binary XML Encoding Types

XML Schema Type	Binary XML Encoding Type	Compatible Encoding Types
byte	int	oranum, string
unsignedByte	unsigned-int	oranum, string
dateTime	orats	oradate, epoch, epochtz, string
time	orats	oradate, epoch, epochtz, string
date	orats	oradate, epoch, epochtz, string
gDay	orats	oradate, epoch, epochtz, string
gMonth	orats	oradate, epoch, epochtz, string
gYear	orats	oradate, epoch, epochtz, string
gYearMonth	orats	oradate, epoch, epochtz, string
gMonthDay	orats	oradate, epoch, epochtz, string
duration	string	none
boolean	boolean	string
language	string	none
NMTOKEN	string	none
NMTOKENS	string	none
Name	string	none
NCName	string	none
ID	string	none
IDREF	string	none
IDREFS	string	none
ENTITY	string	none
ENTITITES	string	none
NOTATION	string	none
anyURI	string	none
anyType	internal encoding[1]	none
anySimpleType	string	none
QName	qname	string
normalizedString	string	none
token	string	none

[1] You cannot specify the encoding type for XML Schema type anyType.

8

XPath Rewrite for Structured Storage

This chapter explains the fundamentals of XPath rewrite for structured (object-relational) storage in Oracle XML DB. It details the rewriting of XPath-expression arguments to various SQL functions.

This chapter contains these topics:

- Overview of XPath Rewrite for Structured Storage
- Sample of XPath Expressions that Are Rewritten
- Analyzing and Optimizing XPath Queries using Execution Plans

See Also: "Performance Tuning for XQuery" on page 5-30

Overview of XPath Rewrite for Structured Storage

Oracle XML DB can often optimize queries that use XPath expressions—for example, queries involving SQL functions such as `XMLQuery`, `XMLTable`, `XMLExists`, and `updateXML`, which take XPath (XQuery) expressions as arguments. The XPath expression is, in effect, evaluated against the XML document without ever constructing the XML document in memory.

This optimization is called **XPath rewrite**. It is a proper subset of XML query optimization, which also involves optimization of XQuery expressions, such as FLWOR expressions, that are not XPath expressions. XPath rewrite also enables indexes, if present on the column, to be used in query evaluation by the Optimizer.

The XPath expressions that can be rewritten by Oracle XML DB are a proper subset of those that are supported by Oracle XML DB. Whenever you can do so without losing functionality, use XPath expressions that can be rewritten.

XPath rewrite can occur in these contexts (or combinations thereof):

- When `XMLType` data is stored in an object-relational column or table (structured storage) or when an `XMLType` view is built on relational data.
- When you use an `XMLIndex` index. See "XMLIndex" on page 6-7.
- When `XMLType` data is stored as binary XML. See "How Oracle XML DB Processes XMLType Methods and SQL Functions" on page 3-58 for information about streaming evaluation.

This chapter covers the first case: rewriting queries that use structured XML data or `XMLType` views. The `XMLType` views can be XML schema-based or not. Structured storage of `XMLType` data is always XML schema-based. Examples in this chapter are related to XML schema-based tables.

Example 8–1 illustrates XPath rewrite for a simple query that uses an XPath expression.

Example 8–1 XPath Rewrite

```
SELECT po.OBJECT_VALUE FROM purchaseorder po
  WHERE XMLCast(XMLQuery('$p/PurchaseOrder/Requestor'
                         PASSING po.OBJECT_VALUE AS "p" RETURNING CONTENT)
                 AS VARCHAR2(128))
        = 'Sarah J. Bell';
```

The `XMLCast(XMLQuery...))` expression here is rewritten to the underlying relational column that stores the requestor information for the purchase order. The query is rewritten to something like the following:[1]

```
SELECT OBJECT_VALUE FROM purchaseorder p
 WHERE CAST (p."XMLDATA"."REQUESTOR" AS VARCHAR2(128)) = 'Sarah J. Bell';
```

Sample of XPath Expressions that Are Rewritten

Table 8–1 describes some XPath expressions that are rewritten during XPath rewrite.

Table 8–1 Sample of XPath Expressions that Are Rewritten to Underlying SQL Constructs

XPath Expression for Translation	Description

Simple XPath expressions (expressions with `child` and `attribute` axes only):
`/PurchaseOrder/@Reference`
`/PurchaseOrder/Requestor`
	Involves traversals over object type attributes only, where the attributes are simple scalar or object types themselves.

Collection traversal expressions: `/PurchaseOrder/LineItems/LineItem/Part/@Id`	Involves traversal of collection expressions. The only axes supported are child and attribute axes. Collection traversal is not supported if the SQL function is used during a `CREATE INDEX` operation.
Predicates: `[Requestor = "Sarah J. Bell"]`	Predicates in the XPath are rewritten into SQL predicates.
List index (positional predicate): `LineItem[1]`	Indexes are rewritten to access the nth item in a collection.
Wildcard traversals: `/PurchaseOrder/*/Part/@Id`	If the wildcard can be translated to one or more simple XPath expressions, then it is rewritten.

[1] This example uses sample database schema `OE` and its table `purchaseorder`. The XML schema for this table is annotated with attribute `SQLName` to specify SQL object attribute names such as `REQUESTOR`—see Example 3–10 on page 3-20. Without such annotations, this example would use `p."XMLDATA"."Requestor"`, not `p."XMLDATA".".REQUESTOR"`.

Table 8–1 (Cont.) Sample of XPath Expressions that Are Rewritten to Underlying SQL Constructs

XPath Expression for Translation	Description
Descendant axis (XML schema-based data only), without recursion: `/PurchaseOrder//Part/@Id`	Similar to a wildcard expression. The `descendant` axis is rewritten if it can be mapped to one or more simple XPath expressions.
Descendant axis (XML schema-based data only), with *recursion*: `/PurchaseOrder//Part/@Id`	The `descendant` axis is rewritten if both of these conditions holds: ▪ All simple XPath expressions to which this XPath expression expands map to the same out-of-line table. ▪ Any simple XPath expression to which this XPath expression does not expand does not map to that out-of-line table.
XPath functions	Some XPath functions are rewritten. These functions include `not`, `floor`, `ceiling`, `substring`, and `string-length`.

See Also: "Performance Tuning for XQuery" on page 5-30 for information about rewrite of XQuery expressions

Analyzing and Optimizing XPath Queries using Execution Plans

This section presents some guidelines for using execution plans to do the following, for queries that use XPath expressions:

- Analyze query execution, to determine whether XPath rewrite occurs.
- Optimize query execution, by using secondary indexes.

Use these guidelines together, taking all that apply into consideration.

As is true also for the rest of this chapter, this section is applicable only to `XMLType` data that is stored object-relationally (structured storage).

XPath rewrite for object-relational storage means that a query that selects XML fragments defined by an XPath expression is rewritten to a SQL `SELECT` statement on the underlying object-relational tables and columns. These underlying tables can include out-of-line tables.

See Also: "XPath Rewrite for Out-Of-Line Tables" on page 9-7

Guideline: Look for underlying tables versus XML functions in execution plans

The execution plan of a query that has been rewritten refers to the object-relational tables and columns that underlie the queried `XMLType` data.

The names of the underlying tables can be meaningful to you, if they are derived from XML element or attribute names or if the governing XML schema explicitly names them by using annotation `xdb:defaultTable`. Otherwise, these names are system-generated and have no obvious meaning. In particular, they do not reflect the corresponding XML element or attribute names. Also, some system-generated columns are hidden. You do not see them if you use the SQL `describe` command. They nevertheless show up in execution plans.

The plan of a query that has not been rewritten shows only the base table names, and it typically refers to user-level XML functions, such as `XMLExists`. Look for this difference to determine whether a query has been optimized. The XML function name

shown in an execution plan is actually the internal name (for example, XMLEXISTS2), which is sometimes slightly different from the user-level name.

Example 8–2 shows the kind of execution plan output that is generated when Oracle XML DB cannot perform XPath rewrite. The plan here is for a query that uses SQL/XML function XMLExists. The corresponding internal function XMLExists2 appears in the plan output, indicating that the query is not rewritten.

Example 8–2 Execution Plan Generated When XPath Rewrite Does Not Occur

```
Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter(XMLEXISTS2('$p/PurchaseOrder[User="SBELL"]' PASSING BY VALUE
             SYS_MAKEXML('61687B202644E297E040578C8A175C1D',4215,"PO"."XMLEXTRA","PO"."X
             MLDATA") AS "p")=1)
```

In this situation, Oracle XML DB constructs a pre-filtered result set based on any other conditions specified in the query WHERE clause. It then filters the rows in this potential result set to determine which rows belong in the result set. The filtering is performed by *constructing a DOM on each document* and performing a *functional evaluation* (using the methods defined by the DOM API) to determine whether or not each document is a member of the result set.

Guideline: Name the default tables, so you recognize them in execution plans

When designing an XML schema, use annotation xdb:defaultTable to name the underlying tables that correspond to elements that you select in queries where performance is important. This lets you easily recognize them in an execution plan, indicating by their presence or absence whether the query has been rewritten.

Guideline: Create an index on a column targeted by a predicate

A query resulting from XPath rewrite sometimes includes a SQL predicate (WHERE clause). This can happen even if the original query does not use an XPath predicate, and it can happen even if the original query does not have a SQL WHERE clause.

When this happens, you can sometimes improve performance by creating an index on the column that is targeted by the SQL predicate, or by creating an index on a function application to that column. Example 8–1 illustrates XPath rewrite for a query that includes a WHERE clause. Example 8–3 shows the predicate information from an execution plan for this query.

Example 8–3 Analyzing an Execution Plan to Determine a Column to Index

```
Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter(CAST("PURCHASEORDER"."SYS_NC00021$" AS VARCHAR2(128))='Sarah
             J. Bell' AND SYS_CHECKACL("ACLOID","OWNERID",xmltype('<privilege
             xmlns="http://xmlns.oracle.com/xdb/acl.xsd"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://xmlns.oracle.com/xdb/acl.xsd
             http://xmlns.oracle.com/xdb/acl.xsd DAV:http://xmlns.oracle.com/xdb/dav.xsd
             "><read-properties/><read-contents/></privilege>'))=1)
```

The predicate information indicates that the expression XMLCast(XMLQuery...)) is rewritten to an application of SQL function cast to the underlying relational column that stores the requestor information for the purchase order, SYS_NC0021$. This

column name is system-generated. The execution plan refers to this system-generated name, in spite of the fact that the governing XML schema uses annotation `SQLName` to name this column `REQUESTOR`.

Because these two names (user-defined and system-generated) refer to the same column, you can create a B-tree index on this column using either name. Alternatively, you can use the `extractValue` shortcut to create the index, by specifying an XPath expression that targets the purchase-order requestor data. Example 8–4 shows these three equivalent ways to create the B-tree index on the predicate-targeted column.

Example 8–4 Creating an Index on a Column Targeted by a Predicate

```
CREATE INDEX requestor_index ON purchaseorder ("SYS_NC00021$");

CREATE INDEX requestor_index ON purchaseorder ("XMLDATA"."REQUESTOR");

CREATE INDEX requestor_index ON purchaseorder
  (extractvalue(OBJECT_VALUE, '/PurchaseOrder/Requestor'));
```

However, for this particular query it makes sense to create a function-based index, using a functional expression that matches the one in the rewritten query. Example 8–5 illustrates this.

Example 8–5 Creating a Function-Based Index for a Column Targeted by a Predicate

```
CREATE INDEX requestor_index ON purchaseorder
  (cast("XMLDATA"."REQUESTOR" AS VARCHAR2(128)));
```

Example 8–6 shows an execution plan that indicates that the index is picked up.

Example 8–6 Execution Plan Showing that Index Is Picked Up

```
-----------------------------------------------------------------------------------------------
| Id  | Operation                    | Name            | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                 |     1 |   524 |     2   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS BY INDEX ROWID| PURCHASEORDER   |     1 |   524 |     2   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN           | REQUESTOR_INDEX |     1 |       |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter(SYS_CHECKACL("ACLOID","OWNERID",xmltype('<privilege
              xmlns="http://xmlns.oracle.com/xdb/acl.xsd"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xsi:schemaLocation="http://xmlns.oracle.com/xdb/acl.xsd
                                  http://xmlns.oracle.com/xdb/acl.xsd
              DAV:http://xmlns.oracle.com/xdb/dav.xsd">
              <read-properties/><read-contents/></privilege>'))=1)
   2 - access(CAST("SYS_NC00021$" AS VARCHAR2(128))='Sarah J. Bell')
```

In the particular case of this query, the original functional expression applies `XMLCast` to `XMLQuery` to target a singleton element, `Requestor`. This is a special case, where you can as a shortcut use such a functional expression directly in the `CREATE INDEX` statement. That statement is rewritten to create an index on the underlying scalar data. Example 8–7, which targets an XPath expression, thus has the same effect as Example 8–5, which targets the corresponding object-relational column.

Example 8–7 Creating a Function-Based Index for a Column Targeted by a Predicate

```
CREATE INDEX requestor_index
  ON purchaseorder po
     (XMLCast(XMLQuery('$p/PurchaseOrder/Requestor' PASSING po.OBJECT_VALUE AS "p"
                                                    RETURNING CONTENT)
              AS VARCHAR2(128)));
```

See Also: "Indexing Non-Repeating text() Nodes or Attribute Values" on page 6-6 for information about using the shortcut of `XMLCast` applied to `XMLQuery` and the `extractValue` shortcut to index singleton data

Guideline: Create indexes on ordered collection tables

If a collection is stored as an ordered collection table or an `XMLType` instance, then you can directly access members of the collection. Each member of the collection becomes a row in a table, so you can access it directly with SQL.

You can often improve performance by indexing such collection members. You do this by creating a *composite* index on (a) the object attribute that corresponds to the collection XML element or its attribute and (b) pseudocolumn `NESTED_TABLE_ID`.

Example 8–8 shows the execution plan for a query to find the `Reference` elements in documents that contain an order for part number 717951002372 (`Part` element with an `Id` attribute of value `717951002372`). The collection of `LineItem` elements is stored as rows in the ordered collection table `lineitem_table`.

Note: Example 8–8 does not use the `purchaseorder` table from sample database schema `OE`. It uses the `purchaseorder` table defined in Example 3–13 on page 3-28. This table uses an ordered collection table (OCT) named `lineitem_table` for the collection element `LineItem`.

Example 8–8 Execution Plan for a Selection of Collection Elements

```
SELECT XMLCast(XMLQuery('$p/PurchaseOrder/Reference'
                        PASSING OBJECT_VALUE AS "p" RETURNING CONTENT)
               AS VARCHAR2(4000)) "Reference"
  FROM purchaseorder
  WHERE XMLExists('$p/PurchaseOrder/LineItems/LineItem/Part[@Id="717951002372"]'
                  PASSING OBJECT_VALUE AS "p");

-----------------------------------------------------------------------------------------------
| Id  | Operation                    | Name                   | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                        |     1 |   122 |    16  (13)| 00:00:01 |
|   1 |  NESTED LOOPS                |                        |       |       |            |          |
|   2 |   NESTED LOOPS               |                        |     1 |   122 |    16  (13)| 00:00:01 |
|   3 |    SORT UNIQUE               |                        |     1 |    50 |    14   (8)| 00:00:01 |
|*  4 |     TABLE ACCESS FULL        | LINEITEM_TABLE         |     1 |    50 |    14   (8)| 00:00:01 |
|*  5 |    INDEX UNIQUE SCAN         | LINEITEM_TABLE_MEMBERS |     1 |       |     0   (0)| 00:00:01 |
|   6 |   TABLE ACCESS BY INDEX ROWID| PURCHASEORDER          |     1 |    72 |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   4 - filter("SYS_NC00009$" IS NOT NULL AND "SYS_NC00011$"='717951002372')
   5 - access("NESTED_TABLE_ID"="PURCHASEORDER"."SYS_NC0003400035$")
```

The execution plan shows a full scan of ordered collection table `lineitem_table`. This could be acceptable if there were only a few hundred documents in the `purchaseorder` table, but it would be unacceptable if there were thousands or millions of documents in the table.

To improve the performance of such a query, you can create an index that provides direct access to pseudocolumn `NESTED_TABLE_ID`, given the value of attribute `Id`. Unfortunately, Oracle XML DB does not allow indexes on collections to be created using XPath expressions directly. To create the index, you must understand the structure of the SQL object that is used to manage the `LineItem` elements. Given this information, you can create the required index using conventional object-relational SQL.

In this case, element `LineItem` is stored as an instance of object type `lineitem_t`. Element `Part` is stored as an instance of SQL data type `part_t`. XML attribute `Id` is mapped to object attribute `part_number`. Given this information, you can create a *composite index* on attribute `part_number` and pseudocolumn `NESTED_TABLE_ID`, as shown in Example 8–9. This index provides direct access to those purchase-order documents that have `LineItem` elements that reference the required part.

Example 8–9 Creating an Index for Direct Access to an Ordered Collection Table

```
CREATE INDEX lineitem_part_index ON lineitem_table l (l.part.part_number, l.NESTED_TABLE_ID);
```

Guideline: Use XMLOptimizationCheck to determine why a query is not rewritten

If a query has not been optimized, you can use system variable `XMLOptimizationCheck` to try to determine why.

See Also: "Diagnosing XQuery Optimization: XMLOptimizationCheck" on page 5-35

9

XML Schema Storage and Query: Advanced

This chapter describes advanced techniques for storing structured XML schema-based `XMLType` objects.

See Also:

- Chapter 7, "XML Schema Storage and Query: Basic" for basic information about using XML Schema with Oracle XML DB
- Chapter 8, "XPath Rewrite for Structured Storage" for information about the optimization of XPath expressions in Oracle XML DB
- Chapter 10, "XML Schema Evolution" for information about updating an XML schema after you have registered it with Oracle XML DB
- `http://www.w3.org/TR/xmlschema-0/` for an introduction to XML Schema

This chapter contains these topics:

- Generating XML Schemas with DBMS_XMLSCHEMA.GENERATESCHEMA
- Adding Unique Constraints to the Parent Element of an Attribute
- Setting Annotation Attribute SQLInline to false for Out-Of-Line Storage
- Storing Collections in Out-Of-Line Tables
- Partitioning XMLType Tables and Columns Stored Object-Relationally
- Fully Qualified XML Schema URLs
- Mapping XML Fragments to Large Objects (LOBs)
- complexType Extensions and Restrictions in Oracle XML DB
- XML Schema: Working with Circular and Cyclical Dependencies
- Support for Recursive Schemas
- Loading and Retrieving Large Documents with Collections

Generating XML Schemas with DBMS_XMLSCHEMA.GENERATESCHEMA

An XML schema can be generated from an object-relational type automatically using a default mapping. PL/SQL functions `generateSchema` and `generateSchemas` in package `DBMS_XMLSCHEMA` take in a string that has the object type name and another that has the Oracle XML DB XML schema.

- Function `generateSchema` returns an `XMLType` containing an XML schema. It can optionally generate an XML schema for all types referenced by the given object type or restricted only to the top-level types.
- Function `generateSchemas` is similar, except that it returns an `XMLSequenceType` value. This is a varray of `XMLType` instances, each of which is an XML schema that corresponds to a different namespace. It also takes an additional optional argument that specifies the root URL of the preferred XML schema location:

  `http://xmlns.oracle.com/xdb/schemas/<schema>.xsd`

Both `generateSchema` and `generateSchemas` return XML schemas that have Oracle XML DB annotations.

Example 9–1 shows the definition of an object type, `employee_t`, and an invocation of `generateSchema` that generates an XML schema for that object type.

Example 9–1 Generating an XML Schema with Function GENERATESCHEMA

```
CREATE TYPE employee_t AS OBJECT(empno NUMBER(10),
                                 ename VARCHAR2(200),
                                 salary NUMBER(10,2)):

SELECT DBMS_XMLSCHEMA.generateSchema('T1', 'EMPLOYEE_T') FROM DUAL;

DBMS_XMLSCHEMA.GENERATESCHEMA('T1', 'EMPLOYEE_T')
-------------------------------------------------------------------------
<xsd:schema targetNamespace="http://ns.oracle.com/xdb/T1"
            xmlns="http://ns.oracle.com/xdb/T1"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:xdb="http://xmlns.oracle.com/xdb"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="http://xmlns.oracle.com/xdb
                                http://xmlns.oracle.com/xdb/XDBSchema.xsd">
  <xsd:element name="EMPLOYEE_T" type="EMPLOYEE_TType"
               xdb:SQLType="EMPLOYEE_T" xdb:SQLSchema="T1"/>
  <xsd:complexType name="EMPLOYEE_TType">
    <xsd:sequence>
      <xsd:element name="EMPNO" type="xsd:double" xdb:SQLName="empno"
                   xdb:SQLType="NUMBER"/>
      <xsd:element name="ENAME" type="xsd:string" xdb:SQLName="ename"
                   xdb:SQLType="VARCHAR2"/>
      <xsd:element name="SALARY" type="xsd:double" xdb:SQLName="salary"
                   xdb:SQLType="NUMBER"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

The generated XML schema declares element `EMPLOYEE_T` and complex type `EMPLOYEE_TType`. It uses annotations from namespace `http://xmlns.oracle.com/xdb`.

> **See Also:** "Creating XMLType Tables and Columns Based on XML Schemas" on page 7-27

Adding Unique Constraints to the Parent Element of an Attribute

After creating an `XMLType` table based on an XML schema, how can you add a unique constraint to the parent element of an attribute? You might, for example, want to create a unique key based on an attribute of an element that repeats itself (a collection).

To create constraints on elements that can occur more than once, store the varray as an ordered collection table (OCT). You can then create constraints on the OCT.

Example 9–2 shows an XML schema that lets attribute `No` of element `<PhoneNumber>` appear more than once. The example shows how you can add a unique constraint to ensure that the same phone number cannot be repeated within a given instance document.

Example 9–2 Adding a Unique Constraint to the Parent Element of an Attribute

```
BEGIN DBMS_XMLSCHEMA.registerSchema(
  SCHEMAURL => 'emp.xsd',
  SCHEMADOC => '<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
                           xmlns:xdb="http://xmlns.oracle.com/xdb">
                 <xs:element name="Employee" xdb:SQLType="EMP_TYPE">
                   <xs:complexType>
                     <xs:sequence>
                       <xs:element name="EmployeeId" type="xs:positiveInteger"/>
                       <xs:element name="PhoneNumber" maxOccurs="10"/>
                         <xs:complexType>
                           <xs:attribute name="No" type="xs:integer"/>
                         </xs:complexType>
                       </xs:element>
                     </xs:sequence>
                   </xs:complexType>
                 </xs:element>
               </xs:schema>',
   LOCAL     => FALSE,
   GENTYPES  => FALSE);
END;
/

PL/SQL procedure successfully completed.

CREATE TABLE emp_tab OF XMLType
  XMLSCHEMA "emp.xsd" ELEMENT "Employee"
  VARRAY XMLDATA."PhoneNumber" STORE AS TABLE phone_tab;

Table created.

ALTER TABLE phone_tab ADD UNIQUE (NESTED_TABLE_ID, "No");

Table altered.

INSERT INTO emp_tab
  VALUES(XMLType('<Employee>
                   <EmployeeId>1234</EmployeeId>
                   <PhoneNumber No="1234"/>
                   <PhoneNumber No="2345"/>
                 </Employee>').createSchemaBasedXML('emp.xsd'));

1 row created.

INSERT INTO emp_tab
  VALUES(XMLType('<Employee>
```

```
  <EmployeeId>3456</EmployeeId>
  <PhoneNumber No="4444"/>
  <PhoneNumber No="4444"/>
</Employee>').createSchemaBasedXML('emp.xsd'));
```

This returns the expected result:

```
*
ERROR at line 1:
ORA-00001: unique constraint (SCOTT.SYS_C002136) violated
```

The constraint in this example applies to each collection, and not across all instances. This is achieved by creating a concatenated index with the collection id column. To apply the constraint across all collections of all instance documents, omit the collection id column.

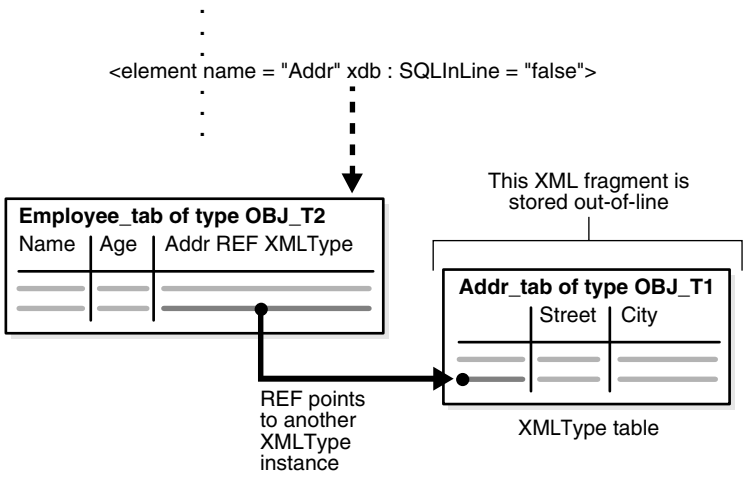
Note: You can create only a *functional* constraint as a unique or foreign key constraint on `XMLType` data stored as binary XML.

Setting Annotation Attribute SQLInline to false for Out-Of-Line Storage

By default, a child XML element is mapped to an embedded SQL object attribute, when `XMLType` data is stored object-relationally. However, there are scenarios where out-of-line storage offers better performance. In such cases, set XML schema annotation (attribute) `xdb:SQLInline` to `false`, so Oracle XML DB generates a SQL object type with an embedded `REF` attribute. The `REF` points to another `XMLType` instance that is stored out of line and that corresponds to the XML fragment. Default `XMLType` tables are also created, to store the out-of-line fragments.

Figure 9–1 illustrates the mapping of `complexType` to SQL for out-of-line storage.

Figure 9–1 Mapping complexType to SQL for Out-Of-Line Storage



Note: Starting with Oracle Database 11g Release 2 (11.2.0.2), you can create only *one* `XMLType` table that uses an XML schema that results in `an out-of-line table`. An error is raised if you try to create a second table that uses the same XML schema.

Example 9–3 Setting SQLInline to False for Out-Of-Line Storage

```
DECLARE
  doc VARCHAR2(3000) :=
    '<schema xmlns="http://www.w3.org/2001/XMLSchema"
             targetNamespace="http://www.oracle.com/emp.xsd"
             xmlns:emp="http://www.oracle.com/emp.xsd"
             xmlns:xdb="http://xmlns.oracle.com/xdb">
       <complexType name="EmpType" xdb:SQLType="EMP_T">
         <sequence>
           <element name="Name" type="string"/>
           <element name="Age" type="decimal"/>
           <element name="Addr"
                    xdb:SQLInline="false"
                    xdb:defaultTable="ADDR_TAB">
             <complexType xdb:SQLType="ADDR_T">
               <sequence>
                 <element name="Street" type="string"/>
                 <element name="City" type="string"/>
               </sequence>
             </complexType>
           </element>
         </sequence>
       </complexType>
       <element name="Employee" type="emp:EmpType"
                xdb:defaultTable="EMP_TAB"/>
     </schema>';
BEGIN
  DBMS_XMLSCHEMA.registerSchema(
    SCHEMAURL        => 'emp.xsd',
    SCHEMADOC        => doc,
    ENABLE_HIERARCHY => DBMS_XMLSCHEMA.ENABLE_HIERARCHY_NONE);
END;
/
```

In Example 9–3, attribute `xdb:SQLInline` of element `Addr` has value `false`. The resulting SQL object type, `obj_t2`, has an `XMLType` column with an embedded `REF` object attribute. The `REF` attribute points to an `XMLType` instance of SQL object type `obj_t1` in table `addr_tab`. Table `addr_tab` is stored out of line. It has columns `street` and `city`.

When registering this XML schema, Oracle XML DB generates the `XMLType` tables and types shown in Example 9–4.

Example 9–4 Generated XMLType Tables and Types

```
DESCRIBE emp_tab
 Name                          Null?    Type
 ----------------------------- --------
-----------------------------------------------------------
TABLE of SYS.XMLTYPE(XMLSchema "emp.xsd" Element "Employee") STORAGE
Object-relational TYPE "EMP_T"

DESCRIBE addr_tab
 Name                          Null?    Type
 ----------------------------- --------
-----------------------------------------------------------
TABLE of SYS.XMLTYPE(XMLSchema "emp.xsd" Element "Addr") STORAGE Object-relational
TYPE "ADDR_T"

DESCRIBE emp_t
```

```
 emp_t is NOT FINAL
 Name                          Null?    Type
 ----------------------------- -------- --------------------
 SYS_XDBPD$                             XDB.XDB$RAW_LIST_T
 Name                                   VARCHAR2(4000 CHAR)
 Age                                    NUMBER
 Addr                                   REF OF XMLTYPE

DESCRIBE addr_t
 Name                          Null?    Type
 ----------------------------- -------- --------------------
 SYS_XDBPD$                             XDB.XDB$RAW_LIST_T
 Street                                 VARCHAR2(4000 CHAR)
 City                                   VARCHAR2(4000 CHAR)
```

Table `emp_tab` holds all of the employee information, and it contains an object reference that points to the address information that is stored out of line, in table `addr_tab`.

An advantage of this model is that it lets you query the out-of-line table (`addr_tab`) directly, to look up address information. Example 9–5 illustrates querying table `addr_tab` directly to obtain the distinct city information for all employees.

Example 9–5 Querying an Out-Of-Line Table

```
INSERT INTO emp_tab
  VALUES
    (XMLType('<x:Employee
                 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                 xmlns:x="http://www.oracle.com/emp.xsd"
                 xsi:schemaLocation="http://www.oracle.com/emp.xsd emp.xsd">
                <Name>Abe Bee</Name>
                <Age>22</Age>
                <Addr>
                  <Street>A Street</Street>
                  <City>San Francisco</City>
                </Addr>
              </x:Employee>'));

INSERT INTO emp_tab
  VALUES
    (XMLType('<x:Employee
                 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                 xmlns:x="http://www.oracle.com/emp.xsd"
                 xsi:schemaLocation="http://www.oracle.com/emp.xsd emp.xsd">
                <Name>Cecilia Dee</Name>
                <Age>23</Age>
                <Addr>
                  <Street>C Street</Street>
                  <City>Redwood City</City>
                </Addr>
              </x:Employee>'));
. . .

SELECT DISTINCT XMLCast(XMLQuery('/Addr/City' PASSING OBJECT_VALUE AS "."
                                              RETURNING CONTENT)
                        AS VARCHAR2(20))
  FROM addr_tab;

CITY
```

```
-------------
Redwood City
San Francisco
```

The disadvantage of this storage model is that, in order to obtain the entire `Employee` element, you must access an additional table for the address.

XPath Rewrite for Out-Of-Line Tables

XPath expressions that involve elements stored out of line can be rewritten. The rewritten query involves a join with the out-of-line table. Example 9–6 shows such a query.

Example 9–6 XPath Rewrite for an Out-Of-Line Table

```
SELECT XMLCast(XMLQuery('declare namespace x = "http://www.oracle.com/emp.xsd"; (: :)
                        /x:Employee/Name' PASSING OBJECT_VALUE RETURNING CONTENT)
               AS VARCHAR2(20))
  FROM emp_tab
  WHERE XMLExists('declare namespace x = "http://www.oracle.com/emp.xsd"; (: :)
                   /x:Employee/Addr[City="San Francisco"]' PASSING OBJECT_VALUE);

XMLCAST(XMLQUERY(...
--------------------
Abe Bee
Eve Fong
George Hu
Iris Jones
Karl Luomo
Marina Namur
Omar Pinano
Quincy Roberts

8 rows selected.
```

The XQuery expression here is rewritten to a SQL `EXISTS` subquery that queries table `addr_tab`, joining it with table `emp_tab` using the object identifier column in `addr_tab`. The optimizer uses full table scans of tables `emp_tab` and `addr_tab`. If there are many entries in the `addr_tab`, then you can try to make this query more efficient by creating an index on the city, as shown in Example 9–7. An explain-plan fragment for the same query as in Example 9–6 shows that the city index is picked up.

Example 9–7 Using an Index with an Out-Of-Line Table

```
CREATE INDEX addr_city_idx
  ON addr_tab (extractValue(OBJECT_VALUE, '/Addr/City'));

|   2 |   TABLE ACCESS BY INDEX ROWID| ADDR_TAB      |     1 |  2012 |     1   (0)| 00:00:01 |
|*  3 |    INDEX RANGE SCAN          | ADDR_CITY_IDX |     1 |       |     1   (0)| 00:00:01 |
|   4 |   TABLE ACCESS FULL          | EMP_TAB       |    16 | 32464 |     2   (0)| 00:00:01 |
```

Note: When gathering statistics for the optimizer on an `XMLType` table that is stored object-relationally, Oracle recommends that you gather statistics on *all* of the tables defined by the XML schema, that is, all of the tables in `USER_XML_TABLES`. You can use procedure `DBMS_STATS.gather_schema_stats` to do this, or use `DBMS_STATS.gather_table_stats` on each such table. This informs the optimizer about all of the dependent tables that are used to store the `XMLType` data.

See Also: Chapter 8, "XPath Rewrite for Structured Storage"

Storing Collections in Out-Of-Line Tables

You can also map collection items to be stored out of line. In this case, instead of a single `REF` column, the parent element contains a varray of `REF` values that point to the collection members. For example, suppose that there is a list of addresses for each employee and that list is mapped to out-of-line storage, as shown in Example 9–8.

Example 9–8 Storing a Collection Out of Line

```
DECLARE
  doc VARCHAR2(3000) :=
    '<schema xmlns="http://www.w3.org/2001/XMLSchema"
             targetNamespace="http://www.oracle.com/emp.xsd"
             xmlns:emp="http://www.oracle.com/emp.xsd"
             xmlns:xdb="http://xmlns.oracle.com/xdb">
       <complexType name="EmpType" xdb:SQLType="EMP_T">
         <sequence>
           <element name="Name" type="string"/>
           <element name="Age" type="decimal"/>
           <element name="Addr" xdb:SQLInline="false"
                    maxOccurs="unbounded" xdb:defaultTable="ADDR_TAB">
             <complexType xdb:SQLType="ADDR_T">
               <sequence>
                 <element name="Street" type="string"/>
                 <element name="City" type="string"/>
               </sequence>
             </complexType>
           </element>
         </sequence>
       </complexType>
       <element name="Employee" type="emp:EmpType"
                xdb:defaultTable="EMP_TAB"/>
     </schema>';
BEGIN
  DBMS_XMLSCHEMA.registerSchema(
    SCHEMAURL        => 'emp.xsd',
    SCHEMADOC        => doc,
    ENABLE_HIERARCHY => DBMS_XMLSCHEMA.ENABLE_HIERARCHY_NONE);
END;
/
```

During registration of this XML schema, Oracle XML DB generates tables `emp_tab` and `addr_tab` and types `emp_t` and `addr_t`, just as in Example 9–3. However, this time, type `emp_t` contains a varray of `REF` values that point to addresses, instead of a single `REF` attribute, as shown in Example 9–9.

Example 9–9 Generated Out-Of-Line Collection Type

```
DESCRIBE emp_t
 emp_t is NOT FINAL
 Name                                    Null?    Type
 ----------------------------------------- -------- ---------------------------
 SYS_XDBPD$                                         XDB.XDB$RAW_LIST_T
 Name                                               VARCHAR2(4000 CHAR)
 Age                                                NUMBER
 Addr                                               XDB.XDB$XMLTYPE_REF_LIST_T
```

By default, XML schema attribute `storeVarrayAsTable` has value `true`, which means that the varray of `REF` values is stored out of line, in an intermediate table.

That is, in addition to creating the tables and types just mentioned, XML schema registration also creates the intermediate table that stores the list of `REF` values. This table has a system-generated name, but you can rename it. That can be useful, for example, in order to create an index on it.

Example 9–10 Renaming an Intermediate Table of REF Values

```
DECLARE
  gen_name VARCHAR2 (4000);
BEGIN
  SELECT TABLE_NAME INTO gen_name FROM USER_NESTED_TABLES
    WHERE PARENT_TABLE_NAME = 'EMP_TAB';
  EXECUTE IMMEDIATE 'RENAME "' || gen_name || '"TO emp_tab_reflist';
END;
/

DESCRIBE emp_tab_reflist
 Name                      Null?    Type
 ------------------------- -------- ----------------
 COLUMN_VALUE                       REF OF XMLTYPE
```

Example 9–11 shows a query that selects the names of all San Francisco-based employees and the streets in which they live. The example queries the address table on element `City`, and joins back with the employee table. The explain-plan fragment shown indicates a join between tables `emp_tab_reflist` and `emp_tab`.

Example 9–11 XPath Rewrite for an Out-Of-Line Collection

```
SELECT em.name, ad.street
  FROM emp_tab,
       XMLTable(XMLNAMESPACES ('http://www.oracle.com/emp.xsd' AS "x"),
                '/x:Employee' PASSING OBJECT_VALUE
                COLUMNS name   VARCHAR2(20) PATH 'Name') em,
       XMLTable(XMLNAMESPACES ('http://www.oracle.com/emp.xsd' AS "x"),
                '/x:Employee/Addr' PASSING OBJECT_VALUE
                COLUMNS street VARCHAR2(20) PATH 'Street',
                        city   VARCHAR2(20) PATH 'City') ad
  WHERE ad.city = 'San Francisco';

NAME                 STREET
-------------------- --------------------
Abe Bee              A Street
Eve Fong             E Street
George Hu            G Street
Iris Jones           I Street
Karl Luomo           K Street
Marina Namur         M Street
```

```
                Omar Pinano          O Street
                Quincy Roberts       Q Street

                8 rows selected.

|   4 |    TABLE ACCESS FULL          | EMP_TAB_REFLIST |     32 |    640 |      2   (0)| 00:00:01 |
|   5 |   TABLE ACCESS BY INDEX ROWID| EMP_TAB         |      1 |     29 |      1   (0)| 00:00:01 |
|*  6 |    INDEX UNIQUE SCAN          | SYS_C005567     |      1 |        |      0   (0)| 00:00:01 |
```

To improve performance you can create an index on the `REF` values in the intermediate table, `emp_tab_reflist`. This lets Oracle XML DB query the address table, obtain an object reference (`REF`) to the relevant row, join it with the intermediate table storing the list of `REF` values, and join that table back with the employee table.

You can create an index on `REF` values only if the `REF` is *scoped* or has a referential constraint. A scoped `REF` column stores pointers only to objects in a particular table. The `REF` values in table `emp_tab_reflist` point only to objects in table `addr_tab`, so you can create a scope constraint and an index on the `REF` column, as shown in Example 9–12.

Example 9–12 XPath Rewrite for an Out-Of-Line Collection, with Index on REFs

```
ALTER TABLE emp_tab_reflist ADD SCOPE FOR (COLUMN_VALUE) IS addr_tab;
CREATE INDEX reflist_idx ON emp_tab_reflist (COLUMN_VALUE);
```

The explain-plan fragment for the same query as in Example 9–11 shows that index `reflist_idx` is picked up.

```
|   4 |    TABLE ACCESS BY INDEX ROWID| EMP_TAB_REFLIST |      1 |     20 |      1   (0)| 00:00:01 |
|*  5 |     INDEX RANGE SCAN          | REFLIST_IDX     |      1 |        |      0   (0)| 00:00:01 |
|   6 |   TABLE ACCESS BY INDEX ROWID | EMP_TAB         |        |        |             |          |
|*  7 |    INDEX UNIQUE SCAN          | SYS_C005567     |      1 |        |      0   (0)| 00:00:01 |
```

In cases where the more selective predicates in the query are on the employee table, you might want to set XML schema attribute `storeVarrayAsTable` to `false`, in order to store the varray of `REF` values in line in table `emp_tab`. Storing the varray in line effectively forces any query involving the two tables `emp_tab` and `addr_tab` to always be driven from `emp_tab`. There is then no way to efficiently join back from the address table. This approach is inappropriate when the number of employees is large, because it involves a full table scan of table `emp_tab`, which can be expensive.

Partitioning XMLType Tables and Columns Stored Object-Relationally

This section is about XML data that is stored using structured storage, that is, object-relationally.

When you partition an `XMLType` table or a table with an `XMLType` column using list, range, or hash partitioning, any ordered collection tables (OCTs) or out-of-line tables within the data are automatically partitioned accordingly, by default.

This **equipartitioning** means that the partitioning of an OCT or an out-of-line table follows the partitioning scheme of its parent (base) table. There is a corresponding child-table partition for each partition of the base table. A child element is stored in the child-table partition that corresponds to the base-table partition of its parent element.

Storage attributes for a base table partition are, by default, also used for the corresponding child-table partitions. You can override these storage attributes for a given child-table partition.

Similarly, by default, the name of an OCT partition is the same as its base (parent) table, but you can override this behavior by specifying the name to use. The name of an out-of-line table partition is always the same as the partition of its parent-table (which could be a base table or an OCT).

Note:

- Equipartitioning of `XMLType` data stored object-relationally is not available in releases prior to Oracle Database 11*g* Release 1 (11.1).
- Equipartitioning of `XMLType` data that is stored out of line is not available in releases prior to Oracle Database 11*g* Release 2 (11.2.0.2). Starting with that release, out-of-line tables are not shared: You cannot create two top-level tables that are based on the same XML schema, if that schema specifies an out-of-line table.

You can prevent partitioning of OCTs by specifying the keyword `GLOBAL` in a `CREATE TABLE` statement. (Starting with Oracle Database 11g Release 1 (11.1), the default behavior uses keyword `LOCAL`). For information about converting a non-partitioned collection table to a partitioned collection table, see *Oracle Database VLDB and Partitioning Guide*.

You can prevent partitioning of out-of-line tables, and thus allow out-of-line sharing, by turning on event 31178 with level 0x200:

```
ALTER SESSION SET EVENTS '31178 TRACE NAME CONTEXT FOREVER, LEVEL
0x200'
```

See Also:

- "Controlling How Collections Are Stored for Object-Relational XMLType Storage" on page 3-19 for information about OCTs
- *Oracle Database SQL Language Reference* for information about creating tables with partitions using keywords `GLOBAL` and `LOCAL`

Examples of Partitioning XMLType Data

You can specify partitioning information for an `XMLType` base table in two ways:

- During XML schema registration, using XML Schema annotation `xdb:tableProps`
- During table creation using `CREATE TABLE`

Example 9–13 and Example 9–14 illustrate this. These two examples have exactly the same effect. They partition the base `purchaseorder` table using the `Reference` element to specify ranges. They equipartition the child table of line items with respect to the base table.

Example 9–13 shows element `PurchaseOrder` from the purchase-order XML schema, annotated to partition the base table and its child table of line items.

Example 9–13 Specifying Partitioning Information During XML Schema Registration

```
<xs:element name="PurchaseOrder" type="PurchaseOrderType"
            xdb:defaultTable="PURCHASEORDER"
```

```
xdb:tableProps =
  "VARRAY XMLDATA.LINEITEMS.LINEITEM
     STORE AS TABLE lineitem_table
       ((PRIMARY KEY (NESTED_TABLE_ID, SYS_NC_ARRAY_INDEX$)))
       PARTITION BY RANGE (XMLDATA.Reference)
         (PARTITION p1 VALUES LESS THAN (1000)
            VARRAY XMLDATA.LINEITEMS.LINEITEM
              STORE AS TABLE lineitem_p1 (STORAGE (MINEXTENTS 13)),
          PARTITION p2 VALUES LESS THAN (2000)
            VARRAY XMLDATA.LINEITEMS.LINEITEM
              STORE AS TABLE lineitem_p2 (STORAGE (MINEXTENTS 13)))"/>
```

Example 9–14 specifies the same partitioning as in Example 9–13, but it does so during the creation of the base table `purchaseorder`.

Example 9–14 Specifying Partitioning Information During Table Creation

```
CREATE TABLE purchaseorder OF XMLType
  XMLSCHEMA "http://localhost:8080/source/schemas/poSource/xsd/purchaseOrder.xsd"
  ELEMENT "PurchaseOrder"
  VARRAY "XMLDATA"."LINEITEMS"."LINEITEM" STORE AS TABLE lineitem_table
    ((PRIMARY KEY (NESTED_TABLE_ID, SYS_NC_ARRAY_INDEX$)))
    PARTITION BY RANGE (XMLDATA.Reference)
      (PARTITION p1 VALUES LESS THAN (1000)
         VARRAY "XMLDATA"."LINEITEMS"."LINEITEM" STORE AS TABLE lineitem_p1
           (STORAGE (MINEXTENTS 13)),
       PARTITION p2 VALUES LESS THAN (2000)
         VARRAY  "XMLDATA"."LINEITEMS"."LINEITEM" STORE AS TABLE lineitem_p2
           (STORAGE (MINEXTENTS 13)));
```

Example 9–13 and Example 9–14 also show how you can specify object storage options for the individual child-table partitions. In this case, the `STORAGE` clauses specify that extents of size 14M are to be allocated initially for each of the child-table partitions.

See Also:

- "Annotated Purchase-Order XML Schema, purchaseOrder.xsd" on page A-30
- Example 3–13, "Creating an XMLType Table that Conforms to an XML Schema" on page 3-28
- *Oracle Database Object-Relational Developer's Guide* for more information about partitioning object-relational data
- *Oracle Database VLDB and Partitioning Guide* for more information about partitioning

Partition Maintenance

You need not define or maintain child-table partitions manually. When you perform partition maintenance on the base (parent) table, corresponding maintenance is automatically performed on the child tables as well.

There are a few exceptions to the general rule that you perform partition maintenance only on the base table. In the following cases you perform maintenance on a child table:

- Modify the *default* physical storage attributes of a collection partition
- Modify the physical storage attributes of a collection partition

- Move a collection partition to a different segment, possibly in a different tablespace
- Rename a collection partition

For example, if you change the tablespace of a base table, that change is not cascaded to its child-table partitions. You must manually use `ALTER TABLE MODIFY PARTITION` on the child-table partitions to change their tablespace.

Other than those exceptional operations, you perform all partition maintenance on the base table only. This includes operations such as adding, dropping, and splitting a partition.

Online partition redefinition is also supported for child tables. You can copy unpartitioned child tables to partitioned child tables during online redefinition of a base table. You typically specify parameter values `copy_indexes => 0` and `copy_constraints => false` for PL/SQL procedure `DBMS_REDEFINITION.copy_table_dependents`, to protect the indexes and constraints of the newly defined child tables.

See Also:

- *Oracle Database SQL Language Reference* for information about SQL statement `ALTER TABLE`
- *Oracle Database PL/SQL Packages and Types Reference* for information about online partition redefinition using PL/SQL package `DBMS_REDEFINITION`

Fully Qualified XML Schema URLs

By default, XML schema URLs are referenced within the scope of the current database user. XML schema URLs are first resolved as the names of *local* XML schemas owned by the current user.

- If there are no such XML schemas, then they are resolved as names of *global* XML schemas.
- If there are no *global* XML schemas either, then Oracle XML DB raises an error.

To permit explicit reference to particular XML schemas, Oracle XML DB supports the notion of *fully qualified* XML schema URLs. The name of the database user owning the XML schema is specified as part of the XML schema URL. Fully qualified XML schema URLs belong to the Oracle XML DB namespace:

```
http://xmlns.oracle.com/xdb/schemas/<database-user>/<schemaURL-minus-protocol>
```

For example, suppose there is a registered global XML schema with the URL `http://www.example.com/po.xsd`, and user `QUINE` has a local registered XML schema with the same URL. Another user can reference the schema owned by `QUINE` as follows using this fully qualified XML Schema URL:

```
http://xmlns.oracle.com/xdb/schemas/QUINE/www.example.com/po.xsd
```

The fully qualified URL for the global XML schema is:

```
http://xmlns.oracle.com/xdb/schemas/PUBLIC/www.example.com/po.xsd
```

See Also: "Local and Global XML Schemas" on page 7-14

Mapping XML Fragments to Large Objects (LOBs)

You can specify the SQL data type to use for a complex element as being `CLOB` or `BLOB`. In Figure 9–2, for example, an entire XML fragment is stored in a LOB attribute. This is useful when parts of an XML document are typically retrieved and stored as whole, and are seldom queried. By storing XML fragments as LOBs, you can save on parsing, decomposition, and recomposition overheads.

In Example 9–15, the XML schema defines element `Addr` using the annotation `SQLType = "CLOB"`:

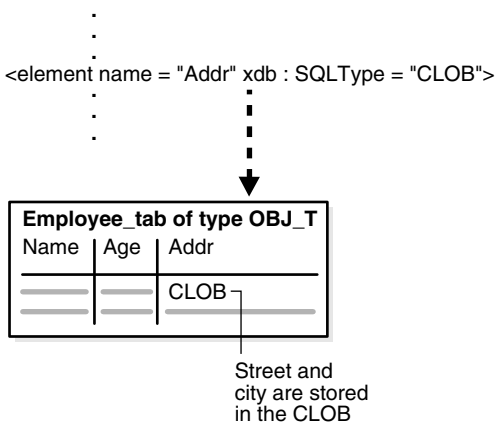
Example 9–15 Oracle XML DB XML Schema: Mapping complexType XML Fragments to LOBs

```
DECLARE
  doc VARCHAR2(3000) :=
    '<schema xmlns="http://www.w3.org/2001/XMLSchema"
             targetNamespace="http://www.oracle.com/emp.xsd"
             xmlns:emp="http://www.oracle.com/emp.xsd"
             xmlns:xdb="http://xmlns.oracle.com/xdb">
       <complexType name="Employee" xdb:SQLType="OBJ_T">
         <sequence>
           <element name="Name" type="string"/>
           <element name="Age" type="decimal"/>
           <element name="Addr" xdb:SQLType="CLOB">
             <complexType >
               <sequence>
                 <element name="Street" type="string"/>
                 <element name="City" type="string"/>
               </sequence>
             </complexType>
           </element>
         </sequence>
       </complexType>
     </schema>';
BEGIN
  DBMS_XMLSCHEMA.registerSchema(
    SCHEMAURL => 'http://www.oracle.com/PO.xsd',
    SCHEMADOC => doc);
END;
```

When registering this XML schema, Oracle XML DB generates the following types and `XMLType` tables:

```
CREATE TYPE obj_t AS OBJECT(SYS_XDBPD$ XDB.XDB$RAW_LIST_T,
                            Name VARCHAR2(4000),
                            Age NUMBER,
                            Addr CLOB);
```

Figure 9–2 Mapping complexType XML Fragments to Character Large Objects (CLOB)



complexType Extensions and Restrictions in Oracle XML DB

In XML Schema, `complexType` values are declared based on `complexContent` and `simpleContent`.

- `simpleContent` is declared as an extension of `simpleType`.
- `complexContent` is declared as one of the following:
  - Base type
  - `complexType` extension
  - `complexType` restriction

This section describes the Oracle XML DB extensions and restrictions to `complexType`.

complexType Declarations in XML Schema: Handling Inheritance

For `complexType`, Oracle XML DB handles inheritance in the XML schema as follows:

- For complex types declared to *extend* other complex types, the SQL type corresponding to the base type is specified as the supertype for the current SQL type. Only the additional attributes and elements declared in the sub-complextype are added as attributes to the sub-object-type.
- For complex types declared to *restrict* other complex types, the SQL type for the sub-complex type is set to be the same as the SQL type for its base type. This is because SQL does not support restriction of object types through the inheritance mechanism. Any constraints are imposed by the restriction in XML schema.

Example 9–16 shows the registration of an XML schema that defines a base `complexType` `Address` and two extensions `USAddress` and `IntlAddress`.

Example 9–16 XML Schema Inheritance: complexContent as an Extension of complexTypes

```
DECLARE
  doc VARCHAR2(3000) :=
    '<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
                xmlns:xdb="http://xmlns.oracle.com/xdb">
       <xs:complexType name="Address" xdb:SQLType="ADDR_T">
```

```
          <xs:sequence>
            <xs:element name="street" type="xs:string"/>
            <xs:element name="city" type="xs:string"/>
          </xs:sequence>
        </xs:complexType>
        <xs:complexType name="USAddress" xdb:SQLType="USADDR_T">
          <xs:complexContent>
            <xs:extension base="Address">
              <xs:sequence>
                <xs:element name="zip" type="xs:string"/>
              </xs:sequence>
            </xs:extension>
          </xs:complexContent>
        </xs:complexType>
        <xs:complexType name="IntlAddress" final="#all" xdb:SQLType="INTLADDR_T">
          <xs:complexContent>
            <xs:extension base="Address">
              <xs:sequence>
                <xs:element name="country" type="xs:string"/>
              </xs:sequence>
            </xs:extension>
          </xs:complexContent>
        </xs:complexType>
      </xs:schema>';
BEGIN
  DBMS_XMLSCHEMA.registerSchema(
    SCHEMAURL => 'http://www.oracle.com/PO.xsd',
    SCHAMEDOC => doc);
END;
```

Note: Type `intladdr_t` is created as a *final* type because the corresponding `complexType` specifies the "`final`" attribute. By default, all `complexTypes` can be extended and restricted by other types, so all SQL object types are created as types that are *not* final.

```
CREATE TYPE addr_t AS OBJECT(SYS_XDBPD$ XDB.XDB$RAW_LIST_T,
                             "street" VARCHAR2(4000),
                             "city" VARCHAR2(4000)) NOT FINAL;
CREATE TYPE usaddr_t UNDER addr_t ("zip" VARCHAR2(4000)) NOT FINAL;
CREATE TYPE intladdr_t UNDER addr_t ("country" VARCHAR2(4000)) FINAL;
```

Example 9–17 shows the registration of an XML schema that defines a base `complexType` `Address` and a restricted type `LocalAddress` that prohibits the specification of `country` attribute.

Example 9–17 Inheritance in XML Schema: Restrictions in complexTypes

```
DECLARE
  doc varchar2(3000) :=
    '<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
                xmlns:xdb="http://xmlns.oracle.com/xdb">
       <xs:complexType name="Address" xdb:SQLType="ADDR_T">
         <xs:sequence>
           <xs:element name="street" type="xs:string"/>
           <xs:element name="city" type="xs:string"/>
           <xs:element name="zip" type="xs:string"/>
           <xs:element name="country" type="xs:string" minOccurs="0"
                       maxOccurs="1"/>
```

```
        </xs:sequence>
      </xs:complexType>
      <xs:complexType name="LocalAddress" xdb:SQLType="USADDR_T">
        <xs:complexContent>
          <xs:restriction base="Address">
            <xs:sequence>
              <xs:element name="street" type="xs:string"/>
              <xs:element name="city" type="xs:string"/>
              <xs:element name="zip" type="xs:string"/>
              <xs:element name="country" type="xs:string"
                          minOccurs="0" maxOccurs="0"/>
            </xs:sequence>
          </xs:restriction>
        </xs:complexContent>
      </xs:complexType>
    </xs:schema>';
BEGIN
  DBMS_XMLSCHEMA.registerSchema(
    SCHEMAURL => 'http://www.oracle.com/PO.xsd',
    SCHEMADOC => doc);
END;
```

Because SQL inheritance does not support a notion of restriction, the SQL data type corresponding to a restricted `complexType` is a empty subtype of the parent object type. For the XML schema of Example 9–17, Oracle XML DB generates the following SQL types:

```
CREATE TYPE addr_t AS OBJECT (SYS_XDBPD$ XDB.XDB$RAW_LIST_T,
                              "street"   VARCHAR2(4000),
                              "city"     VARCHAR2(4000),
                              "zip"      VARCHAR2(4000),
                              "country"  VARCHAR2(4000)) NOT FINAL;
CREATE TYPE usaddr_t UNDER addr_t;
```

Mapping complexType: simpleContent to Object Types

A complex type based on a `simpleContent` declaration is mapped to an object type with attributes corresponding to the XML attributes and an extra `SYS_XDBBODY$` attribute corresponding to the body value. The data type of the body attribute is based on `simpleType` which defines the body type.

Example 9–18 XML Schema complexType: Mapping complexType to simpleContent

```
DECLARE
  doc VARCHAR2(3000) :=
    '<schema xmlns="http://www.w3.org/2001/XMLSchema"
             targetNamespace="http://www.oracle.com/emp.xsd"
             xmlns:emp="http://www.oracle.com/emp.xsd"
             xmlns:xdb="http://xmlns.oracle.com/xdb">
      <complexType name="name" xdb:SQLType="OBJ_T">
        <simpleContent>
          <restriction base="string">
          </restriction>
        </simpleContent>
      </complexType>
    </schema>';
BEGIN
  DBMS_XMLSCHEMA.registerSchema(
    SCHEMAURL => 'http://www.oracle.com/emp.xsd',
    SCHEMADOC => doc);
```

```
END;
```

For the XML schema of Example 9–18, Oracle XML DB generates the following type:

```
CREATE TYPE obj_t AS OBJECT(SYS_XDBPD$ XDB.XDB$RAW_LIST_T,
                            SYS_XDBBODY$ VARCHAR2(4000));
```

Mapping complexType: any and anyAttribute

Oracle XML DB maps the element declaration, `any`, and the attribute declaration, `anyAttribute`, to `VARCHAR2` attributes (or optionally to Large Objects (LOBs)) in the created object type. The object attribute stores the text of the XML fragment that matches the `any` declaration.

- The `namespace` attribute can be used to restrict the contents so that they belong to a specified namespace.
- The `processContents` attribute within the `any` element declaration, indicates the level of validation required for the contents matching the `any` declaration.

The code in Example 9–19 declares an `any` element and maps it to the column `SYS_XDBANY$`, in object type `obj_t`. It also declares that attribute `processContents` does not validate contents that match the `any` declaration.

Example 9–19 XML Schema: Mapping complexType to any/anyAttribute

```
DECLARE
  doc VARCHAR2(3000) :=
    '<schema xmlns="http://www.w3.org/2001/XMLSchema"
             targetNamespace="http://www.oracle.com/any.xsd"
             xmlns:emp="http://www.oracle.com/any.xsd"
             xmlns:xdb="http://xmlns.oracle.com/xdb">
       <complexType name="Employee" xdb:SQLType="OBJ_T">
         <sequence>
           <element name="Name" type="string"/>
           <element name="Age" type="decimal"/>
           <any namespace="http://www/w3.org/2001/xhtml"
                processContents="skip"/>
         </sequence>
       </complexType>
     </schema>';
BEGIN
  DBMS_XMLSCHEMA.registerSchema(
    SCHEMAURL => 'http://www.oracle.com/emp.xsd',
    SCHEMADOC => doc);
END;
```

For the XML schema of Example 9–19, Oracle XML DB generates the following type:

```
CREATE TYPE obj_t AS OBJECT(SYS_XDBPD$ XDB.XDB$RAW_LIST_T,
                            Name VARCHAR2(4000),
                            Age NUMBER,
                            SYS_XDBANY$ VARCHAR2(4000));
```

XML Schema: Working with Circular and Cyclical Dependencies

The W3C XML Schema Recommendation lets `complexTypes` and global elements contain recursive references. For example, a `complexType` definition can *contain* an element based on that same `complexType`, or a global element can contain a reference to itself. In both cases the reference can be direct or indirect. This kind of structure

allows for instance documents where the element in question can appear an infinite number of times in a recursive hierarchy.

Example 9–20 An XML Schema with Circular Dependency

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:xdb="http://xmlns.oracle.com/xdb"
           elementFormDefault="qualified" attributeFormDefault="unqualified">
  <xs:element name="person" type="personType" xdb:defaultTable="PERSON_TABLE"/>
  <xs:complexType name="personType" xdb:SQLType="PERSON_T">
    <xs:sequence>
      <xs:element name="descendant" type="personType" minOccurs="0"
                  maxOccurs="unbounded" xdb:SQLName="DESCENDANT"
                  xdb:defaultTable="DESCENDANT_TABLE"/>
    </xs:sequence>
    <xs:attribute name="personName" use="required" xdb:SQLName="PERSON_NAME">
      <xs:simpleType>
        <xs:restriction base="xs:string">
          <xs:maxLength value="20"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:attribute>
  </xs:complexType>
</xs:schema>
```

The XML schema in Example 9–20 includes a circular dependency. The `complexType` `personType` consists of a `personName` attribute and a collection of `descendant` elements. The `descendant` element is defined as being of type `personType`.

For Circular XML Schema Dependencies Set Parameter GENTABLES to TRUE

Oracle XML DB supports XML schemas that define this kind of structure. It does this by detecting the cycles, breaking them, and storing the recursive elements as rows in a separate `XMLType` table that is created during XML schema registration.

Consequently, it is important to ensure that parameter `GENTABLES` is set to `TRUE` when registering an XML schema that defines this kind of structure. The name of the table used to store the recursive elements can be specified by adding an `xdb:defaultTable` annotation to the XML schema.

complexType Declarations XML Schema: Handling Cycles

SQL object types do not allow cycles. Cycles in an XML schema are broken while generating the object types, by introducing a `REF` attribute at the point where the cycle would be completed. Thus, part of the data is stored out of line, but it is still retrieved as part of the parent XML document.

> **Note:** Starting with Oracle Database 11g Release 2 (11.2.0.2), you can create only *one* `XMLType` table that uses an XML schema that results in `an out-of-line table`. An error is raised if you try to create a second table that uses the same XML schema.

XML schemas permit cycling between definitions of complex types. Figure 9–3 shows this, where the definition of complex type `CT1` can reference another complex type `CT2`, whereas the definition of `CT2` references the first type `CT1`.

XML schemas permit cycles among definitions of complex types. Example 9–21 creates a cycle of length two:

Example 9–21 XML Schema: Cycling Between complexTypes

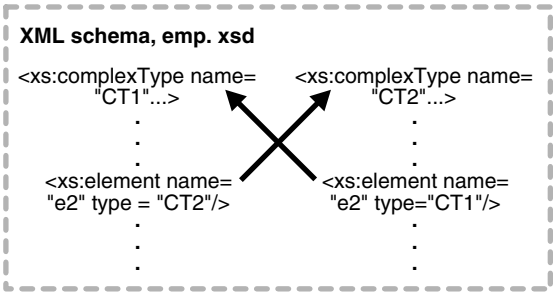
```
DECLARE
  doc VARCHAR2(3000) :=
    '<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
                xmlns:xdb="http://xmlns.oracle.com/xdb">
       <xs:complexType name="CT1" xdb:SQLType="CT1">
         <xs:sequence>
           <xs:element name="e1" type="xs:string"/>
           <xs:element name="e2" type="CT2"/>
         </xs:sequence>
       </xs:complexType>
       <xs:complexType name="CT2" xdb:SQLType="CT2">
         <xs:sequence>
           <xs:element name="e1" type="xs:string"/>
           <xs:element name="e2" type="CT1"/>
         </xs:sequence>
       </xs:complexType>
     </xs:schema>';
BEGIN
  DBMS_XMLSCHEMA.registerSchema(
    SCHEMAURL => 'http://www.oracle.com/emp.xsd',
    SCHEMADOC => doc);
END;
```

SQL types do not allow cycles in type definitions. However, they do support **weak cycles**, that is, cycles involving `REF` (reference) object attributes. Cyclic XML schema definitions are mapped to SQL object types in such a way that cycles are avoided by forcing `SQLInline = "false"` at the appropriate points. This creates a weak SQL cycle.

For the XML schema of Example 9–21, Oracle XML DB generates the following types:

```
CREATE TYPE ct1 AS OBJECT (SYS_XDBPD$  XDB.XDB$RAW_LIST_T,
                           "e1"        VARCHAR2(4000),
                           "e2"        REF XMLType) NOT FINAL;
CREATE TYPE ct2 AS OBJECT (SYS_XDBPD$  XDB.XDB$RAW_LIST_T,
                           "e1"        VARCHAR2(4000),
                           "e2"        CT1) NOT FINAL;
```

Figure 9–3 Cross Referencing Between Different complexTypes in the Same XML Schema



Another example of a cyclic complex type involves the declaration of the complex type that refers to itself. In Example 9–22, type `SectionT` does this.

Example 9–22 XML Schema: Cycling Between complexTypes, Self-Reference

```
DECLARE
  doc VARCHAR2(3000) :=
    '<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
                xmlns:xdb="http://xmlns.oracle.com/xdb">
       <xs:complexType name="SectionT" xdb:SQLType="SECTION_T">
         <xs:sequence>
           <xs:element name="title" type="xs:string"/>
           <xs:choice maxOccurs="unbounded">
             <xs:element name="body" type="xs:string"
                         xdb:SQLCollType="BODY_COLL"/>
             <xs:element name="section" type="SectionT"/>
           </xs:choice>
         </xs:sequence>
       </xs:complexType>
     </xs:schema>';
BEGIN
  DBMS_XMLSCHEMA.registerSchema(
    SCHEMAURL => 'http://www.oracle.com/section.xsd',
    SCHEMADOC => doc);
END;
```

For the XML schema of Example 9–22, Oracle XML DB generates the following types:

```
CREATE TYPE body_coll AS VARRAY(32767) OF VARCHAR2(4000);
CREATE TYPE section_t AS OBJECT (SYS_XDBPD$  XDB.XDB$RAW_LIST_T,
                                 "title"     VARCHAR2(4000),
                                 "body"      BODY_COLL,
                                 "section"   XDB.XDB$REF_LIST_T) NOT FINAL;
```

Note: In Example 9–22, object attribute `section` is declared as a varray of `REF` references to `XMLType` instances. Because there can be more than one occurrence of embedded sections, the attribute is a varray. It is a varray of `REF` references to `XMLType` instances, to avoid forming a cycle of SQL objects.

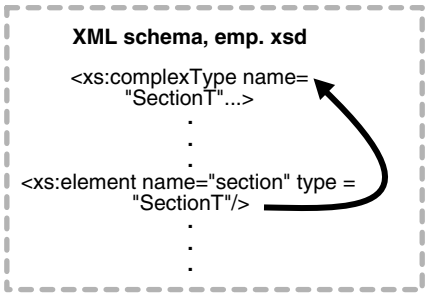
How a complexType Can Reference Itself

Assume that your XML schema, identified by "`http://www.oracle.com/PO.xsd`", has been registered. An `XMLType` table, `purchaseorder`, can then be created to store instances conforming to element `PurchaseOrder` of this XML schema, in an object-relational format:

```
CREATE TABLE purchaseorder OF XMLType
   ELEMENT "http://www.oracle.com/PO.xsd#PurchaseOrder";
```

Figure 9–4 illustrates schematically how a `complexType` can reference itself.

Figure 9–4 Self-Referencing Complex Type within an XML Schema



See Also: "Cyclical References Among XML Schemas" on page 9-22

Hidden columns are created that correspond to the object type to which the `PurchaseOrder` element has been mapped. In addition, an `XMLEXTRA` object column is created, to store top-level instance data such as namespace declarations. `XMLEXTRA` is reserved for internal use.

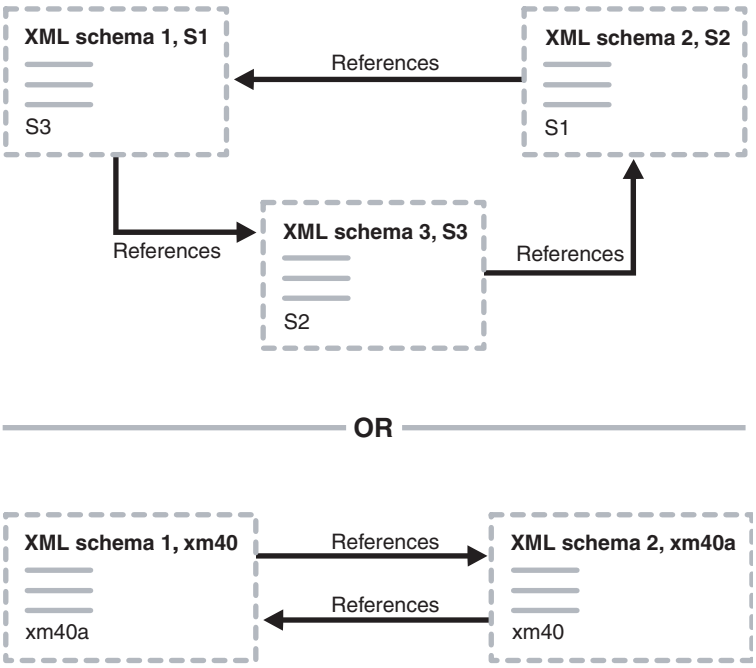
Cyclical References Among XML Schemas

XML schemas can depend on each other in such a way that they cannot be registered one after the other in the usual manner. Illustrations of such XML schemas follow in Figure 9–5.

In the top half of the illustration, an example of indirect cyclical references between three XML schemas is shown.

In the bottom half of the illustration, an example of cyclical dependencies between two XML schemas is shown. The details of this simpler example are presented first.

Figure 9–5 Cyclical References Between XML Schemas



An XML schema that includes another XML schema cannot be created if the included XML schema does not exist. The registration of XML schema `xm40.xsd` in Example 9–23 fails, if `xm40a.xsd` does not exist.

Example 9–23 An XML Schema that Includes a Non-Existent XML Schema

```
BEGIN DBMS_XMLSCHEMA.registerSchema(
  SCHEMAURL => 'xm40.xsd',
  SCHEMADOC => '<schema xmlns="http://www.w3.org/2001/XMLSchema"
                        xmlns:my="xm40"
                        targetNamespace="xm40">
                  <include schemaLocation="xm40a.xsd"/>
                  <!-- Define a global complextype here -->
                  <complexType name="Company">
                    <sequence>
                      <element name="Name" type="string"/>
                      <element name="Address" type="string"/>
                    </sequence>
                  </complexType>
                  <!-- Define a global element depending on included schema -->
                  <element name="Emp" type="my:Employee"/>
                </schema>',
  LOCAL     => TRUE,
  GENTYPES  => TRUE,
  GENTABLES => TRUE);
END;
/
```

XML schema `xm40.xsd` can, however, be created if you specify option `FORCE => TRUE`, as in Example 9–24:

Example 9–24 Using the FORCE Option to Register XML Schema xm40.xsd

```
BEGIN DBMS_XMLSCHEMA.registerSchema(
  SCHEMAURL => 'xm40.xsd',
  SCHEMADOC => '<schema xmlns="http://www.w3.org/2001/XMLSchema"
                        xmlns:my="xm40"
                        targetNamespace="xm40">
                  <include schemaLocation="xm40a.xsd"/>
                  <!-- Define a global complextype here -->
                  <complexType name="Company">
                    <sequence>
                      <element name="Name" type="string"/>
                      <element name="Address" type="string"/>
                    </sequence>
                  </complexType>
                  <!-- Define a global element depending on included schema -->
                  <element name="Emp" type="my:Employee"/>
                </schema>',
  LOCAL     => TRUE,
  GENTYPES  => TRUE,
  GENTABLES => TRUE,
  FORCE     => TRUE);
END;
/
```

However, an attempt to use XML schema `xm40.xsd`, as in Example 9–25, fails.

Example 9–25 Trying to Create a Table Using a Cyclic XML Schema

```
CREATE TABLE foo OF XMLType XMLSCHEMA "xm40.xsd" ELEMENT "Emp";
```

If you register `xm40a.xsd` using the `FORCE` option, as in Example 9–26, then both XML schemas can be used, as shown by the `CREATE TABLE` statements.

Example 9–26 Using the FORCE Option to Register XML Schema xm40a.xsd

```
BEGIN DBMS_XMLSCHEMA.registerSchema(
  SCHEMAURL => 'xm40a.xsd',
  SCHEMADOC => '<schema xmlns="http://www.w3.org/2001/XMLSchema"
                        xmlns:my="xm40"
                        targetNamespace="xm40">
                  <include schemaLocation="xm40.xsd"/>
                  <!-- Define a global complextype here -->
                  <complexType name="Employee">
                    <sequence>
                      <element name="Name" type="string"/>
                      <element name="Age" type="positiveInteger"/>
                      <element name="Phone" type="string"/>
                    </sequence>
                  </complexType>
                  <!-- Define a global element depending on included schema -->
                  <element name="Comp" type="my:Company"/>
                </schema>',
  LOCAL     => TRUE,
  GENTYPES  => TRUE,
  GENTABLES => TRUE,
  FORCE     => TRUE);
END;
/

CREATE TABLE foo  OF XMLType XMLSCHEMA "xm40.xsd"  ELEMENT "Emp";
CREATE TABLE foo2 OF XMLType XMLSCHEMA "xm40a.xsd" ELEMENT "Comp";
```

Thus, to register these XML schemas, which depend on each other, you must use the `FORCE` parameter in `DBMS_XMLSCHEMA.registerSchema` for each schema, as follows:

1. Register `xm40.xsd` with `FORCE` mode set to `TRUE`:

   ```
   DBMS_XMLSCHEMA.registerSchema("xm40.xsd", "<schema ...", ..., FORCE => TRUE)
   ```

   At this point, `xm40.xsd` *cannot* be used.

2. Register `xm40a.xsd` in `FORCE` mode set to `TRUE`:

   ```
   DBMS_XMLSCHEMA.registerSchema("xm40a.xsd", "<schema ...", ..., FORCE => TRUE)
   ```

   The second operation automatically compiles `xm40.xsd` and makes both XML schemas usable.

Support for Recursive Schemas

Storing a `REF` to a recursive structure that is in an out-of-line table has the disadvantage that XPath queries against such documents cannot easily be rewritten, because it is not known at compile time how deep the structure might be. To enable rewrite of such XPath queries, a `DOCID` column is used to store a pointer back to the root document in any recursive structure, enabling some queries to use the out-of-line tables directly and join back using this column.

Example 9–27 shows a recursive XML schema.

Example 9–27 Recursive XML Schema

```
<schema targetNamespace="AbcNS" xmlns="http://www.w3.org/2001/XMLSchema"
          xmlns:abc="AbcNS" xmlnm:xdb="http://xmlns.oracle.com.xdb">
  <element name="AbcCode" xdb:defaultTable="ABCCODETAB">
    <complexType>
      <sequence>
        <element ref="abc:AbcSection"/>
      </sequence>
    </complexType>
  </element>

  <element name="AbcSection">
    <complexType>
      <sequence>
        <element name="ID" type="integer"/>
        <element name="Contents" type="string"/>
        <element ref="abc:AbcSection"/>
      </sequence>
    </complexType>
  </element>
</schema>
```

A **document-correlated recursive query** is a query using a SQL function that accepts an XPath or XQuery expression and an `XMLType` instance, where that XPath or XQuery expression contains `'//'`. A document-correlated recursive query can be *rewritten* if it can be determined at query compilation time that both of the following conditions are met:

- All fragments of the `XMLType` instance that are targeted by the XPath or XQuery expression reside in a single out-of-line table.
- No other fragments of the `XMLType` instance reside in the same out-of-line table.

The rewritten query is a join with the out-of-line table, based on the `DOCID` column.

Other queries with `'//'` can also be rewritten. For example, if there are several `address` elements, all of the same type, in different sections of a schema, and you often query all `address` elements with `'//'`, not caring about their specific location in the document, rewrite can occur.

During schema registration, an additional `DOCID` column is generated for out-of-line `XMLType` tables This column stores the `OID` (Object Identifier Values) of the document, that is, the root element. This column is automatically populated when data is inserted in the tables. You can export tables containing `DOCID` columns and import them later.

Sharing defaultTable Among Common Out-Of-Line Elements

The out-of-line elements of the same qualified name (namespace and local name) and same type are stored in the same default table. As a special case, you can store the root element of a cyclic element structure out of line also, and in the same table as the sub-elements (if the root element is stored out of line also).

Both of the elements sharing the default table must be out-of-line elements, that is, the default table for an out-of-line element cannot be the same as the table for a top-level element. To do this, specify `xdb:SQLInline = "false"` for both elements and specify an explicit `xdb:defaultTable` attribute having the same value in both elements.

Example 9–28 shows an XML schema with an out-of-line table that is stored in ABCSECTIONTAB.

Example 9–28 Out-of-line Table

```
<schema targetNamespace="AbcNS" xmlns="http://www.w3.org/2001/XMLSchema"
           xmlns:abc="AbcNS" xmlns:xdb="http://xmlns.oracle.com/xdb">
  <element name="AbcCode" xdb:defaultTable="ABCCODETAB">
    <complexType>
      <sequence>
        <element ref="abc:AbcSection" xdb:SQLInline="false"/>
      </sequence>
    </complexType>
  </element>

  <element name="AbcSection" xdb:defaultTable="">
    <complexType>
      <sequence>
        <element name="ID" type="integer"/>
        <element name="Contents" type="string"/>
        <element ref="abc:AbcSection" xdb:SQLInline="false"
                 xdb:defaultTable="ABCSECTIONTAB"/>
      </sequence>
    </complexType>
  </element>
</schema>
```

Both of the out-of-line AbcSection elements in Example 9–28 share the same default table, ABCSECTIONTAB.

However, the Example 9–29 illustrates *invalid* default table sharing: recursive elements (XyZSection) do not share the same out-of-line table.

Example 9–29 Invalid Default Table Sharing

```
 <schema targetNamespace="XyzNS" xmlns="http://www.w3.org/2001/XMLSchema"
         xmlns:xyz="XyzNS" xmlns:xdb="http://xmlns.oracle.com/xdb">
   <element name="XyzCode" xdb:defaultTable="XYZCODETAB">
   <complexType>
   <sequence>
      <element name="CodeNumber" type="integer" minOccurs="0"/>
      <element ref="xyz:XyzChapter" xdb:SQLInline="false"/>
      <element ref="xyz:XyzPara" xdb:SQLInline="false" />
   </sequence>
   </complexType>
   </element>

    <element name="XyzChapter" xdb:defaultTable="XYZCHAPTAB">
     <complexType>
     <sequence>
         <element name="Title" type="string"/>
         <element ref="xyz:XyzSection" xdb:SQLInline="false"
                  xdb:defaultTable="XYZSECTIONTAB"/>
      </sequence>
      </complexType>
    </element>

    <element name="XyzPara" xdb:defaultTable="XYZPARATAB">
     <complexType>
     <sequence>
         <element name="Title" type="string"/>
```

```
        <element ref="xyz:XyzSection" xdb:SQLInline="false"
                 xdb:defaultTable="Other_XYZSECTIONTAB"/>
      </sequence>
      </complexType>
    </element>

    <element name="XyzSection">
    <complexType>
    <sequence>
        <element name="ID" type="integer"/>
        <element name="Contents" type="string"/>
        <element ref="xyz:XyzSection" xdb:defaultTable="XYZSECTIONTAB"/>
     </sequence>
     </complexType>
    </element>
 </schema>
```

The following query cannot be rewritten.

```
SELECT XMLQuery('//XyzSection' PASSING OBJECT_VALUE RETURNING CONTENT)
  FROM xyzcode;
```

Query Rewrite when DOCID is Present

Before processing `//` XPath expressions, check to find multiple occurrences of the same element. If all occurrences under the `//` share the same `defaultTable`, then the query can be rewritten to go against that table, using the `DOCID`. If there are other occurrences of the same element under the root sharing that table, but not under `//`, then the query cannot be rewritten. For example, consider this element structure:

`<Book>` contains a `<Chapter>` and a `<Part>`. `<Part>` contains a `<Chapter>`.

Assume that both of the `<Chapter>` elements are stored out of line and they share the same default table. The query `/Book//Chapter` can be rewritten to go against the default table for the `<Chapter>` elements because all of the `<Chapter>` elements under `<Book>` share the same default table. Thus, this XPath query is a document-correlated recursive XPath query.

However, a query such as `/Book/Part//Chapter` cannot be rewritten, even though all the `<Chapter>` elements under `<Part>` share the same table, because there is another `<Chapter>` element under `<Book>`, which is the document root that also shares that table.

Consider the case where you are extracting `//AbcSection` with `DOCID` present, as in the XML schema described in Example 9–28:

```
SELECT XMLQuery('//AbcSection' PASSING OBJECT_VALUE RETURNING CONTENT)
  FROM abccodetab;
```

Both of the `AbcSection` elements are stored in the same table, `abcsectiontab`. The extraction applies to the underlying table, `abcsectiontab`.

Consider this query when `DOCID` is present:

```
SELECT XMLQuery('/AbcCode/AbcSection//AbcSection'
                PASSING OBJECT_VALUE RETURNING CONTENT)
  FROM abccodetab;
```

In both this case and the previous case, all reachable `AbcSection` elements are stored in the same out-of-line table. However, the first `AbcSection` element at `/AbcCode/AbcSection` cannot be retrieved by this query. Since the join condition is

a DOCID, which cannot distinguish between different positions in the parent document, the correct result cannot be achieved by a direct query on table abcsectiontab. In this case, query rewrite does not occur since it is not a document-correlated recursive XPath. If this top-level AbcSection were not stored out of line with the rest, then the query could be rewritten.

Disabling DOCID Column Creation

You can disable the creation of column DOCID by specifying an OPTIONS parameter when calling DBMS_XMLSCHEMA.registerSchema. This disables DOCID creation in all XMLType tables generated during schema registration.

The OPTIONS parameter is an input parameter of data type PLS_INTEGER. Its default value is 0, meaning no options are used. To inhibit generation of column DOCID, set parameter OPTIONS to DBMS_XMLSCHEMA.REGISTER_NODOCID (which is 1).

See Also: *Oracle Database PL/SQL Packages and Types Reference*

Loading and Retrieving Large Documents with Collections

Configuration file /xdbconfig.xml has parameters that control the amount of memory used by the loading operation. These let you optimize the loading process, provided the following conditions are met:

- The document is loaded using one of the following:
  - Protocols (FTP, HTTP(S), or DAV)
  - PL/SQL function DBMS_XDB.createResource
  - A SQL INSERT statement into an XMLType table (but not an XMLType column)
- The document is XML schema-based and contains large collections (elements with maxoccurs set to a large number).
- Collections in the document are stored as OCTs. This is the default behavior.

In the following situations, these optimizations are sometimes suboptimal:

- When there are triggers on the base table.
- When the base table is partitioned.
- When collections are stored out of line (applies only to SQL INSERT).

The basic idea behind this optimization is that it lets the collections be swapped into or out of the memory in bounded sizes. As an illustration of this idea consider the following example conforming to a purchase-order XML schema:

```
<PurchaseOrder>
  <LineItem itemID="1">
    ...
  </LineItem>
    .
    .
  <LineItem itemID="10240">
    ...
  </LineItem>
</PurchaseOrder>
```

The purchase-order document here contains a collection of 10240 LineItem elements. Creating the entire document in memory and then pushing it out to disk can lead to

excessive memory usage and in some instances a load failure due to inadequate system memory. To avoid that, you can create the documents in finite chunks of memory called **loadable units**.

In the example case, assume that each line item needs 1 KB of memory and that you want to use loadable units of 512 KB each. Each loadable unit then contains 512 line items, and there are approximately 20 such units. If you want the entire memory representation of the document to never exceed 2 MB, then you must ensure that at any time no more than 4 loadable units are maintained in the memory. You can use an LRU mechanism to swap out the loadable units.

By controlling the size of the loadable unit and the bound on the size of the document you can tune the memory usage and performance of the load or retrieval. Typically a larger loadable unit size translates into lesser number of disk accesses but takes up more memory. This is controlled by the parameter `xdbcore-loadableunit-size` whose default value is 16 KB. The user can indicate the amount of memory to be given to the document by setting the `xdbcore-xobmem-bound` parameter which defaults to 1 MB. The values to these parameters are specified in Kilobytes. So, the default value of `xdbcore-xobmem-bound` is 1024 and that of `xdbcore-loadableunit-size` is 16. These are soft limits that provide some guidance to the system as to how to use the memory optimally.

When the document is loaded using FTP, the pattern in which the loadable units (LU) are created and flushed to the disk is as follows:

```
No LUs
Create LU1[LineItems(LI):1-512]
LU1[LI:1-512], Create LU2[LI:513-1024]
.
.
LU1[LI:1-512],...,Create LU4[LI:1517:2028]    <-   Total memory size = 2M
Swap Out LU1[LI:1-512], LU2[LI:513-1024],...,LU4[LI:1517-2028], Create
LU5[LI:2029-2540]
Swap Out LU2[LI:513-1024], LU3, LU4, LU5, Create LU6[LI:2541-2052]
.
.
.
Swap Out LU16, LU17, LU18, LU10, Create LU20[LI:9729-10240]
Flush LU17,LU18,LU19,LU20
```

Guidelines for Setting xdbcore Parameters

Typically, if you have 1 GB of addressable PG then give about 1/10th of PGA to the document. Set `xobcore-xobmem-bound` to 1/10 of addressable PGA, which is 100M. During full document retrievals and loads, `xdbcore-loadableunit-size` should be as close to `xobcore-xobmem-bound` as possible.

Start by setting `xdbcore-loadableunit-size` to half the value of `xdbcore-xobmem-bound` (50 MB). Then try to load the document.

If you run out of memory then lower `xdbcore-xobmem-bound` and set `xdbcore-loadableunot-size` to half of its value. Continue this way until the documents load successfully.

If the load succeeds then try to increase `xdbcore-loadableunit-size`, to obtain better performance. If `xdbcore-loadableunit-size` equals `xdbcore-xobmem-bound`, then try to increase both parameters for further performance improvements.

10

XML Schema Evolution

This chapter describes how you can update your XML schema after you have registered it with Oracle XML DB. XML schema evolution is the process of updating your registered XML schema.

This chapter contains these topics:

- Overview of XML Schema Evolution
- Using Copy-Based Schema Evolution
- Using In-Place XML Schema Evolution

Oracle XML DB supports the W3C XML Schema recommendation. XML instance documents that conform to an XML schema can be stored and retrieved using SQL and protocols such as FTP, HTTP(S), and WebDAV. In addition to specifying the structure of XML documents, XML schemas determine the mapping between XML and object-relational storage.

See Also: Chapter 7, "XML Schema Storage and Query: Basic"

Overview of XML Schema Evolution

A major challenge for developers using an XML schema with Oracle XML DB is how to deal with changes in the content or structure of XML documents. In some environments, the need for changes may be frequent or extensive, arising from new regulations, internal needs, or external opportunities. For example, you might need to add new elements or attributes to an XML schema definition, modify a data type, or relax or tighten certain minimum and maximum occurrence requirements.

In such cases, you need to "evolve" the XML schema so that new requirements are accommodated, while any existing instance documents (the data) remain valid (or can be made valid), and existing applications can continue to run.

If you do not care about any existing documents, you can of course simply drop the `XMLType` tables that are dependent on the XML schema, delete the old XML schema, and register the new XML schema at the same URL. In most cases, however, you need to keep the existing documents, possibly transforming them to accommodate the new XML schema.

Oracle XML DB supports two kinds of schema evolution:

- **Copy-based schema evolution**, in which all instance documents that conform to the schema are copied to a temporary location in the database, the old schema is deleted, the modified schema is registered, and the instance documents are inserted into their new locations from the temporary area

- **In-place schema evolution**, which does not require copying, deleting, and inserting existing data and thus is much faster than copy-based evolution, but which has restrictions that do not apply to copy-based evolution

  In general, in-place evolution is permitted if you are not changing the storage model and if the changes do not invalidate existing documents (that is, if existing documents are conformant with the new schema or can be made conformant with it). A more detailed explanation of restrictions and guidelines is presented in "Using In-Place XML Schema Evolution" on page 10-15.

Each approach has its own PL/SQL procedure: `DBMS_XMLSCHEMA.copyEvolve` for copy-based evolution, `DBMS_XMLSCHEMA.inPlaceEvolve` for in-place evolution. This chapter explains the use of each procedure and presents guidelines for using its associated approach to schema evolution.

Using Copy-Based Schema Evolution

You perform copy-based XML schema evolution using procedure `copyEvolve` of PL/SQL package `DBMS_XMLSCHEMA`. Procedure `copyEvolve` copies existing instance documents to temporary `XMLType` tables to back them up, drops the old version of the XML schema (which also deletes the associated instance documents), registers the new version, and copies the backed-up instance documents to new `XMLType` tables. In case of a problem, the backup copies are restored—see "Rollback When Procedure DBMS_XMLSCHEMA.COPYEVOLVE Raises an Error" on page 10-9.

Using procedure `copyEvolve`, you can evolve your registered XML schema in such a way that existing XML instance documents continue to be valid.

Scenario for Copy-Based Evolution

Example 10–1 shows a *partial* listing of a revised version of the purchase-order XML schema of Example 3–10. See Example A–2 on page A-33 for the *complete* revised schema listing. Text that is in **bold** here is new or different from that in the original schema.

Example 10–1 Revised Purchase-Order XML Schema

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:xdb="http://xmlns.oracle.com/xdb"
           version="1.0">
  <xs:element
    name="PurchaseOrder" type="PurchaseOrderType"
    xdb:defaultTable="PURCHASEORDER"
    xdb:columnProps=
      "CONSTRAINT purchaseorder_pkey PRIMARY KEY (XMLDATA.reference),
       CONSTRAINT valid_email_address FOREIGN KEY (XMLDATA.userid)
         REFERENCES hr.employees (EMAIL)"
    xdb:tableProps=
      "VARRAY XMLDATA.ACTIONS.ACTION STORE AS TABLE ACTION_TABLE
        ((CONSTRAINT action_pkey PRIMARY KEY (NESTED_TABLE_ID, SYS_NC_ARRAY_INDEX$)))
       VARRAY XMLDATA.LINEITEMS.LINEITEM STORE AS TABLE LINEITEM_TABLE
        ((constraint LINEITEM_PKEY primary key (NESTED_TABLE_ID, SYS_NC_ARRAY_INDEX$)))
       lob (XMLDATA.NOTES) STORE AS (ENABLE STORAGE IN ROW STORAGE(INITIAL 4K NEXT 32K))"/>
  <xs:complexType name="PurchaseOrderType" xdb:SQLType="PURCHASEORDER_T">
    <xs:sequence>
      <xs:element name="Actions" type="ActionsType" xdb:SQLName="ACTIONS"/>
      <xs:element name="Reject" type="RejectionType" minOccurs="0" xdb:SQLName="REJECTION"/>
      <xs:element name="Requestor" type="RequestorType" xdb:SQLName="REQUESTOR"/>
      <xs:element name="User" type="UserType" xdb:SQLName="USERID"/>
      <xs:element name="CostCenter" type="CostCenterType" xdb:SQLName="COST_CENTER"/>
      <xs:element name="BillingAddress" type="AddressType" minOccurs="0"
```

```
                xdb:SQLName="BILLING_ADDRESS"/>
    <xs:element name="ShippingInstructions" type="ShippingInstructionsType"
                xdb:SQLName="SHIPPING_INSTRUCTIONS"/>
    <xs:element name="SpecialInstructions" type="SpecialInstructionsType"
                xdb:SQLName="SPECIAL_INSTRUCTIONS"/>
    <xs:element name="LineItems" type="LineItemsType" xdb:SQLName="LINEITEMS"/>
    <xs:element name="Notes" type="NotesType" minOccurs="0" xdb:SQLType="CLOB"
                xdb:SQLName="NOTES"/>
  </xs:sequence>
  <xs:attribute name="Reference" type="ReferenceType" use="required" xdb:SQLName="REFERENCE"/>
  <xs:attribute name="DateCreated" type="xs:dateTime" use="required"
                xdb:SQLType="TIMESTAMP WITH TIME ZONE"/>
</xs:complexType>
<xs:complexType name="LineItemsType" xdb:SQLType="LINEITEMS_T">
  <xs:sequence>
    <xs:element name="LineItem" type="LineItemType" maxOccurs="unbounded" xdb:SQLName="LINEITEM"
                xdb:SQLCollType="LINEITEM_V"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="LineItemType" xdb:SQLType="LINEITEM_T">
  <xs:sequence>
    <xs:element name="Part" type="PartType" xdb:SQLName="PART"/>
    <xs:element name="Quantity" type="quantityType"/>
  </xs:sequence>
  <xs:attribute name="ItemNumber" type="xs:integer" xdb:SQLName="ITEMNUMBER"
                xdb:SQLType="NUMBER"/>
</xs:complexType>
<xs:complexType name="PartType" xdb:SQLType="PART_T">
  <xs:simpleContent>
    <xs:extension base="UPCCodeType">
      <xs:attribute name="Description" type="DescriptionType" use="required"
                    xdb:SQLName="DESCRIPTION"/>
      <xs:attribute name="UnitCost" type="moneyType" use="required"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
<xs:simpleType name="ReferenceType">
  <xs:restriction base="xs:string">
    <xs:minLength value="18"/>
    <xs:maxLength value="30"/>
  </xs:restriction>
</xs:simpleType>

. . .

<xs:complexType name="RejectionType" xdb:SQLType="REJECTION_T">
  <xs:all>
    <xs:element name="User" type="UserType" minOccurs="0" xdb:SQLName="REJECTED_BY"/>
    <xs:element name="Date" type="DateType" minOccurs="0" xdb:SQLName="DATE_REJECTED"/>
    <xs:element name="Comments" type="CommentsType" minOccurs="0" xdb:SQLName="REASON_REJECTED"/>
  </xs:all>
</xs:complexType>
<xs:complexType name="ShippingInstructionsType" xdb:SQLType="SHIPPING_INSTRUCTIONS_T">
  <xs:sequence>
    <xs:element name="name" type="NameType" minOccurs="0" xdb:SQLName="SHIP_TO_NAME"/>
    <xs:choice>
      <xs:element name="address" type="AddressType" minOccurs="0"/>
      <xs:element name="fullAddress" type="FullAddressType" minOccurs="0"
                  xdb:SQLName="SHIP_TO_ADDRESS"/>
    </xs:choice>
    <xs:element name="telephone" type="TelephoneType" minOccurs="0" xdb:SQLName="SHIP_TO_PHONE"/>
  </xs:sequence>
</xs:complexType>

. . .
```

```
  <xs:simpleType name="NameType">
    <xs:restriction base="xs:string">
      <xs:minLength value="1"/>
      <xs:maxLength value="20"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="FullAddressType">
    <xs:restriction base="xs:string">
      <xs:minLength value="1"/>
      <xs:maxLength value="256"/>
    </xs:restriction>
  </xs:simpleType>

. . .

  <xs:simpleType name="DescriptionType">
    <xs:restriction base="xs:string">
      <xs:minLength value="1"/>
      <xs:maxLength value="256"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:complexType name="AddressType" xdb:SQLType="ADDRESS_T">
    <xs:sequence>
      <xs:element name="StreetLine1" type="StreetType"/>
      <xs:element name="StreetLine2" type="StreetType" minOccurs="0"/>
      <xs:element name="City" type="CityType"/>
      <xs:choice>
        <xs:sequence>
          <xs:element name="State" type="StateType"/>
          <xs:element name="ZipCode" type="ZipCodeType"/>
        </xs:sequence>
        <xs:sequence>
          <xs:element name="Province" type="ProvinceType"/>
          <xs:element name="PostCode" type="PostCodeType"/>
        </xs:sequence>
        <xs:sequence>
          <xs:element name="County" type="CountyType"/>
          <xs:element name="Postcode" type="PostCodeType"/>
        </xs:sequence>
      </xs:choice>
      <xs:element name="Country" type="CountryType"/>
    </xs:sequence>
  </xs:complexType>
  <xs:simpleType name="StreetType">
    <xs:restriction base="xs:string">
      <xs:minLength value="1"/>
      <xs:maxLength value="128"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="CityType">
    <xs:restriction base="xs:string">
      <xs:minLength value="1"/>
      <xs:maxLength value="64"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="StateType">
    <xs:restriction base="xs:string">
      <xs:minLength value="2"/>
      <xs:maxLength value="2"/>
      <xs:enumeration value="AK"/>
      <xs:enumeration value="AL"/>
      <xs:enumeration value="AR"/>

 . . . -- A value for each US state abbreviation

      <xs:enumeration value="WY"/>
```

```
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="ZipCodeType">
    <xs:restriction base="xs:string">
      <xs:pattern value="\d{5}"/>
      <xs:pattern value="\d{5}-\d{4}"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="CountryType">
    <xs:restriction base="xs:string">
      <xs:minLength value="1"/>
      <xs:maxLength value="64"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="CountyType">
    <xs:restriction base="xs:string">
      <xs:minLength value="1"/>
      <xs:maxLength value="32"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="PostCodeType">
    <xs:restriction base="xs:string">
      <xs:minLength value="1"/>
      <xs:maxLength value="12"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="ProvinceType">
    <xs:restriction base="xs:string">
      <xs:minLength value="2"/>
      <xs:maxLength value="2"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="NotesType">
    <xs:restriction base="xs:string">
      <xs:maxLength value="32767"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="UPCCodeType">
    <xs:restriction base="xs:string">
      <xs:minLength value="11"/>
      <xs:maxLength value="14"/>
      <xs:pattern value="\d{11}"/>
      <xs:pattern value="\d{12}"/>
      <xs:pattern value="\d{13}"/>
      <xs:pattern value="\d{14}"/>
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

copyEvolve Parameters and Errors

This is the signature of procedure `DBMS_XMLSCHEMA.copyEvolve`:

```
procedure copyEvolve(schemaURLs       IN XDB$STRING_LIST_T,
                     newSchemas       IN XMLSequenceType,
                     transforms       IN XMLSequenceType := NULL,
                     preserveOldDocs  IN BOOLEAN := FALSE,
                     mapTabName       IN VARCHAR2 := NULL,
                     generateTables   IN BOOLEAN := TRUE,
                     force            IN BOOLEAN := FALSE,
                     schemaOwners     IN XDB$STRING_LIST_T := NULL
                     parallelDegree   IN PLS_INTEGER := 0,
                     options          IN PLS_INTEGER := 0);
```

Table 10–1 describes the individual parameters. Table 10–2 describes the errors associated with the procedure.

Table 10–1 Parameters of Procedure DBMS_XMLSCHEMA.COPYEVOLVE

Parameter	Description
`schemaURLs`	Varray of URLs of XML schemas to be evolved (varray of `VARCHAR2(4000)`. This should include the dependent schemas as well. Unless the force parameter is `TRUE`, the URLs should be in the dependency order, that is, if URL A comes before URL B in the varray, then schema A should not be dependent on schema B but schema B may be dependent on schema A.
`newSchemas`	Varray of new XML schema documents (`XMLType` instances). Specify this in exactly the same order as the corresponding URLs. If no change is necessary in an XML schema, provide the unchanged schema.
`transforms`	Varray of XSL documents (`XMLType` instances) that are applied to XML schema based documents to make them conform to the new schemas. Specify these in exactly the same order as the corresponding URLs. If no transformations are required, this parameter need not be specified.
`preserveOldDocs`	If this is `TRUE`, then the temporary tables holding old data are not dropped at the end of schema evolution. See also "Guidelines for Using Procedure COPYEVOLVE".
`mapTabName`	Specifies the name of table that maps old `XMLType` table or column names to names of corresponding temporary tables.
`generateTables`	By default this parameter is `TRUE`. If `FALSE` then `XMLType` tables or columns are not generated after registering new XML schemas. If `FALSE`, `preserveOldDocs` must be `TRUE` and `mapTabName` must not be `NULL`.
`force`	If this is `TRUE`, then errors during the registration of new schemas are ignored. If there are circular dependencies among the schemas, set this flag to `TRUE` to ensure that each schema is stored even though there may be errors in registration.
`schemaOwners`	Varray of names of schema owners. Specify these in exactly the same order as the corresponding URLs.
`parallelDegree`	Specifies the degree of parallelism to be used in a `PARALLEL` hint during the data-copy stage. If this is `0` (default value), a `PARALLEL` hint is absent from the data-copy statements.
`options`	Miscellaneous options. The only option is `COPYEVOLVE_BINARY_XML`, which means to register the new XML schemas for binary XML data and create the new tables or columns with binary XML as the storage model.

Table 10–2 Errors Associated with Procedure DBMS_XMLSCHEMA.COPYEVOLVE

Error Number and Message	Cause	Action
30942 XML Schema Evolution error for schema '<schema_url>' table "<owner_name>.<table_name>" column '<column_name>'	The given `XMLType` table or column that conforms to the given XML schema had errors during evolution. In the case of a table, the column name is empty. See also the more specific error that follows this.	Based on the schema, table, and column information in this error and the more specific error that follows, take corrective action.

*Table 10–2 (Cont.) Errors Associated with Procedure DBMS_*

Error Number and Message	Cause	Action
30943 XML Schema '<schema_url>' is dependent on XML schema '<schema_url>'	Not all dependent XML schemas were specified or the schemas were not specified in dependency order, that is, if schema S1 is dependent on schema S, S must appear before S1.	Include the previously unspecified schema in the list of schemas or correct the order in which the schemas are specified. Then retry the operation.
30944 Error during rollback for XML schema '<schema_url>' table "<owner_name>.<table_name>" column '<column_name>'	The given `XMLType` table or column that conforms to the given XML schema had errors during a rollback of XML schema evolution. For a table, the column name is empty. See also the more specific error that follows this.	Based on the schema, table, and column information in this error and the more specific error that follows, take corrective action.
30945 Could not create mapping table '<table_name>'	A mapping table could not be created during XML schema evolution. See also the more specific error that follows this.	Ensure that a table with the given name does not exist and retry the operation.
30946 XML Schema Evolution warning: temporary tables not cleaned up	An error occurred after the schema was evolved while cleaning up temporary tables. The schema evolution was successful.	If you need to remove the temporary tables, use the mapping table to get the temporary table names and drop them.

Limitations of Procedure COPYEVOLVE

Keep in mind the following limitations when you use procedure `DBMS_XMLSCHEMA.copyEvolve`:

- Indexes, triggers, constraints, row-level security (RLS) policies, and other metadata related to the `XMLType` tables that are dependent on the schemas are not preserved. These must be re-created after evolution.
- If top-level element names are changed, additional steps are required after `copyEvolve` finishes executing. See "Top-Level Element Name Changes" on page 10-8.
- Copy-based evolution cannot be used if there is a table with an object-type column that has an `XMLType` attribute that is dependent on any of the schemas to be evolved. For example, consider this table:

  ```
  CREATE TYPE t1 AS OBJECT (n NUMBER, x XMLType);
  CREATE TABLE tab1 (e NUMBER, o t1) XMLType
   COLUMN o.x XMLSchema "s1.xsd" ELEMENT "Employee";
  ```

  This assumes that an XML schema with a top-level element `Employee` has been registered under URL `s1.xsd`. It is not possible to evolve this XML schema, because table `tab1` with column `o` with `XMLType` attribute `x` is dependent on the XML schema. Note that although `copyEvolve` does not handle `XMLType` object attributes, it does raise an error in such cases.

Guidelines for Using Procedure COPYEVOLVE

The following general guideline applies to using procedure `DBMS_XMLSCHEMA.copyEvolve`. The rest of this section describes specific guidelines that can also be appropriate in particular contexts.

1. Turn off the recycle bin, to prevent dropped tables from being copied to it:

   ```
   ALTER SESSION SET RECYCLEBIN=off;
   ```

2. Identify the XML schemas that are dependent on the XML schema that is to be evolved. You can acquire the URLs of the dependent XML schemas using the following query, where *`schema_to_be_evolved`* is the schema to be evolved, and *`owner_of_schema_to_be_evolved`* is its owner (database user).

   ```
   SELECT dxs.SCHEMA_URL, dxs.OWNER
       FROM DBA_DEPENDENCIES dd, DBA_XML_SCHEMAS dxs
       WHERE dd.REFERENCED_NAME = (SELECT INT_OBJNAME
                                     FROM DBA_XML_SCHEMAS
                                     WHERE SCHEMA_URL = schema_to_be_evolved
                                       AND OWNER = owner_of_schema_to_be_evolved)
         AND dxs.INT_OBJNAME = dd.NAME;
   ```

   In many cases, no changes are needed in the dependent XML schemas. But if the dependent XML schemas need to be changed, then you must also prepare new versions of those XML schemas.

3. If the existing instance documents do not conform to the new XML schema, then you must provide an XSL style sheet that, when applied to an instance document, transforms it to conform to the new schema. You must do this for each XML schema identified in Step 2. The transformation must handle documents that conform to all top-level elements in the new XML schema.

4. Call procedure `DBMS_XMLSCHEMA.copyEvolve`, specifying the XML schema URLs, new schemas, and transformation style sheet.

Top-Level Element Name Changes

Procedure `DBMS_XMLSCHEMA.copyEvolve` assumes that top-level elements have not been dropped and that their names have not been changed in the new XML schemas. If there are such changes in your new XML schemas, then you can call procedure `copyEvolve` with parameter `generateTables` set to `FALSE` and parameter `preserveOldDocs` set to `TRUE`. In this way, new tables are not generated, and the temporary tables holding the old documents (backup copies) are not dropped at the end of the procedure. You can then store the old documents in whatever form is appropriate and drop the temporary tables. See "copyEvolve Parameters and Errors" on page 10-5 for more details on using these parameters.

User-Created Virtual Columns of Tables Other Than Default Tables

For tables that are not default tables, any virtual columns that you create are not re-created during copy-based evolution. If the columns are needed, then set parameter `preserveOldDocs` to `TRUE`, create the tables, and copy the old documents after procedure `copyEvolve` has finished.

Ensure that the XML Schema and Dependents Are Not Used by Concurrent Sessions

Ensure that the XML schema and its dependents are not used by any concurrent session during the XML schema evolution process. If other, concurrent sessions have

shared locks on this schema at the beginning of the evolution process, then procedure DBMS_XMLSCHEMA.copyEvolve waits for these sessions to release the locks so that it can acquire an exclusive lock. However, this lock is released immediately to allow the rest of the process to continue.

Rollback When Procedure DBMS_XMLSCHEMA.COPYEVOLVE Raises an Error

Procedure DBMS_XMLSCHEMA.copyEvolve either completely succeeds or raises an error, in which case it attempts to roll back as much of the operation as possible. Evolving an XML schema involves many database DDL statements. When an error occurs, compensating DDL statements are executed to undo the effect of all steps executed to that point. If the old tables or schemas have been dropped, they are re-created, but any table, column, and storage properties and any auxiliary structures (such as indexes, triggers, constraints, and RLS policies) associated with the tables and columns are lost.

Failed Rollback From Insufficient Privileges

In certain cases you cannot roll back the copy-based evolution operation. For example, if table creation fails due to reasons not related to the new XML schema, then there is no way to roll back. An example is failure due to insufficient privileges. The temporary tables are not deleted even if preserveOldDocs is FALSE, so the data can be recovered. If the mapTabName parameter is null, the mapping table name is XDB$MAPTAB followed by a sequence number. The exact table name can be found using a query such as the following:

```
SELECT TABLE_NAME FROM USER_TABLES WHERE TABLE_NAME LIKE 'XDB$MAPTAB%';
```

Privileges Needed for XML Schema Evolution

Copy-based XML schema evolution may involve dropping or creating data types. Hence, you need type-related privileges such as DROP TYPE, CREATE TYPE, and ALTER TYPE.

You need privileges to delete and register the XML schemas involved in the evolution. You need all privileges on XMLType tables that conform to the schemas being evolved. For XMLType columns, the ALTER TABLE privilege is needed on corresponding tables. If there are schema-based XMLType tables or columns in other database schemas, you need privileges such as the following:

- CREATE ANY TABLE
- CREATE ANY INDEX
- SELECT ANY TABLE
- UPDATE ANY TABLE
- INSERT ANY TABLE
- DELETE ANY TABLE
- DROP ANY TABLE
- ALTER ANY TABLE
- DROP ANY INDEX

To avoid needing to grant all these privileges to the database- schema owner, Oracle recommends that a database administrator perform the evolution if there are XML schema-based XMLType table or columns belonging to other database schemas.

Updating Existing XML Instance Documents using a Style Sheet

After you modify a registered XML schema, you must update any existing XML instance documents that use the XML schema. You do this by applying an XSLT style sheet to each of the instance documents. The style sheet represents the difference between the old and new XML schemas.

Example 10–2 is a style sheet, in file `evolvePurchaseOrder.xsl`, that transforms existing purchase-order documents that use the old XML schema, so they use the new XML schema instead.

Example 10–2 evolvePurchaseOrder.xsl: Style Sheet to Update Instance Documents

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
  version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <xsl:output method="xml" encoding="UTF-8"/>
  <xsl:template match="/PurchaseOrder">
    <PurchaseOrder>
      <xsl:attribute name="xsi:noNamespaceSchemaLocation">
        http://localhost:8080/source/schemas/poSource/xsd/purchaseOrder.xsd
      </xsl:attribute>
      <xsl:for-each select="Reference">
        <xsl:attribute name="Reference">
          <xsl:value-of select="."/>
        </xsl:attribute>
      </xsl:for-each>
      <xsl:variable name="V264_394" select="'2004-01-01T12:00:00.000000-08:00'"/>
      <xsl:attribute name="DateCreated">
        <xsl:value-of select="$V264_394"/>
      </xsl:attribute>
      <xsl:for-each select="Actions">
        <Actions>
          <xsl:for-each select="Action">
            <Action>
              <xsl:for-each select="User">
                <User>
                  <xsl:value-of select="."/>
                </User>
              </xsl:for-each>
              <xsl:for-each select="Date">
                <Date>
                  <xsl:value-of select="."/>
                </Date>
              </xsl:for-each>
            </Action>
          </xsl:for-each>
        </Actions>
      </xsl:for-each>
      <xsl:for-each select="Reject">
        <Reject>
          <xsl:for-each select="User">
            <User>
              <xsl:value-of select="."/>
            </User>
          </xsl:for-each>
          <xsl:for-each select="Date">
            <Date>
              <xsl:value-of select="."/>
            </Date>
          </xsl:for-each>
          <xsl:for-each select="Comments">
            <Comments>
```

```
        <xsl:value-of select="."/>
      </Comments>
    </xsl:for-each>
  </Reject>
</xsl:for-each>
<xsl:for-each select="Requestor">
  <Requestor>
    <xsl:value-of select="."/>
  </Requestor>
</xsl:for-each>
<xsl:for-each select="User">
  <User>
    <xsl:value-of select="."/>
  </User>
</xsl:for-each>
<xsl:for-each select="CostCenter">
  <CostCenter>
    <xsl:value-of select="."/>
  </CostCenter>
</xsl:for-each>
<ShippingInstructions>
  <xsl:for-each select="ShippingInstructions">
    <xsl:for-each select="name">
      <name>
        <xsl:value-of select="."/>
      </name>
    </xsl:for-each>
  </xsl:for-each>
  <xsl:for-each select="ShippingInstructions">
    <xsl:for-each select="address">
      <fullAddress>
        <xsl:value-of select="."/>
      </fullAddress>
    </xsl:for-each>
  </xsl:for-each>
  <xsl:for-each select="ShippingInstructions">
    <xsl:for-each select="telephone">
      <telephone>
        <xsl:value-of select="."/>
      </telephone>
    </xsl:for-each>
  </xsl:for-each>
</ShippingInstructions>
<xsl:for-each select="SpecialInstructions">
  <SpecialInstructions>
    <xsl:value-of select="."/>
  </SpecialInstructions>
</xsl:for-each>
<xsl:for-each select="LineItems">
  <LineItems>
    <xsl:for-each select="LineItem">
      <xsl:variable name="V22" select="."/>
      <LineItem>
        <xsl:for-each select="@ItemNumber">
          <xsl:attribute name="ItemNumber">
            <xsl:value-of select="."/>
          </xsl:attribute>
        </xsl:for-each>
        <xsl:for-each select="$V22/Part">
          <xsl:variable name="V24" select="."/>
          <xsl:for-each select="@Id">
            <Part>
              <xsl:for-each select="$V22/Description">
                <xsl:attribute name="Description">
                  <xsl:value-of select="."/>
                </xsl:attribute>
```

```
                  </xsl:for-each>
                  <xsl:for-each select="$V24/@UnitPrice">
                    <xsl:attribute name="UnitCost">
                      <xsl:value-of select="."/>
                    </xsl:attribute>
                  </xsl:for-each>
                  <xsl:value-of select="."/>
                </Part>
              </xsl:for-each>
            </xsl:for-each>
            <xsl:for-each select="$V22/Part">
              <xsl:for-each select="@Quantity">
                <Quantity>
                  <xsl:value-of select="."/>
                </Quantity>
              </xsl:for-each>
            </xsl:for-each>
          </LineItem>
        </xsl:for-each>
      </LineItems>
    </xsl:for-each>
  </PurchaseOrder>
 </xsl:template>
</xsl:stylesheet>
```

Examples of Using Procedure COPYEVOLVE

Example 10–3 loads a revised XML schema and evolution XSL style sheet into Oracle XML DB Repository.

Example 10–3 Loading Revised XML Schema and XSL Style Sheet

```
DECLARE
  res BOOLEAN;
BEGIN
  res := DBMS_XDB.createResource(               -- Load revised XML schema
           '/source/schemas/poSource/revisedPurchaseOrder.xsd',
           bfilename('XMLDIR', 'revisedPurchaseOrder.xsd'),
           nls_charset_id('AL32UTF8'));
  res := DBMS_XDB.createResource(               -- Load revised XSL style sheet
           '/source/schemas/poSource/evolvePurchaseOrder.xsl',
           bfilename('XMLDIR', 'evolvePurchaseOrder.xsl'),
           nls_charset_id('AL32UTF8'));
END;
/
```

Example 10–4 shows how to use procedure `DBMS_XMLSCHEMA.copyEvolve` to evolve the XML schema `purchaseOrder.xsd` to `revisedPurchaseOrder.xsd` using the XSL style sheet `evolvePurchaseOrder.xsl`.

Example 10–4 Updating an XML Schema using DBMS_XMLSCHEMA.COPYEVOLVE

```
BEGIN
  DBMS_XMLSCHEMA.copyEvolve(
    xdb$string_list_t('http://localhost:8080/source/schemas/poSource/xsd/purchaseOrder.xsd'),
    XMLSequenceType(XDBURIType('/source/schemas/poSource/revisedPurchaseOrder.xsd').getXML()),
    XMLSequenceType(XDBURIType('/source/schemas/poSource/evolvePurchaseOrder.xsl').getXML()));
END;

SELECT XMLQuery('$p/PurchaseOrder/LineItems/LineItem[1]'
                PASSING po.OBJECT_VALUE AS "p" RETURNING CONTENT) line_item
  FROM purchaseorder po
```

```
  WHERE XMLExists('$p/PurchaseOrder[Reference="SBELL-2003030912333601PDT"]'
                  PASSING po.OBJECT_VALUE AS "p");

LINE_ITEM
--------------------------------------------------------------------------------
<LineItem ItemNumber="1">
  <Part Description="A Night to Remember" UnitCost="39.95">715515009058</Part>
  <Quantity>2</Quantity>
</LineItem>
```

The same query would have produced the following result before the schema evolution:

```
LINE_ITEM
-----------------------------------------------------------
<LineItem ItemNumber="1">
  <Description>A Night to Remember</Description>
  <Part Id="715515009058" UnitPrice="39.95" Quantity="2"/>
</LineItem>
```

Procedure `DBMS_XMLSCHEMA.copyEvolve` evolves registered XML schemas in such a way that existing instance documents continue to remain valid.

> **Caution:** Before executing procedure `DBMS_XMLSCHEMA.copyEvolve`, always *back up* all registered XML schemas and all XML documents that conform to them. Procedure `copyEvolve` *deletes* all documents that conform to registered XML schemas.

First, procedure `copyEvolve` copies the data in XML schema-based `XMLType` tables and columns to temporary tables. It then drops the original tables and columns, and deletes the old XML schemas. After registering the new XML schemas, it creates `XMLType` tables and columns and populates them with data (unless parameter `GENTABLES` is `FALSE`) but it does not create any auxiliary structures such as indexes, constraints, triggers, and row-level security (RLS) policies. Procedure `copyEvolve` creates the tables and columns as follows:

- It creates default tables while registering the new schemas.
- It creates tables that are not default tables using a statement of the following form:

  ```
  CREATE TABLE table_name OF XMLType OID 'oid'
     XMLSCHEMA schema_url ELEMENT element_name
  ```

  where *`OID`* is the original OID of the table, before it was dropped.
- It adds `XMLType` columns using a statement of the following form:

  ```
  ALTER TABLE table_name ADD (column_name XMLType) XMLType COLUMN
   column_name XMLSCHEMA schema_url ELEMENT element_name
  ```

When a new XML schema is registered, types are generated if the registration of the corresponding old schema had generated types. If an XML schema was global before the evolution, then it is also global after the evolution. Similarly, if an XML schema was local before the evolution, then it is also local (owned by the same user) after the evolution.

You have the option to preserve the temporary tables that contain the old documents, by setting parameter `preserveOldDocs` to `TRUE`. All temporary tables are created in

the database schema of the current user. For XMLType tables, the temporary table has the columns shown in Table 10–3.

Table 10–3 XML Schema Evolution: XMLType Table Temporary Table Columns

Name	Type	Comment
Data	CLOB	XML document from the old table, in CLOB format.
OID	RAW(16)	OID of the corresponding row in the old table.
ACLOID	RAW(16)	This column is present only if the old table is hierarchy enabled. ACLOID of corresponding row in old table.
OWNERID	RAW(16)	This column is present only if old table is hierarchy enabled. OWNERID of corresponding row in old table.

For XMLType columns, the temporary table has the columns shown in Table 10–4.

Table 10–4 XML Schema Evolution: XMLType Column Temporary Table Columns

Name	Type	Comment
Data	CLOB	XML document from the old column, in CLOB format.
RID	ROWID	ROWID of the corresponding row in the table containing this column.

Procedure copyEvolve stores information about the mapping from the old table or column name to the corresponding temporary table name in a separate table specified by parameter mapTabName. If preserveOldDocs is TRUE, then the mapTabName parameter must not be NULL, and it must not be the name of any existing table in the current database schema. Each row in the mapping table has information about one of the old tables/columns. Table 10–5 shows the mapping table columns.

Table 10–5 Procedure copyEvolve Mapping Table

Column Name	Column Type	Comment
SCHEMA_URL	VARCHAR2(700)	URL of the schema to which this table or column conforms.
SCHEMA_OWNER	VARCHAR2(30)	Owner of the schema.
ELEMENT_NAME	VARCHAR2(256)	Element to which this table or column conforms.
TABLE_NAME	VARCHAR2(65)	Qualified name of the table (<owner_name>.<table_name>).
TABLE_OID	RAW(16)	OID of table.
COLUMN_NAME	VARCHAR2(4000)	Name of the column (NULL for XMLType tables).
TEMP_TABNAME	VARCHAR2(30)	Name of temporary table that holds the data for this table or column.

You can avoid generating any tables or columns after registering the new XML schema by setting parameter GENTABLES to FALSE. If GENTABLES is FALSE, parameter PRESERVEOLDDOCS must be TRUE and parameter MAPTABNAME must not be NULL.

This ensures that the data in the old tables is not lost. This is useful if you do not want the tables to be created by the procedure, as described in section "copyEvolve Parameters and Errors" on page 10-5.

By default, it is assumed that all XML schemas are owned by the current user. If this is not true, then you must specify the owner of each XML schema in the `schemaOwners` parameter.

> **See Also:** *Oracle Database SQL Language Reference* for the complete description of `ALTER TABLE`

Using In-Place XML Schema Evolution

In-place XML schema evolution makes changes to an XML schema without requiring that existing data be copied, deleted, and reinserted. In-place evolution is thus much faster than copy-based evolution. However, in-place evolution also has several restrictions that do not apply to copy-based evolution.

You use procedure `DBMS_XMLSCHEMA.inPlaceEvolve` to perform in-place evolution. Using this procedure, you identify the changes to be made to an existing XML schema by specifying an XML schema-differences document, and you optionally specify flags to be applied to the evolution process.

In-place evolution constructs a new version of an XML schema by applying changes specified in a `diffXML` document, validates that new XML schema (against the XML schema for XML schemas), constructs DDL statements to evolve the disk structures used to store the XML instance documents associated with the XML schema, executes these DDL statements, and replaces the old version of the XML schema with the new, in that order. If the new version of the XML schema is not a valid schema, then in-place evolution fails.

Restrictions for In-Place XML Schema Evolution

Because in-place XML schema evolution avoids copying data, it does not permit arbitrary changes to an XML schema. This section describes why certain changes are not permitted. For the list of changes supported by in-place evolution, see "Supported Operations for In-Place XML Schema Evolution" on page 10-17.

The primary restriction on using in-place evolution can be stated generally as a requirement that a given XML schema can be evolved in place in only a backward-compatible way. **Backward-compatible** here means that any possible instance document that would validate against a given XML schema must also validate against a later (evolved) version of that XML schema.

This applies to *all possible* conforming instance documents, not only to *existing* instance documents For XML data that is stored as binary XML, backward compatibility also means that any XML schema annotations that affect binary XML treatment must not change during evolution. Backward compatibility is described in section "Backward-Compatibility Restrictions" on page 10-15.

In addition to this general backward-compatibility restriction, there are some other restrictions for in-place evolution. These are described in section "Other Restrictions on In-Place Evolution" on page 10-17.

Backward-Compatibility Restrictions

The restrictions described in this section ensure backward compatibility of an evolved XML schema, so that any possible instance documents that satisfy the old XML schema also satisfy the new schema.

Changes in Data Layout on Disk Certain changes to an XML schema alter the layout of the associated instance documents on disk, and are therefore not permitted. This situation is more common when the storage layer is tightly integrated with information derived from the XML schema, as is the case for object-relational storage.

One such example is an XML schema, registered for object-relational storage mapping, that is evolved by splitting a complex type into two complex types. In Example 10–5, complex type `ShippingInstructionsType` is split into two complex types, `Person-Name` and `Contact-Info`, and the `ShippingInstructionsType` complex type is deleted.

Example 10–5 Splitting a Complex Type into Two Complex Types

These code excerpts show the definitions of the original `ShippingInstructionsType` type and the new `Person-Name` and `Contact-Info` types.

```
<complexType name="ShippingInstructionsType">
    <sequence>
        <element name="name"   type="NameType" minOccurs="0"/>
        <element name="address" type="AddressType" minOccurs="0"/>
        <element name="telephone" type="TelephoneType" minOccurs="0"/>
    </sequence>
</complexType>

<complexType name="Person-Name">
    <sequence>
        <element name="name" type="NameType" minOccurs="0"/>
    </sequence>
</complexType>

<complexType name="Contact-Info">
    <sequence>
        <element name="address" type="AddressType" minOccurs="0"/>
        <element name="telephone" type="TelephoneType" minOccurs="0"/>
    </sequence>
</complexType>
```

Even if this XML schema has no associated instance documents, and therefore no data copy is required, a change in the layout of existing tables is required to accommodate future instance documents.

Reordering of XML Schema Constructs You cannot use in-place evolution to reorder schema elements in a way that affects the DOM fidelity of instance documents. For example, you cannot change the order of elements within a `<sequence>` element in a complex type definition. As an example, if a complex type named `ShippingInstructionsType` requires that its child elements `name`, `address`, and `telephone` be in that order, you cannot use in-place evolution to change the order to `name`, `telephone`, and `address`.

Changes from a Collection to a Non-Collection You cannot use in-place evolution to change a collection to a non-collection. An example would be changing from a `maxOccurs` value greater than one to a `maxOccurs` value of one. You cannot use in-place evolution to delete an element from a complex type if the deletion requires that a collection be evolved to a non-collection.

Other Restrictions on In-Place Evolution

The restrictions on in-place XML schema evolution that are described in this section are necessary for reasons other than backward compatibility of the evolved XML schema.

Changes to Attributes in Namespace xdb Except for attribute `xdb:defaultTable`, you cannot use in-place evolution to modify any attributes in namespace `http://xmlns.oracle.com/xdb` (which has the predefined prefix `xdb`).

Changes from a Non-Collection to a Collection When XML data is stored object-relationally, you cannot use in-place evolution to change a non-collection object type to a collection object type. An example would be adding an element to a complex type with the element name matching the name of an element already present in the type (or in another type that is related to the first type through inheritance).

Supported Operations for In-Place XML Schema Evolution

This section describes operations that are supported for in-place schema evolution. This list of supported operations is not necessarily exhaustive. Some of the operations listed here are not permitted in specific contexts, which are specified. In particular, some of the operations described here are not permitted for XML schemas that are used with binary XML.

- **Add an optional element to a complex type or group:** Always permitted. An example is the addition of the optional element `shipmethod` in the following complex type definition:

```
<xs:complexType name="ShippingInstructionsType">
    <xs:sequence>
        <xs:element name="name" type="NameType" minOccurs="0"/>
        <xs:element name="address" type="AddressType" minOccurs="0"/>
        <xs:element name="telephone" type="TelephoneType" minOccurs="0"/>
        <xs:element name = "shipmethod" type = "xs:string" minOccurs = "0"/>
    </xs:sequence>
</xs:complexType>
```

- **Add an optional attribute to a complex type or attribute group:** Always permitted. An example is the addition of the optional attribute `shipbydate` in the following complex type definition:

```
<xs:complexType name="ShippingInstructionsType">
    <xs:sequence>
        <xs:element name="name" type="NameType" minOccurs="0"/>
        <xs:element name="address" type="AddressType" minOccurs="0"/>
        <xs:element name="telephone" type="TelephoneType" minOccurs="0"/>
    </xs:sequence>
    <xs:attribute name="shipbydate" type="DateType" use="optional"/>
</xs:complexType>
```

- **Convert an element from simple type to complex type with simple content:** Supported only if the storage model is binary XML.
- **Modify the value attribute of an existing maxLength element:** Always permitted. The value can only be increased, not decreased.
- **Add an enumeration value:** You can add a new enumeration value only to the end of an enumeration list.

- **Add a global element:** Always permitted. An example is the addition of the global element `PurchaseOrderComment` in the following schema definition:

```
<xs:schema ...>
...
   <xs:element name="PurchaseOrderComment" type="string" xdb:defaultTable=""/>
..
</xs:schema>
```

- **Add a global attribute:** Always permitted.
- **Add or delete a global complex type:** Always permitted. An example is the addition of the global complex type `ComplexAddressType` in the following schema definition:

```
<xs:schema ...>
....
    <xs:complexType name="ComplexAddressType">
        <xs:sequence>
            <xs:element name="street" type="string"/>
            <xs:element name="city" type="string"/>
            <xs:element ref="zip" type="positiveInteger"/>
            <xs:element name="country"  type="string"/>
        </xs:sequence>
     </xs:complexType>
...
</xs:schema>
```

- **Add or delete a global simple type:** Always permitted.
- **Change the minOccurs attribute value:** The value of `minOccurs` can only be decreased.
- **Change the maxOccurs attribute value:** The value of `maxOccurs` can only be increased, and this is only possible for data stored as binary XML. That is, you cannot make any change to the `maxOccurs` attribute for data stored object-relationally.
- **Add or delete a global group or attributeGroup:** Always permitted. An example is the addition of an `Instructions` group in the following type definition:

```
<xsd:schema ...>
...
  <xsd:group name="Instructions">
    <xsd:sequence>
     <xsd:element name="ShippingInstructions" type="ShippingInstructionsType"/>
      <xsd:element name="SpecialInstructions" type=" SpecialInstructionsType"/>
    </xsd:sequence>
  </xsd:group>
...
</xsd:schema>
```

- **Change the xdb:defaultTable attribute value:** Always permitted. Changes are *not* permitted to any other attributes in the `xdb` namespace.
- **Add, modify, or delete a comment or processing instruction:** Always permitted.

Guidelines for Using In-Place XML Schema Evolution

The following guidelines apply to in-place XML-schema evolution:

- *Before* you perform an in-place XML-schema evolution:

- *Back up all existing data* (instance documents) for the XML schema to be evolved.

  Caution: Make sure that you back up your data before performing in-place XML schema evolution, in case the result is not what you intended. There is *no rollback* possible after an in-place evolution. If any errors occur during evolution, or if you make a major mistake and need to redo the entire operation, you must be able to go back to the backup copy of your original data.

- *Perform a dry run* using trace only, that is, without actually evolving the XML schema or updating any instance documents, produce a trace of the update operations that would be performed during evolution. To do this, set the `flag` parameter value to only `INPLACE_TRACE`. Do not also use `INPLACE_EVOLVE`.

  After performing the dry run, examine the trace file, verifying that the listed DDL operations are in fact those that you intend.

- *After* you perform an in-place XML-schema evolution:

  If you are accessing the database using a client that caches data, or if you are not sure whether this is the case, then *restart your client*. Otherwise, the pre-evolution version of the XML schema might continue to be used locally, with unpredictable results.

  See Also: *Oracle Database Administrator's Guide* for information about using trace files

inPlaceEvolve Parameters

This is the signature of procedure `DBMS_XMLSCHEMA.inPlaceEvolve`:

```
procedure inPlaceEvolve(schemaURL IN VARCHAR2,
                        diffXML   IN XMLType,
                        flags     IN NUMBER);
```

Table 10–6 describes the individual parameters.

Table 10–6 Parameters of Procedure DBMS_XMLSCHEMA.INPLACEEVOLVE

Parameter	Description
`schemaURL`	URL of the XML schema to be evolved (`VARCHAR2`).
`diffXML`	XML document (`XMLType` instance) that conforms to the `xdiff` XML schema, and that specifies the changes to apply and the locations in the XML schema where the changes are to be applied. For information about how to create the document for this parameter, see "Creating the Document for the diffXML Parameter" on page 10-20.

Table 10–6 (Cont.) Parameters of Procedure DBMS_XMLSCHEMA.INPLACEEVOLVE

Parameter	Description
`flags`	A bit mask that controls the behavior of the procedure. You can set the following bit values in this mask independently, summing them to define the overall effect. The default `flags` value is 1 (bit 1 on, bit 2 off), meaning that in-place evolution is performed and no trace is written. ■ `INPLACE_EVOLVE` (value 1, meaning that bit 1 is on) – Perform in-place XML schema evolution. Construct a new XML schema and validate it (against the XML schema for XML schemas). Construct the DDL statements needed to evolve the instance-document disk structures. Execute the DDL statements. Replace the old XML schema with the new. ■ `INPLACE_TRACE` (value 2, meaning that bit 2 is on) – Perform all steps necessary for in-place evolution, *except* executing the DDL statements and overwriting the old XML schema with the new, then write both the DDL statements and the new XML schema to a trace file. That is, each of the bits constructs the new XML schema, validates it, and determines the steps needed to evolve the disk structures underlying the instance documents. In addition: ■ Bit `INPLACE_EVOLVE` carries out those evolution steps and replaces the old XML schema with the new. ■ Bit `INPLACE_TRACE` saves the evolution steps and the new XML schema in a trace file (it does not carry out the evolution steps).

Procedure `DBMS_XMLSCHEMA.inPlaceEvolve` raises an error in the following cases:

- An XPath expression is invalid, or is syntactically correct but does not target a node in the XML schema.
- The `diffXML` document does not conform to the `xdiff` XML schema.
- The change makes the XML schema invalid or not well formed.
- A generated DDL statement (`CREATE TYPE`, `ALTER TYPE`, and so on) causes a problem when it is executed.
- An index object associated with an `XMLType` table is in an unsafe state, which could be caused by partition management operations.

Creating the Document for the diffXML Parameter

The value of the `diffXML` parameter to procedure `DBMS_XMLSCHEMA.inPlaceEvolve` is an XML document (as an XMLType instance) that specifies the changes to be applied to an XML schema for in-place evolution. This `diffXML` document contains a sequence of operations that describe the changes between the old XML schema and the new (the intended evolution result). The changes specified by the `diffXML` document are applied in order.

You must create the XML document to be used for the `diffXML` parameter You can do this in any of the following ways:

- The `XMLDiff` JavaBean (`oracle.xml.differ.XMLDiff`)
- The `xmldiff` command-line utility
- SQL function `XMLDiff`

The `diffXML` parameter document must conform to the `xdiff` XML schema.

The rest of this section presents examples of some operations in a document that conforms to the `xdiff` XML schema.

See Also:

- "xdiff.xsd: XML Schema for Comparing Schemas for In-Place Evolution" on page A-28
- *Oracle XML Developer's Kit Programmer's Guide* for information on using the `XMLDiff` JavaBean
- *Oracle XML Developer's Kit Programmer's Guide* for information on command-line utility `xmldiff`
- *Oracle Database SQL Language Reference* for information on SQL function `XMLDiff`

diffXML Operations and Examples

This section describes some operations that can be specified in the document for the `diffXML` document supplied to procedure `DBMS_XMLSCHEMA.inPlaceEvolve`. It presents an example XML document that conforms to the `xdiff` XML schema.

The `<append-node>` element is used for most of the supported changes, such as adding a new attribute to a complex type or appending a new element to a group.

The `<insert-node-before>` element specifies that a node of the given type should be inserted before the specified node. The `xpath` attribute specifies the location of the specified node and the `node-type` attribute specifies the type of node to be inserted. The node to be inserted is specified by the `<content>` child element. The `<insert-node-before>` element is mainly used for inserting comments and processing instructions, and for changing and adding add annotation elements.

The `<delete-node>` element specifies that the node with the given XPath (specified by the `xpath` attribute) should be deleted along with all its children. For example, you can use this element to delete comments and annotation elements. You can also use this element, in conjunction with `<append-node>` or `<insert-node-before>`, to make changes to an existing node.

Example 10–6 shows an XML document for the `diffXML` parameter that specifies the following changes:

- Delete complex type `PartType`.
- Add complex type `PartType` with a maximum length of 28.
- Add a comment before element `ShippingInstructions`.
- Add a required element `shipmethod` to element `ShippingInstructions`.

Example 10–6 diffXML Parameter Document

```
<xd:xdiff  xmlns="http://www.w3c.org/2001/XMLSchema"
           xmlns:xd="http://xmlns.oracle.com/xdb/xdiff.xsd"
           xmlns:xsi="http://www.w3c.org/2001/XMLSchema-Instance"
           xsi:schemaLocation="http://xmlns.oracle.com/xdb/xdiff.xsd
           http://xmlns.oracle.com/xdb/xdiff.xsd">
 <xd:delete-node xpath="/schema/complexType[@name=&quote;PartType&quote;]//maxLength/>
 <xd:append-node
  parent-xpath = "/schema/complexType[@name=&quote;PartType&quote;]//restriction"
  node-type = "element">
  <xd:content>
    <xs:maxLength value = "28"/>
  </xd:content>
 </xd:append-node>
 <xd:insert-node-before
```

```
  xpath="/schema/complexType[@name =&quote;ShippingInstructionsType&quote;]/sequence"
  node-type="comment">
  <xd:content>
    <!-- A type representing instructions for shipping -->
  </xd:content>
 </xd:insert-node-before>
 <xd:append-node
  parent-xpath="/schema/complexType[@name=&quote;ShippingInstructionsType&quote;]/sequence"
  node-type="element">
  <xd:content>
   <xs:element name = "shipmethod" type = "xs:string" minOccurs = "1"/>
  </xd:content>
 </xd:append-node>
</xd:xdiff>
```

11

Transforming and Validating XMLType Data

This chapter describes the SQL functions and `XMLType` APIs for transforming `XMLType` data using XSLT style sheets. It also explains the various functions and APIs available for validating the `XMLType` instance against an XML schema.

This chapter contains these topics:

- Transforming XMLType Instances
- XMLTRANSFORM and XMLType.transform(): Examples
- Validating XMLType Instances
- Validating XML Data Stored as XMLType: Examples

Transforming XMLType Instances

XML documents have structure but no format. To add format to the XML documents you can use Extensible Stylesheet Language (XSL). XSL provides a way of displaying XML semantics. It can map XML elements into other formatting or mark-up languages such as HTML.

In Oracle XML DB, `XMLType` instances or XML data stored in `XMLType` tables, columns, or views in Oracle Database, can be (formatted) transformed into HTML, XML, and other mark-up languages, using XSL style sheets and `XMLType` method `transform()`. This process conforms to the W3C XSL Transformations 1.0 Recommendation.

`XMLType` instance can be transformed in the following ways:

- Using Oracle SQL function `XMLtransform` or `XMLType` method `transform()` in the database.
- Using XDK transformation options in the middle tier, such as XSLT Processor for Java.

Note: PL/SQL package `DBMS_XSLPROCESSOR` provides a convenient and efficient way of applying a single style sheet to multiple documents. The performance of this package is better than that of method `transform()`, because the style sheet is parsed only once.

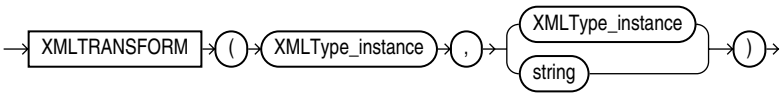
See Also:

- Chapter 3, "Using Oracle XML DB", the section, "XSL Transformation and Oracle XML DB" on page 3-64
- "PL/SQL XSLT Processor for XMLType (DBMS_XSLPROCESSOR)" on page 13-20
- *Oracle XML Developer's Kit Programmer's Guide*, the chapter on XSQL Pages Publishing Framework

SQL Function XMLTRANSFORM and XMLType Method TRANSFORM()

Figure 11–1 shows the syntax of Oracle SQL function `XMLtransform`. This function takes as arguments an `XMLType` instance and an XSLT style sheet. The style sheet can be an `XMLType` instance or a `VARCHAR2` string literal. It applies the style sheet to the instance and returns an `XMLType` instance.

Figure 11–1 XMLTRANSFORM Syntax



You can alternatively use `XMLType` method `transform()` as an alternative to Oracle SQL function `XMLtransform`. It has the same functionality.

Figure 11–2 shows how `XMLtransform` transforms an XML document by using an XSLT style sheet. It returns the processed output as XML, HTML, and so on, as specified by the XSLT style sheet. You typically use `XMLtransform` when retrieving or generating XML documents stored as `XMLType` in the database.

See Also: Figure 1–1, "XMLType Storage and Oracle XML DB Repository" in Chapter 1, "Introduction to Oracle XML DB"

Figure 11–2 Using XMLTRANSFORM

XMLTRANSFORM and XMLType.transform(): Examples

The examples in this section illustrate how to use Oracle SQL function `XMLtransform` and `XMLType` method `transform()` to transform XML data stored as `XMLType` to various formats.

Example 11–1 sets up an XML schema and tables that are needed to run other examples in this chapter. The call to `deleteSchema` here ensures that there is no existing XML schema before creating one. If no such schema exists, then `deleteSchema` produces an error.

Example 11–1 Registering an XML Schema and Inserting XML Data

```
BEGIN
  -- Delete the schema, if it already exists.
  DBMS_XMLSCHEMA.deleteSchema('http://www.example.com/schemas/ipo.xsd',4);
END;
```

```
/
BEGIN
  -- Register the schema
  DBMS_XMLSCHEMA.registerSchema(
    SCHEMAURL => 'http://www.example.com/schemas/ipo.xsd',
    SCHEMADOC => '<schema targetNamespace="http://www.example.com/IPO"
                          xmlns="http://www.w3.org/2001/XMLSchema"
                          xmlns:ipo="http://www.example.com/IPO">
                    <!-- annotation>
                      <documentation xml:lang="en">
                       International Purchase order schema for Example.com
                       Copyright 2000 Example.com. All rights reserved.
                      </documentation>
                    </annotation -->
                    <element name="purchaseOrder" type="ipo:PurchaseOrderType"/>
                    <element name="comment" type="string"/>
                    <complexType name="PurchaseOrderType">
                      <sequence>
                        <element name="shipTo"     type="ipo:Address"/>
                        <element name="billTo"     type="ipo:Address"/>
                        <element ref="ipo:comment" minOccurs="0"/>
                        <element name="items"      type="ipo:Items"/>
                      </sequence>
                      <attribute name="orderDate" type="date"/>
                    </complexType>
                    <complexType name="Items">
                      <sequence>
                        <element name="item" minOccurs="0" maxOccurs="unbounded">
                          <complexType>
                            <sequence>
                              <element name="productName" type="string"/>
                              <element name="quantity">
                                <simpleType>
                                  <restriction base="positiveInteger">
                                    <maxExclusive value="100"/>
                                  </restriction>
                                </simpleType>
                              </element>
                              <element name="USPrice"    type="decimal"/>
                              <element ref="ipo:comment" minOccurs="0"/>
                              <element name="shipDate"   type="date" minOccurs="0"/>
                            </sequence>
                            <attribute name="partNum" type="ipo:SKU" use="required"/>
                          </complexType>
                        </element>
                      </sequence>
                    </complexType>
                    <complexType name="Address">
                      <sequence>
                        <element name="name"    type="string"/>
                        <element name="street"  type="string"/>
                        <element name="city"    type="string"/>
                        <element name="state"   type="string"/>
                        <element name="country" type="string"/>
                        <element name="zip"     type="string"/>
                      </sequence>
                    </complexType>
                    <simpleType name="SKU">
                      <restriction base="string">
                        <pattern value="[0-9]{3}-[A-Z]{2}"/>
```

```
                    </restriction>
                  </simpleType>
                </schema>',
    LOCAL     => TRUE,
    GENTYPES  => TRUE);
END;
/

-- Create table to hold XML instance documents
DROP TABLE po_tab;

CREATE TABLE po_tab (id NUMBER, xmlcol XMLType)
 XMLType COLUMN xmlcol
 XMLSCHEMA "http://www.example.com/schemas/ipo.xsd"
 ELEMENT "purchaseOrder";

INSERT INTO po_tab
  VALUES(1, XMLType(
              '<?xml version="1.0"?>
               <ipo:purchaseOrder
                 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                 xmlns:ipo="http://www.example.com/IPO"
                 xsi:schemaLocation="http://www.example.com/IPO
                                     http://www.example.com/schemas/ipo.xsd"
                 orderDate="1999-12-01">
                 <shipTo>
                   <name>Helen Zoe</name>
                   <street>121 Broadway</street>
                   <city>Cardiff</city>
                   <state>Wales</state>
                   <country>UK</country>
                   <zip>CF2 1QJ</zip>
                 </shipTo>
                 <billTo>
                   <name>Robert Smith</name>
                   <street>8 Oak Avenue</street>
                   <city>Old Town</city>
                   <state>CA</state>
                   <country>US</country>
                   <zip>95819</zip>
                 </billTo>
                 <items>
                   <item partNum="833-AA">
                     <productName>Lapis necklace</productName>
                     <quantity>1</quantity>
                     <USPrice>99.95</USPrice>
                     <ipo:comment>Want this for the holidays!</ipo:comment>
                     <shipDate>1999-12-05</shipDate>
                   </item>
                 </items>
               </ipo:purchaseOrder>'));
```

Example 11–2 shows how to retrieve a style sheet using SQL function `XMLtransform` and `DBURIType`. See Chapter 20, "Accessing Data Through URIs" for information about `DBURIType`.

Example 11–2 Retrieving a Style Sheet using XMLTRANSFORM and DBURITYPE

```
DROP TABLE stylesheet_tab;
```

```
CREATE TABLE stylesheet_tab (id NUMBER, stylesheet XMLType);

INSERT INTO stylesheet_tab
  VALUES (1,
          XMLType(
            '<?xml version="1.0" ?>
             <xsl:stylesheet version="1.0"
                             xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
               <xsl:template match="*">
                 <td>
                   <xsl:choose>
                     <xsl:when test="count(child::*) > 1">
                       <xsl:call-template name="nested"/>
                     </xsl:when>
                     <xsl:otherwise>
                       <xsl:value-of select="name(.)"/>:<xsl:value-of
                                                          select="text()"/>
                     </xsl:otherwise>
                   </xsl:choose>
                 </td>
               </xsl:template>
               <xsl:template match="*" name="nested" priority="-1" mode="nested2">
                 <b>
                   <!-- xsl:value-of select="count(child::*)"/ -->
                   <xsl:choose>
                     <xsl:when test="count(child::*) > 1">
                       <xsl:value-of select="name(.)"/>:<xsl:apply-templates
                                                          mode="nested2"/>
                     </xsl:when>
                     <xsl:otherwise>
                       <xsl:value-of select="name(.)"/>:<xsl:value-of
                                                          select="text()"/>
                     </xsl:otherwise>
                   </xsl:choose>
                 </b>
               </xsl:template>
             </xsl:stylesheet>'));

SELECT XMLSerialize(DOCUMENT XMLtransform(x.xmlcol, y.stylesheet)
                    AS varchar2(1000))
  AS result FROM po_tab x, stylesheet_tab y WHERE y.id = 1;
```

This produces the following output (pretty-printed here for readability):

```
RESULT
-------------------------------------------------------
<td>
  <b>ipo:purchaseOrder:
    <b>shipTo:
      <b>name:Helen Zoe</b>
      <b>street:100 Broadway</b>
      <b>city:Cardiff</b>
      <b>state:Wales</b>
      <b>country:UK</b>
      <b>zip:CF2 1QJ</b>
    </b>
    <b>billTo:
      <b>name:Robert Smith</b>
      <b>street:8 Oak Avenue</b>
      <b>city:Old Town</b>
      <b>state:CA</b>
```

```
      <b>country:US</b>
      <b>zip:95819</b>
    </b>
    <b>items:</b>
  </b>
</td>
```

Example 11–3 illustrates the use of a stored style sheet to transform `XMLType` instances. Unlike Example 11–2, Example 11–3 uses a scalar subquery to retrieve the stored style sheet.

Example 11–3 Retrieving a Style Sheet using XMLTRANSFORM and a Subquery

```
SELECT XMLSerialize(DOCUMENT
                    XMLtransform(
                      x.xmlcol,
                      (SELECT stylesheet FROM stylesheet_tab WHERE id = 1))
                    AS VARCHAR2(1000))
  AS result FROM po_tab x;
```

Example 11–4 uses `XMLType` method `transform()` to transform an `XMLType` instance using a transient style sheet.

Example 11–4 Using Method TRANSFORM() with a Transient Style Sheet

```
SELECT XMLSerialize(
         DOCUMENT
         x.xmlcol.transform(
           XMLType('<?xml version="1.0" ?>
                  <xsl:stylesheet
                      version="1.0"
                      xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
                    <xsl:template match="*">
                      <td>
                        <xsl:choose>
                          <xsl:when test="count(child::*) > 1">
                            <xsl:call-template name="nested"/>
                          </xsl:when>
                          <xsl:otherwise>
                            <xsl:value-of
                               select =
                                 "name(.)"/>:<xsl:value-of select="text()"/>
                          </xsl:otherwise>
                        </xsl:choose>
                      </td>
                    </xsl:template>
                    <xsl:template match="*" name="nested" priority="-1"
                                  mode="nested2">
                      <b>
                        <!-- xsl:value-of select="count(child::*)"/ -->
                        <xsl:choose>
                          <xsl:when test="count(child::*) > 1">
                            <xsl:value-of select="name(.)"/>:
                            <xsl:apply-templates mode="nested2"/>
                          </xsl:when>
                          <xsl:otherwise>
                            <xsl:value-of
                               select =
                                 "name(.)"/>:<xsl:value-of select="text()"/>
                          </xsl:otherwise>
```

```
                    </xsl:choose>
                  </b>
                </xsl:template>
              </xsl:stylesheet>'))
        AS varchar2(1000))
FROM po_tab x;
```

Validating XMLType Instances

Often, besides knowing whether a particular XML document is well-formed, you need to know whether it conforms to a given XML schema, that is, whether it is valid with respect to that XML schema.

XML schema-based data that is stored as binary XML it is automatically validated fully whenever it is inserted or updated. This validation does not require building a DOM. It is done using streaming, which is efficient and minimizes memory use.

For `XMLType` data that is stored object-relationally, full validation requires building a DOM, which can be costly in terms of memory management. For this reason, Oracle XML DB does not automatically perform full validation when you insert or update data that is stored object-relationally.

However, in the process of decomposing XML data to store it object-relationally, Oracle XML DB does automatically perform partial validation, to ensure that the structure of the XML document conforms to the SQL data type definitions that were derived from the XML schema.

If you require full validation for `XMLType` data stored object-relationally, then consider validating on the client before inserting the data into the database or updating it.

You can use the following to perform full validation and manipulate the recorded validation status of XML documents:

- Oracle SQL function **`XMLIsValid`** and `XMLType` method **`IsSchemaValid()`** – Run the validation process unconditionally. Do not record any validation status. Return:
  - `1` if the document is determined to be *valid*.
  - `0` if the document is determined to be *invalid* or the validity of the document *cannot be determined*.
- `XMLType` method **`SchemaValidate()`** – Runs the validation process if the validation status is `0`, which it is by default. Sets the validation status to `1` if the document is determined to be *valid*. (Otherwise, the status remains `0`.)
- `XMLType` method **`isSchemaValidated()`** returns the recorded validation status of an `XMLType` instance.
- `XMLType` method **`setSchemaValidated()`** sets (records) the validation status of an `XMLType` instance.

Note that the validation status indicates knowledge of validity, as follows:

- `1` means that the document is known to be *valid*.
- `0` means that validity of the document is *unknown*. The document might have been shown to be invalid during a validation check, but that invalidity is not recorded. A recorded validation status of `0` indicates only a lack of knowledge about the document's validity.

See Also:

- "Comparing Partial to Full XML Schema Validation" on page 3-32
- *Oracle Database SQL Language Reference* for information about Oracle SQL function `XMLIsValid`
- *Oracle Database PL/SQL Packages and Types Reference* for information about `XMLType` methods `IsSchemaValid()`, `IsSchemaValidated()`, `SchemaValidate()`, and `setSchemaValidated()`

Validating XML Data Stored as XMLType: Examples

The examples in this section illustrate how to use Oracle SQL function `XMLIsValid` and `XMLType` methods `isSchemaValid()` and `schemaValidate()` to validate XML data being stored as `XMLType` in Oracle XML DB.

Example 11–5 and Example 11–6 show how to validate an XML instance against an XML schema using PL/SQL method `isSchemaValid`().

Example 11–5 Validating XML using Method ISSCHEMAVALID() in SQL

```
SELECT x.xmlcol.isSchemaValid('http://www.example.com/schemas/ipo.xsd',
                              'purchaseOrder')
    FROM po_tab x;
```

Example 11–6 Validating XML using Method ISSCHEMAVALID() in PL/SQL

```
DECLARE
  xml_instance XMLType;
BEGIN
  SELECT x.xmlcol INTO xml_instance FROM po_tab x WHERE id = 1;
  IF xml_instance.isSchemaValid('http://www.example.com/schemas/ipo.xsd') = 0
    THEN raise_application_error(-20500, 'Invalid Instance');
    ELSE DBMS_OUTPUT.put_line('Instance is valid');
  END IF;
END;
/
Instance is valid

PL/SQL procedure successfully completed.
```

`XMLType` method `schemaValidate()` can be used within `INSERT` and `UPDATE` triggers to ensure that all instances stored in the table are validated against the XML schema. Example 11–7 illustrates this.

Example 11–7 Validating XML using Method SCHEMAVALIDATE() within Triggers

```
DROP TABLE po_tab;
CREATE TABLE po_tab OF XMLType
  XMLSCHEMA "http://www.example.com/schemas/ipo.xsd" ELEMENT "purchaseOrder";

CREATE TRIGGER emp_trig BEFORE INSERT OR UPDATE ON po_tab FOR EACH ROW

DECLARE
  newxml XMLType;
BEGIn
  newxml := :new.OBJECT_VALUE;
  XMLTYPE.schemavalidate(newxml);
END;
```

```
/
```

Example 11–8 uses Oracle SQL function `XMLIsValid` to do the following:

- Verify that the `XMLType` instance conforms to the specified XML schema
- Ensure that the incoming XML documents are valid by using `CHECK` constraints

Example 11–8 Checking XML Validity using XMLISVALID within CHECK Constraints

```
DROP TABLE po_tab;
CREATE TABLE po_tab OF XMLType
   (CHECK(XMLIsValid(OBJECT_VALUE) = 1))
   XMLSCHEMA "http://www.example.com/schemas/ipo.xsd" ELEMENT "purchaseOrder";
```

Note: The validation functions and procedures described in section "Validating XMLType Instances" facilitate validation checking. Of these, `isSchemaValid` is the only one that raises errors that indicate why validation has failed.

12

Full-Text Search Over XML Data

This chapter describes full-text search over XML using Oracle. It explains how to use Oracle SQL function `contains` and Oracle XPath function `ora:contains`. These are the two functions used by Oracle Database to do full-text search over XML data.

> **See Also:** *Oracle Text Reference* and *Oracle Text Application Developer's Guide* for more information about Oracle Text

This chapter contains these topics:

- Overview of Full-Text Search for XML
- About the Full-Text Search Examples
- Overview of CONTAINS and ora:contains
- CONTAINS SQL Function
- ora:contains XQuery Function
- Text Path BNF Specification
- Support for Full-Text XML Examples

Overview of Full-Text Search for XML

Oracle supports full-text search on documents that are managed by the Oracle Database.

If your documents are XML, then you can use the XML structure of the document to restrict the full-text search. For example, you may want to find all purchase orders that contain the word "electric" using full-text search. If the purchase orders are in XML form, then you can restrict the search by finding all purchase orders that contain the word "electric" in a comment, or by finding all purchase orders that contain the word "electric" in a comment under line items.

If your XML documents are of type `XMLType`, then you can project the results of your query using the XML structure of the document. For example, after finding all purchase orders that contain the word "electric" in a comment, you may want to return just the comments, or just the comments that contain the word "electric".

Comparison of Full-Text Search and Other Search Types

Full-text search differs from structured search or substring search in the following ways:

- A full-text search looks for whole *words* rather than substrings. A substring search for comments that contain the *string* "law" can return a comment that contains "my *law*n is going wild". A full-text search for the *word* "law" cannot.
- A full-text search supports some language-based and word-based searches that substring searches do not. You can use a language-based search, for example, to find all the comments that contain a word with the same linguistic stem as "mouse", and Oracle Text finds "mouse" and "mice". You can use a word-based search, for example, to find all the comments that contain the word "lawn" within 5 words of "wild".
- A full-text search generally involves some notion of relevance. When you do a full-text search for all the comments that contain the word "lawn", for example, some results are more relevant than others. Relevance is often related to the number of times the search word (or similar words) occur in the document.

Searching XML Data

XML search is different from unstructured document search. In unstructured document search you generally search across a set of documents to return the documents that satisfy your text predicate. In XML search you often want to use the structure of the XML document to restrict the search. And you often want to return just the part of the document that satisfies the search.

Searching Documents using Full-Text Search and XML Structure

There are two ways to do a search that includes full-text search and XML structure:

- Include the structure inside the full-text predicate, using Oracle SQL function `contains`:

  ```
  WHERE contains(doc, 'electric INPATH (/purchaseOrder/items/item/comment)') > 0
  ```

  Function `contains` is an extension to SQL, and can be used in any query. It requires a `CONTEXT` full-text index.
- Include the full-text predicate inside the structure, using XPath function `ora:contains`:

  ```
  '/purchaseOrder/items/item/comment[ora:contains(text(), "electric")>0]'
  ```

  XPath function `ora:contains` is an extension to XPath, and can be used in a call to SQL/XML function `XMLQuery`, `XMLTable`, or `XMLExists`.

About the Full-Text Search Examples

This section describes details about the examples included in this chapter.

Roles and Privileges

To run the examples, you need database roles `CTXAPP`, `CONNECT`, and `RESOURCE`. You must also have `EXECUTE` privilege on the `CTXSYS` package `CTX_DDL`.

Schema and Data for Full-Text Search Examples

Examples in this chapter are based on "The Purchase Order Schema", W3C XML Schema Part 0: Primer.

See Also:

http://www.w3.org/TR/xmlschema-0/#POSchema

The data in the examples is from the document "Purchase-Order XML Document, po001.xml" on page 12-26.

The tables used in the examples of this chapter are defined in section "CREATE TABLE Statements""CREATE TABLE Statements". Some of the performance examples are, however, based on a larger table (purchase_orders_xmltype_big), which is included in the downloadable version only. See http://www.w3.org/TR/xmlschema-0/#po.xml.

Some of the examples here use data type VARCHAR2. Others use type XMLType. All examples that use VARCHAR2 also work with XMLType.

Overview of CONTAINS and ora:contains

This section contains these topics:

- Overview of SQL Function CONTAINS
- Overview of XPath Function ora:contains
- Comparison of CONTAINS and ora:contains

Overview of SQL Function CONTAINS

Oracle SQL function contains returns a positive number for rows where [schema.]column matches text_query. Otherwise, it returns zero. It requires an index of type CONTEXT. If there is no CONTEXT index on the column being searched, then contains raises an error.

CONTAINS Syntax

```
contains([schema.]column, text_query VARCHAR2 [,label NUMBER])
RETURN NUMBER
```

Example 12–1 shows a typical query that uses Oracle SQL function contains. It returns the id for each row in table purchase_orders where the doc column contains the word "lawn" and id is less than 25.

Example 12–1 Simple Query using Oracle SQL Function CONTAINS

```
SELECT id FROM purchase_orders WHERE contains(doc, 'lawn') > 0 AND id < 25;
```

Suppose doc is a column that contains a set of XML documents. You can do full-text search over doc, using its XML structure to restrict the query. The query in Example 12–2 returns id values for table purchaseorders where column doc contains the word "lawn" in the text() node of XML element comment.

Example 12–2 Restricting a Query using CONTAINS and WITHIN

```
SELECT id FROM purchase_orders WHERE contains(doc, 'lawn WITHIN comment') > 0;
```

More complex XML structure restrictions can be applied using the INPATH operator and an XPath expression. The query in Example 12–3 finds purchase orders that contain the word "electric" in the text() node of a comment element that is targeted by XPath expression/purchaseOrder/items/item/comment.

Example 12–3 Restricting a Query using CONTAINS and INPATH

```
SELECT id FROM purchase_orders
  WHERE contains(doc, 'electric INPATH (/purchaseOrder/items/item/comment)') > 0;
```

Overview of XPath Function ora:contains

XPath function `ora:contains` can be used in an XPath expression inside an XQuery expression or in a call to SQL/XML function `XMLQuery`, `XMLTable`, or `XMLExists`. It is used to restrict a structural search with a full-text predicate. It extends XPath through a standard mechanism: it is a user-defined function in the Oracle XML DB namespace, `ora`. It requires no index, but you can use an index with it to improve performance.

ora:contains Syntax

```
ora:contains(input_text NODE*, text_query STRING
             [,policy_name STRING]
             [,policy_owner STRING])
```

Function `ora:contains` returns a positive integer when the *`input_text`* matches *`text_query`* (the higher the number, the more relevant the match), and zero otherwise. When used in an XQuery expression, the XQuery return type is `xs:integer()`. When used in an XPath expression outside of an XQuery expression, the XPath return type is `number`.

Argument *`input_text`* must evaluate to a single text node or an attribute. The syntax and semantics of *`text_query`* in `ora:contains` are the same as *`text_query`* in `contains`, with the following restrictions:

- Argument *`text_query`* cannot include any structure operators (`WITHIN`, `INPATH`, or `HASPATH`).
- If the `weight` score-weighting operator is used, the weights are *ignored*.

Example 12–4 shows a call to `ora:contains` in the XPath parameter to `XMLExists`. Notice the namespace declaration that declares prefix `ora` as representing the Oracle XML DB namespace.

Example 12–4 ora:contains with an Arbitrarily Complex Text Query

```
SELECT id
  FROM purchase_orders_xmltype
  WHERE
    XMLExists(
      'declare namespace ora = "http://xmlns.oracle.com/xdb"; (: :)
       $d/purchaseOrder/comment
         [ora:contains(text(), "($lawns AND wild) OR flamingo") > 0]'
      PASSING doc AS "d");
```

See Also: "ora:contains XQuery Function" on page 12-17 for more on the `ora:contains` XPath function

Comparison of CONTAINS and ora:contains

Both Oracle SQL function `contains` and Oracle XPath function `ora:contains` let you combine searching on XML structure with full-text searching. These are the main differences between them:

Oracle SQL function `contains`:

- Needs a `CONTEXT` index to run. If there is no index, then an error is raised.

- Does an indexed search and is generally very fast.
- Returns a score (through Oracle SQL function `score`).
- Restricts a search based on documents (rows in a table) rather than nodes.
- *Cannot* be used for XML structure-based projection (extracting parts of an XML document).

Oracle XPath function `ora:contains`:

- Does not need an index to run, but you can use an index to improve performance.
- Might do an unindexed search, so it might be resource-intensive.
- Separates application logic from storing and indexing considerations.
- Does *not* return a score.
- Can be used for XML structure-based projection (extracting parts of an XML document).

Use Oracle SQL function `contains` when you want a fast, index-based, full-text search over XML documents, possibly with simple XML structure constraints. Use Oracle XPath function `ora:contains` when you need the flexibility of full-text search combined with XPath navigation (possibly without an index) or when you need to do projection, and you do not need a score.

CONTAINS SQL Function

This section contains these topics:

- Full-Text Search using SQL Function CONTAINS
- SCORE SQL Function
- Restricting the Scope of a CONTAINS Search
- Projecting the CONTAINS Result
- Indexing with a CONTEXT Index

Full-Text Search using SQL Function CONTAINS

The second argument to Oracle SQL function `contains`, `text_query`, is a string that specifies the full-text search. `text_query` has its own language, based on the SQL/MM Full-Text standard.

See Also:

- ISO/IEC 13249-2:2000, Information technology - Database languages - SQL Multimedia and Application Packages - Part 2: Full-Text, International Organization For Standardization, 2000
- *Oracle Text Reference* for more information about the operators in the `text_query` language

The examples in the rest of this section show some of the power of full-text search. They use only a few of the available operators. The example queries search over a `VARCHAR2` column (`PURCHASE_ORDERS.doc`) with a text index (index type `CTXSYS.CONTEXT`).

Full-Text Boolean Operators AND, OR, and NOT

The `text_query` language supports arbitrary combinations of `AND`, `OR`, and `NOT`. Precedence can be controlled using parentheses. The Boolean operators can be written in any of the following ways:

- `AND`, `OR`, `NOT`
- `and`, `or`, `not`
- `&`, `|`, `~`

Note that `NOT` is a *binary*, not a unary operator here. The expression `alpha NOT(beta)` is equivalent to `alpha AND` *unary-not*`(beta)`, where unary-not stands for unary negation.

> **See Also:** *Oracle Text Reference* for complete information about the operators you can use in `contains` and `ora:contains`

Example 12–5 CONTAINS Query with a Simple Boolean Operator

```
SELECT id FROM purchase_orders WHERE contains(doc, 'lawn AND wild') > 0;
```

Example 12–6 CONTAINS Query with Complex Boolean

```
SELECT id FROM purchase_orders
  WHERE contains(doc, '((lawn OR garden) AND (wild OR flooded)) NOT(flamingo)')
        > 0;
```

Full-Text Stemming: $

The `text_query` language supports stemmed search. Example 12–7 returns all documents that contain some word with the same linguistic stem as "lawns", so it finds "lawn" or "lawns". The stem operator is written as a dollar sign (`$`).

Example 12–7 CONTAINS Query with Stemming

```
SELECT id FROM purchase_orders WHERE contains(doc, '$(lawns)') > 0;
```

Combining Boolean and Stemming Operators

You can combine operators in the `text_query` language, as shown in Example 12–8.

Example 12–8 CONTAINS Query with Complex Query Expression

```
SELECT id FROM purchase_orders
  WHERE contains(doc, '($lawns AND wild) OR flamingo') > 0;
```

> **See Also:** *Oracle Text Reference* for a full list of `text_query` operators

SCORE SQL Function

Oracle SQL function `score` is related to Oracle SQL function `contains`. Function `score` can be used anywhere in a query. It is a measure of relevance, and it is especially useful when doing full-text searches across large document sets. Function `score` is typically returned as part of the query result, used in the `ORDER BY` clause, or both.

SCORE Syntax

```
score(label NUMBER) RETURN NUMBER
```

In Example 12–9, `score(10)` returns the score for each row in the result set. Oracle SQL function `score` returns the relevance of a row in the result set with respect to a particular call to function `contains`. A call to `score` is linked to a call to `contains` by a `LABEL` (in this case the number 10).

Example 12–9 Simple CONTAINS Query with SCORE

```
SELECT score(10), id FROM purchase_orders
  WHERE contains(doc, 'lawn', 10) > 0 AND score(10) > 2
  ORDER BY score(10) DESC;
```

Function `score` always returns `0` if, for the corresponding `contains` expression, argument `text_query` does not match `input_text`, according to the matching rules dictated by the text index. If the `contains text_query` matches the `input_text`, then `score` returns a number greater than `0` and less than or equal to `100`. This number indicates the relevance of the `text_query` to the `input_text`. A higher number means a better match.

If the `contains text_query` consists of only the `HASPATH` operator and a Text Path, the score is either `0` or `100`, because `HASPATH` tests for an exact match.

> **See Also:** *Oracle Text Reference* for details on how the score is calculated

Restricting the Scope of a CONTAINS Search

Oracle SQL function `contains` does a full-text search across the whole document, by default. In the example heres, a search for "lawn" with no structure restriction finds all purchase orders with the word "lawn" anywhere in them.

There are three ways to restrict `contains` queries using XML structure:

- `WITHIN`
- `INPATH`
- `HASPATH`

> **Note:** For the purposes of this discussion, consider *section* to be the same as an *XML node*.

WITHIN Structure Operator

The `WITHIN` operator restricts a query to some section within an XML document. A search for purchase orders that contain the word "lawn" somewhere inside a comment section might use `WITHIN`. Section names are case-sensitive.

Example 12–10 WITHIN

```
SELECT id FROM purchase_orders WHERE contains(DOC, 'lawn WITHIN comment') > 0;
```

Nested WITHIN You can restrict the query further by nesting `WITHIN`. Example 12–11 finds all documents that contain the word "`lawn`" within a section "`comment`", where that occurrence of "`lawn`" is also within a section "`item`".

Example 12–11 Nested WITHIN

```
SELECT id FROM purchase_orders
  WHERE contains(doc, '(lawn WITHIN comment) WITHIN item') > 0;
```

Example 12–11 returns no rows. Our sample purchase order does contain the word "lawn" within a comment. But the only comment within an item is "Confirm this is electric". So the nested WITHIN query returns no rows.

WITHIN Attributes You can also search within attributes. Example 12–12 finds all purchase orders that contain the word 10 in the orderDate attribute of a purchaseOrder element.

Example 12–12 WITHIN an Attribute

```
SELECT id FROM purchase_orders
  WHERE contains(doc, '10 WITHIN purchaseOrder@orderDate') > 0;
```

By default, the minus sign ("-") is treated as a word separator: "1999-10-20" is treated as the three words "1999", "10" and "20". So this query returns one row.

Text in an attribute is not a part of the main searchable document. A search for 10 without qualifying the text_query with WITHIN purchaseOrder@orderDate returns no rows.

You cannot search attributes in a nested WITHIN.

WITHIN and AND Suppose you want to find purchase orders that contain two words within a comment section: "lawn" and "electric". There can be more than one comment section in a purchaseOrder. So there are two ways to write this query, with two distinct results.

If you want to find purchase orders that contain both words, where each word occurs in *some comment section*, you would write a query like Example 12–13.

Example 12–13 WITHIN and AND: Two Words in Some Comment Section

```
SELECT id FROM purchase_orders
  WHERE contains(doc, '(lawn WITHIN comment) AND (electric WITHIN comment)') > 0;
```

If you run this query against the purchaseOrder data, then it returns 1 row. Note that the parentheses are not needed in this example, but they make the query more readable.

If you want to find purchase orders that contain both words, where both words occur *in the same comment*, you would write a query like Example 12–14.

Example 12–14 WITHIN and AND: Two Words in the Same Comment

```
SELECT id FROM purchase_orders
  WHERE contains(doc, '(lawn AND electric) WITHIN comment') > 0;
```

The query in Example 12–14 returns no rows. The query in Example 12–15, which omits the parentheses around lawn AND electric, returns one row.

Example 12–15 WITHIN and AND: No Parentheses

```
SELECT id FROM purchase_orders
  WHERE contains(doc, 'lawn AND electric WITHIN comment') > 0;
```

Operator WITHIN has a higher precedence than AND, so Example 12–15 is parsed as Example 12–16.

Example 12–16 WITHIN and AND: Parentheses Illustrating Operator Precedence

```
SELECT id FROM purchase_orders
  WHERE contains(doc, 'lawn AND (electric WITHIN comment)') > 0;
```

Definition of Section The preceding examples have used the `WITHIN` operator to search within a section. A **section** can be a:

- **path** or **zone** section

  This is a concatenation, in document order, of all text nodes that are descendants of a node, with whitespace separating the text nodes. To convert from a node to a zone section, you must serialize the node and replace all tags with whitespace. path sections have the same scope and behavior as zone sections, except that path sections support queries with `INPATH` and `HASPATH` structure operators.

- **field** section

  This is the same as a zone section, except that repeating nodes in a document are concatenated into a single section, with whitespace as a separator.

- attribute section
- special section (sentence or paragraph)

> **See Also:** *Oracle Text Reference* for more information about special sections

INPATH Structure Operator

Operator `WITHIN` provides an easy and intuitive way to express simple structure restrictions in the `text_query`. For queries that use abundant XML structure, you can use operator `INPATH` plus a text path instead of nested `WITHIN` operators.

Operator `INPATH` takes a `text_query` on the left and a Text Path, enclosed in parentheses, on the right. Example 12–17 finds `purchaseOrders` that contain the word "`electric`" in the path `/purchaseOrder/items/item/comment`.

Example 12–17 Structure Inside Full-Text Predicate: INPATH

```
SELECT id FROM purchase_orders
  WHERE contains(doc, 'electric INPATH (/purchaseOrder/items/item/comment)') > 0;
```

The scope of the search in Example 12–17 is the section indicated by the Text Path. The query in Example 12–18 uses a broader path than the query in Example 12–17, but it too returns one row.

Example 12–18 Structure Inside Full-Text Predicate: INPATH

```
SELECT id FROM purchase_orders
  WHERE contains(doc, 'electric INPATH (/purchaseOrder/items)') > 0;
```

Text Path The syntax and semantics of Text Path are based on the w3c XPath 1.0 recommendation. Simple path expressions are supported (abbreviated syntax only), but functions are not. The following examples are meant to give the general flavor.

> **See Also:**
>
> - `http://www.w3.org/TR/xpath` for information about the W3C XPath 1.0 recommendation
> - "Text Path BNF Specification" on page 12-25 for the Text Path grammar

Example 12–19 finds all purchase orders that contain the word "electric" in a comment element that is the child of an item element with a partNum attribute whose value is "872-AA", which in turn is the child of an items element that is any number of levels under the root node.

Example 12–19 INPATH with Complex Path Expression (1)

```
SELECT id FROM purchase_orders
  WHERE contains(doc, 'electric INPATH (//items/item[@partNum="872-AA"]/comment)')
        > 0;
```

Example 12–20 finds all purchase orders that contain the word "lawnmower" in a third-level item element (or any of its descendants) that has a comment element descendant at any level. This query returns one row. The scope of the query is *not* a comment element, but the set of items elements that each have a comment element as a descendant.

Example 12–20 INPATH with Complex Path Expression (2)

```
SELECT id FROM purchase_orders
  WHERE contains(doc, 'lawnmower INPATH (/*/*/item[.//comment])') > 0;
```

Text Path Compared to XPath The Text Path language differs from the XPath language in the following ways:

- Not all XPath operators are included in the Text Path language.
- XPath built-in functions are not included in the Text Path language.
- Text Path language operators are case-insensitive.
- If you use = inside a filter (brackets), then matching follows text-matching rules.

  Rules for case-sensitivity, normalization, stopwords and whitespace depend on the text index definition. To emphasize this difference, this kind of equality is referred to here as text-equals.
- Namespace support is not included in the Text Path language.

  The name of an element, including a namespace prefix if it exists, is treated as a string. So two different namespace prefixes that map to the same namespace URI are not treated as equivalent in the Text Path language.
- In a Text Path, the context is always the root node of the document.

  So in the purchase-order data, purchaseOrder/items/item, /purchaseOrder/items/item, and ./purchaseOrder/items/item are all equivalent.
- If you want to search within an attribute value, then the direct parent of the attribute must be specified (wildcards cannot be used).
- A Text Path may not end in a wildcard (*).

> **See Also:** "Text Path BNF Specification" on page 12-25 for the Text Path grammar

Nested INPATH You can nest INPATH expressions. The context for the Text Path is always the root node. It is not changed by a nested INPATH.

Example 12–21 finds purchase orders that contain the word "electric" inside a comment element at any level, where the occurrence of that word is also in an items element that is a child of the top-level purchaseOrder element.

Example 12–21 Nested INPATH

```
SELECT id FROM purchase_orders
  WHERE contains(doc,
                 '(electric INPATH (//comment)) INPATH (/purchaseOrder/items)')
        > 0;
```

This nested `INPATH` query could be written more concisely as shown in Example 12–22.

Example 12–22 Nested INPATH Rewritten

```
SELECT id FROM purchase_orders
  WHERE contains(doc, 'electric INPATH (/purchaseOrder/items//comment)') > 0;
```

HASPATH Structure Operator

Operator `HASPATH` takes only one operand: a Text Path, enclosed in parentheses, on the right. Use `HASPATH` when you want to find documents that contain a particular section in a particular path, possibly with predicate `=`. This is a path search rather than a full-text search. You can check for the existence of a section, or you can match the contents of a section, but you cannot do word searches. If your data is of type `XMLType`, then consider using SQL/XML function `XMLExists` instead of structure operator `HASPATH`.

Example 12–23 finds `purchaseOrders` that have some item that has a `USPrice`.

Example 12–23 Simple HASPATH

```
SELECT id FROM purchase_orders
  WHERE contains(DOC, 'HASPATH (/purchaseOrder//item/USPrice)') > 0;
```

Example 12–24 finds `purchaseOrders` that have some item that has a `USPrice` that text-equals `"148.95"`.

> **See Also:** "Text Path Compared to XPath" on page 12-10 for an explanation of text-equals

Example 12–24 HASPATH Equality

```
SELECT id FROM purchase_orders
  WHERE contains(doc, 'HASPATH (/purchaseOrder//item/USPrice="148.95")') > 0;
```

`HASPATH` can be combined with other `contains` operators such as `INPATH`. Example 12–25 finds `purchaseOrders` that contain the word `electric` anywhere in the document *and* have some `item` that has a `USPrice` that text-equals `148.95` *and* contain `10` in the `purchaseOrder` attribute `orderDate`.

Example 12–25 HASPATH with Other Operators

```
SELECT id FROM purchase_orders
  WHERE contains(doc,
                 'electric
                  AND HASPATH (/purchaseOrder//item/USPrice="148.95")
                  AND 10 INPATH (/purchaseOrder/@orderDate)')
        > 0;
```

Projecting the CONTAINS Result

The result of a SQL query with a `contains` expression in the `WHERE` clause is always a set of rows (and possibly `score` information), or a projection over the rows that match the query.

If you want to return only a part of each XML document that satisfies a `contains` expression, then use SQL/XML function `XMLQuery`. The examples in this section use the `XMLType` table `purchase_orders_xmltype`.

Example 12–26 finds `purchaseOrder`s that contain the word "`electric`" inside a `comment` element that is a descendant of the top-level element `purchaseOrder`. Instead of returning the ID of the row for each result, `XMLQuery` is used to return only the `comment` element.

Example 12–26 Scoping the Results of a CONTAINS Query

```
SELECT XMLQuery('declare namespace ora = "http://xmlns.oracle.com/xdb"; (: :)
                 $d/purchaseOrder//comment'
                PASSING doc AS "d" RETURNING CONTENT) "Item Comment"
  FROM purchase_orders_xmltype
  WHERE CONTAINS(doc, 'electric INPATH (/purchaseOrder//comment)') > 0;
```

The result of Example 12–26 is *two* instances of element `comment`. Function `contains` indicates which rows contain the word "`electric`" inside a `comment` element (the row with `ID = 1`), and function `XMLQuery` extracts all of the instances of element `comment` in the document at that row. There are two instances of element `comment` inside the `purchaseOrder` element, and the query returns both of them.

This might not be what you want. If you want the query to return only the instances of element `comment` that satisfy the `contains` expression, then you must repeat that predicate in the XQuery expression passed to `XMLQuery`. You do that using XPath function `ora:contains`. Example 12–27 illustrates this.

Example 12–27 Projecting the Result of a CONTAINS Query using ora:contains

```
SELECT XMLQuery('declare namespace ora = "http://xmlns.oracle.com/xdb"; (: :)
                 $d/purchaseOrder/items/item/comment
                   [ora:contains(text(), "electric") > 0]'
                PASSING doc AS "d" RETURNING CONTENT) "Item Comment"
  FROM purchase_orders_xmltype
  WHERE CONTAINS(doc, 'electric INPATH (/purchaseOrder/items/item/comment)') > 0;
```

Indexing with a CONTEXT Index

This section contains these topics:

- Introduction to CONTEXT Indexes
- Effect of a CONTEXT Index on CONTAINS
- CONTEXT Index Preferences
- Introduction to Section Groups

Introduction to CONTEXT Indexes

The general-purpose full-text index type is `CONTEXT`, which is owned by database user `CTXSYS`. To create a default full-text index, use the regular SQL `CREATE INDEX` command, and add the clause `INDEXTYPE IS CTXSYS.CONTEXT`, as shown in Example 12–28.

Example 12–28 Simple CONTEXT Index on Table PURCHASE_ORDERS

```
CREATE INDEX po_index ON purchase_orders(doc)
  INDEXTYPE IS CTXSYS.CONTEXT;
```

You have many choices available when building a full-text index. These choices are expressed as indexing **preferences**. To use an indexing preference, add the PARAMETERS clause to CREATE INDEX, as shown in Example 12–29.

> **See Also:** "CONTEXT Index Preferences" on page 12-14

Example 12–29 Simple CONTEXT Index on XMLType Table with Path Section Group

```
CREATE INDEX po_index ON purchase_orders(doc)
  INDEXTYPE IS CTXSYS.CONTEXT
  PARAMETERS ('section group CTXSYS.PATH_SECTION_GROUP');
```

Oracle Text provides other index types, such as CTXCAT and CTXRULE, which are outside the scope of this chapter.

> **See Also:** *Oracle Text Reference* for more information about CONTEXT indexes

CONTEXT Index on XMLType Table You can build a CONTEXT index on any data that contains text. Example 12–28 creates a CONTEXT index on a VARCHAR2 column. The syntax to create a CONTEXT index on a column of type CHAR, VARCHAR, VARCHAR2, BLOB, CLOB, BFILE, XMLType, or URIType is the same. Example 12–30 creates a CONTEXT index on a column of type XMLType. The section group defaults to PATH_SECTION_GROUP.

Example 12–30 Simple CONTEXT Index on XMLType Column

```
CREATE INDEX po_index_xmltype ON purchase_orders_xmltype(doc)
  INDEXTYPE IS CTXSYS.CONTEXT;
```

If you have an XMLType table, then you must use object syntax to create the CONTEXT index, as shown in Example 12–31.

Example 12–31 Simple CONTEXT Index on XMLType Table

```
CREATE INDEX po_index_xmltype_table
  ON purchase_orders_xmltype_table (OBJECT_VALUE)
  INDEXTYPE IS CTXSYS.CONTEXT;
```

You can query the table as shown in Example 12–32.

Example 12–32 CONTAINS Query on XMLType Table

```
SELECT XMLCast(XMLQuery(
                 'declare namespace ora = "http://xmlns.oracle.com/xdb"; (: :)
                  $p/purchaseOrder/@orderDate'
                 PASSING po.OBJECT_VALUE AS "p" RETURNING CONTENT)
               AS DATE) "Order Date"
  FROM purchase_orders_xmltype_table po
  WHERE contains(po.OBJECT_VALUE, 'electric INPATH (/purchaseOrder//comment)')
        > 0;
```

Maintaining a CONTEXT Index The CONTEXT index, like most full-text indexes, is asynchronous. When indexed data is changed, the CONTEXT index might not change until you take some action, such as calling a procedure to synchronize the index.

The CONTEXT index can become fragmented over time. A fragmented index uses more space and leads to slower queries. There are a number of ways to optimize (defragment) the CONTEXT index.

> **See Also:** *Oracle Text Reference* for more information about CONTEXT index maintenance

Roles and Privileges You do not need any special privileges to create a CONTEXT index. You need the CTXAPP role to create and delete preferences and to use the Oracle Text PL/SQL packages. You must also have EXECUTE privilege on the CTXSYS package CTX_DDL.

Effect of a CONTEXT Index on CONTAINS

To use Oracle SQL function contains, you must create an index of type CONTEXT. If you call contains, and the column given in the first argument does not have an index of type CONTEXT, then an error is raised.

The syntax and semantics of text_query depend on the choices you make when you build the CONTEXT index. For example:

- What counts as a word?
- Are very common words processed?
- What is a common word?
- Is the text search case-sensitive?
- Can the text search include themes (concepts) in addition to keywords?

CONTEXT Index Preferences

A preference can be considered a collection of indexing choices. Preferences include section group, datastore, filter, wordlist, stoplist and storage. This section shows how to set up a lexer preference to make searches case-sensitive.

You can use procedure CTX_DDL.create_preference (or CTX_DDL.create_stoplist) to create a preference. Override default choices in that preference group by setting attributes of the new preference, using procedure CTX_DDL.set_attribute. Then use the preference in a CONTEXT index by including *preference type preference_name* in the PARAMETERS string of CREATE INDEX.

Once a preference has been created, you can use it to build any number of indexes.

Making Search Case-Sensitive Full-text searches with contains are *case-insensitive* by default. That is, when matching words in text_query against words in the document, case is not considered. Section names and attribute names, however, are always *case-sensitive*.

If you want full-text searches to be case-sensitive, then you need to make that choice when building the CONTEXT index. Example 12–33 returns 1 row, because "HURRY" in text_query matches "Hurry" in the purchaseOrder with the default case-insensitive index.

Example 12–33 CONTAINS: Default Case Matching

```
SELECT id FROM purchase_orders
  WHERE contains(doc, 'HURRY INPATH (/purchaseOrder/comment)') > 0;
```

Example 12–34 creates a new lexer preference `my_lexer`, with the attribute `mixed_case` set to `TRUE`. It also sets printjoin characters to "-" and "!" and ",". You can use the same preferences for building `CONTEXT` indexes and for building policies.

See Also: *Oracle Text Reference* for a full list of lexer attributes

Example 12–34 Create a Preference for Mixed Case

```
BEGIN
  CTX_DDL.create_preference(PREFERENCE_NAME  =>  'my_lexer',
                            OBJECT_NAME      =>  'BASIC_LEXER');

  CTX_DDL.set_attribute(PREFERENCE_NAME  =>  'my_lexer',
                        ATTRIBUTE_NAME   =>  'mixed_case',
                        ATTRIBUTE_VALUE  =>  'TRUE');

  CTX_DDL.set_attribute(PREFERENCE_NAME  =>  'my_lexer',
                        ATTRIBUTE_NAME   =>  'printjoins',
                        ATTRIBUTE_VALUE  =>  '-,!');
END;
/
```

Example 12–35 builds a `CONTEXT` index using the new `my_lexer` lexer preference. It uses preference `preference-case-mixed`.

Example 12–35 CONTEXT Index on PURCHASE_ORDERS Table, Mixed Case

```
CREATE INDEX po_index ON purchase_orders(doc)
  INDEXTYPE IS CTXSYS.CONTEXT
  PARAMETERS('lexer my_lexer section group CTXSYS.PATH_SECTION_GROUP');
```

Example 12–33 returns no rows, because "`HURRY`" in `text_query` no longer matches "`Hurry`" in the `purchaseOrder`. Example 12–36 returns one row, because the `text_query` term "`Hurry`" exactly matches the word "`Hurry`" in the `purchaseOrder`.

Example 12–36 CONTAINS: Mixed (Exact) Case Matching

```
SELECT id FROM purchase_orders
  WHERE contains(doc, 'Hurry INPATH (/purchaseOrder/comment)') > 0;
```

Introduction to Section Groups

One of the choices you make when creating a `CONTEXT` index is section group. A section group instance is based on a section group type. The section group type specifies the kind of structure in your documents, and how to index (and therefore search) that structure. The section group instance may specify which structure elements are indexed. Most users either take the default section group or use a predefined section group.

Choosing a Section Group Type The section group types useful in XML searching are:

- `PATH_SECTION_GROUP`

  Choose this when you want to use `WITHIN`, `INPATH` and `HASPATH` in queries, and you want to be able to consider all sections to scope the query.

- `XML_SECTION_GROUP`

  Choose this when you want to use `WITHIN`, but not `INPATH` and `HASPATH`, in queries, and you want to be able to consider only explicitly-defined sections to scope the query. `XML_SECTION_GROUP` section group type supports `FIELD`

sections in addition to ZONE sections. In some cases FIELD sections offer significantly better query performance.

- AUTO_SECTION_GROUP

  Choose this when you want to use WITHIN, but not INPATH and HASPATH, in queries, and you want to be able to consider most sections to scope the query. By default all sections are indexed (available for query restriction). You can specify that some sections are *not* indexed (by defining STOP sections).

- NULL_SECTION_GROUP

  Choose this when defining no XML sections.

Other section group types include:

- BASIC_SECTION_GROUP
- HTML_SECTION_GROUP
- NEWS_SECTION_GROUP

Oracle recommends that most users with XML full-text search requirements use PATH_SECTION_GROUP. Some users might prefer XML_SECTION_GROUP with FIELD sections. This choice generally gives better query performance and a smaller index, but it is limited to documents with fielded structure (searchable nodes are all leaf nodes that do not repeat).

> **See Also:** *Oracle Text Reference* for a detailed description of the XML_SECTION_GROUP section group type

Choosing a Section Group When choosing a section group to use with your index, you can choose a supplied section group, take the default, or create a new section group based on the section group type you have chosen.

There are supplied section groups for section group types PATH_SECTION_GROUP, AUTO_SECTION_GROUP, and NULL_SECTION_GROUP. The supplied section groups are owned by CTXSYS and have the same name as their section group types. For example, the supplied section group of section group type PATH_SECTION_GROUP is CTXSYS.PATH_SECTION_GROUP.

There is no supplied section group for section group type XML_SECTION_GROUP, because a default XML_SECTION_GROUP would be empty and therefore meaningless. If you want to use section group type XML_SECTION_GROUP, then you must create a new section group and specify each node that you want to include as a section.

When you create a CONTEXT index on data of type XMLType, the default section group is the supplied section group CTXSYS.PATH_SECTION_GROUP. If the data is VARCHAR or CLOB, then the default section group is CTXSYS.NULL_SECTION_GROUP.

> **See Also:** *Oracle Text Reference* for instructions on creating your own section group

To associate a section group with an index, add section group <section group name> to the PARAMETERS string, as in Example 12–37.

Example 12–37 Simple CONTEXT Index on purchase_orders Table with Path Section Group

```
CREATE INDEX po_index ON purchase_orders(doc)
  INDEXTYPE IS CTXSYS.CONTEXT
  PARAMETERS ('section group CTXSYS.PATH_SECTION_GROUP');
```

ora:contains XQuery Function

Function `ora:contains` is an Oracle-defined XQuery (XPath) function for use in the XQuery expression argument to SQL/XML functions `XMLQuery`, `XMLTable`, and `XMLExists`.

When you use `ora:contains` you must also supply a namespace declaration that maps prefix `ora` to the Oracle XML DB namespace, `xmlns:ora="http://xmlns.oracle.com/xdb"`.

Function `ora:contains` returns a number. It does *not* return a score. It returns a positive number if the `text_query` matches the `input_text`. Otherwise it returns zero.

Full-Text Search using XQuery Function ora:contains

The `ora:contains` argument *`text_query`* is a string that specifies the full-text search. The `ora:contains` *`text_query`* is the same as the `contains` *`text_query`*, with the following restrictions:

- `ora:contains` *`text_query`* must *not* include any of the structure operators `WITHIN`, `INPATH`, or `HASPATH`
- `ora:contains` *`text_query`* can include the score weighting operator `weight(*)`, but weights are *ignored*

If you include any of the following in the `ora:contains` *`text_query`*, the query *cannot* use a `CONTEXT` index:

- Score-based operator `MINUS` (`-`) or `threshold` (`>`)
- Selective, corpus-based expansion operator `FUZZY` (`?`) or `soundex` (`!`)

See Also: "XPath Rewrite and CONTEXT Indexes" on page 12-23

Example 12–4 shows a full-text search using an arbitrary combination of Boolean operators and `$` (stemming).

See Also:

- "Full-Text Search using SQL Function CONTAINS" on page 12-5 for a description of full-text operators
- *Oracle Text Reference* for a full list of the operators you can use in `contains` and `ora:contains`

Matching rules are defined by the *policy*, *`policy_owner.policy_name`*. If *`policy_owner`* is absent, then the policy owner defaults to the current user. If both *`policy_name`* and *`policy_owner`* are absent, then the policy defaults to `CTXSYS.DEFAULT_POLICY_ORACONTAINS`.

Restricting the Scope of an ora:contains Query

When you use `ora:contains` in an XPath expression, the scope is defined by argument *`input_text`*. This argument is evaluated in the current XPath context. If the result is a single text node or an attribute, then that node is the target of the `ora:contains` search. If *`input_text`* does not evaluate to a single text node or an attribute, an error is raised.

The policy determines the matching rules for `ora:contains`. The section group associated with the default policy for `ora:contains` is of type `NULL_SECTION_GROUP`.

`ora:contains` can be used anywhere in an XPath expression, and its *`input_text`* argument can be any XPath expression that evaluates to a single text node or an attribute.

Projecting the ora:contains Result

If you want to return only a part of each XML document, then use function `XMLQuery` to project a node sequence, possibly applying `XMLCast` to the result to project the scalar value of a node.

Example 12–38 returns the `orderDate` for each purchase order that has a `comment` that contains the word "`lawn`".

Example 12–38 Using ora:contains with XMLQuery and XMLExists

```
SELECT XMLCast(XMLQuery(
                 'declare namespace ora = "http://xmlns.oracle.com/xdb"; (: :)
                  $d/purchaseOrder/@orderDate'
                 PASSING doc AS "d" RETURNING CONTENT)
               AS DATE) "Order date"
  FROM purchase_orders_xmltype
  WHERE XMLExists(
          'declare namespace ora = "http://xmlns.oracle.com/xdb"; (: :)
           $d/purchaseOrder/comment
             [ora:contains(text(), "($lawns AND wild) OR flamingo") > 0]'
          PASSING doc AS "d");
```

Function `XMLExists` restricts the result to rows (documents) where the `purchaseOrder` element includes some `comment` that contains the word "`lawn`". Function `XMLQuery` then returns the value of attribute `orderDate` from those `purchaseOrder` elements. Function `XMLCast` casts this result as a SQL `DATE` value.

If `//comment` had been extracted, then both comments from the sample document would have been returned, not just the comment that matches the `WHERE` clause.

> **See Also:** Example 12–26, "Scoping the Results of a CONTAINS Query" on page 12-12

Policies for ora:contains Queries

The `CONTEXT` index on a column determines the semantics of `contains` queries on that column. Because `ora:contains` does not rely on a supporting index, some other means must be found to provide many of the same choices when doing `ora:contains` queries. A **policy** is a collection of preferences that can be associated with an `ora:contains` query to give the same sort of semantic control as the indexing choices give to the `contains` user.

Introduction to Policies for ora:contains Queries

When using Oracle SQL function `contains`, indexing preferences affect the semantics of the query. You create a preference using procedure `CTX_DDL.create_preference` (or `CTX_DDL.create_stoplist`). You override default choices by setting attributes of the new preference, using procedure `CTX_DDL.set_attribute`. Then you use the preference in a `CONTEXT` index by including *`preference_type preference_name`* in the `PARAMETERS` string of `CREATE INDEX`.

See Also: "CONTEXT Index Preferences" on page 12-14

Because `ora:contains` does not have a supporting index, a different mechanism is needed to apply preferences to a query. That mechanism is a policy, consisting of a collection of preferences, and it is used as a parameter to `ora:contains`.

Policy Example: Supplied Stoplist Example 12–39 creates a policy with an empty stopwords list.

Example 12–39 Create a Policy to Use with ora:contains

```
BEGIN
  CTX_DDL.create_policy(POLICY_NAME  =>  'my_nostopwords_policy',
                        STOPLIST     =>  'CTXSYS.EMPTY_STOPLIST');
END;
/
```

For simplicity, this policy consists of an empty stoplist, which is owned by user `CTXSYS`. You could create a new stoplist to include in this policy, or you could reuse a stoplist (or lexer) definition that you created for a `CONTEXT` index.

Refer to this policy in an `ora:contains` expression to search for all words, including the most common ones (stopwords). Example 12–40 returns zero comments, because `"is"` is a stopword by default and cannot be queried.

Example 12–40 Finding a Stopword using ora:contains

```
SELECT id FROM purchase_orders_xmltype
  WHERE XMLExists(
          'declare namespace ora = "http://xmlns.oracle.com/xdb"; (: :)
           $d/purchaseOrder/comment[ora:contains(text(), "is") > 0]'
          PASSING doc AS "d");
```

Example 12–41 uses the policy created in Example 12–39 to specify an empty stopword list. This query finds `"is"` and returns 1 comment.

Example 12–41 Finding a Stopword using ora:contains and Policy my_nostopwords_policy

```
SELECT id FROM purchase_orders_xmltype
  WHERE XMLExists(
          'declare namespace ora = "http://xmlns.oracle.com/xdb"; (: :)
           $d/purchaseOrder/comment
             [ora:contains(text(), "is", "MY_NOSTOPWORDS_POLICY") > 0]'
          PASSING doc AS "d");
```

Example 12–41 uses policy `my_nostopwords_policy`. This policy was implicitly named as all uppercase, in Example 12–39. Because XPath is case-sensitive, it must be referred to in the XPath predicate using all uppercase: `MY_NOSTOPWORDS_POLICY`, not `my_nostopwords_policy`.

Effect of Policies on ora:contains

The `ora:contains` policy affects the matching semantics of `text_query`. The `ora:contains` policy may include a lexer, stoplist, wordlist preference, or any combination of these. Other preferences that can be used to build a `CONTEXT` index are not applicable to `ora:contains`. The effects of the preferences are as follows:

- The wordlist preference tweaks the semantics of the stem operator.

- The stoplist preference defines which words are too common to be indexed (searchable).
- The lexer preference defines how words are tokenized and matched. For example, it defines which characters count as part of a word and whether matching is case-sensitive.

See Also:

- "Policy Example: Supplied Stoplist" on page 12-19 for an example of building a policy with a predefined stoplist
- "Policy Example: User-Defined Lexer" on page 12-20 for an example of a case-sensitive policy

Policy Example: User-Defined Lexer When you search for a document that contains a particular word, you usually want the search to be case-insensitive. If you do a search that is case-sensitive, then you often miss some expected results. For example, if you search for `purchaseOrders` that contain the phrase "baby monitor", then you would not expect to miss our example document just because the phrase is written "Baby Monitor".

Full-text searches with `ora:contains` are case-insensitive by default. Section names and attribute names, however, are always case-sensitive.

If you want full-text searches to be case-sensitive, then you need to make that choice when you create a policy. You can use this procedure:

1. Create a preference using the procedure `CTX_DDL.create_preference` (or `CTX_DDL.create_stoplist`).
2. Override default choices in that preference object by setting attributes of the new preference, using procedure `CTX_DDL.set_attribute`.
3. Use the preference as a parameter to `CTX_DDL.create_policy`.
4. Use the policy name as the third argument to `ora:contains` in a query.

Once you have created a preference, you can reuse it in other policies or in `CONTEXT` index definitions. You can use any policy with any `ora:contains` query.

Example 12–42 returns 1 row, because "`HURRY`" in `text_query` matches "`Hurry`" in the `purchaseOrder` with the default case-insensitive index.

Example 12–42 ora:contains, Default Case-Sensitivity

```
SELECT id FROM purchase_orders_xmltype
  WHERE XMLExists(
          'declare namespace ora = "http://xmlns.oracle.com/xdb"; (: :)
           $d/purchaseOrder/comment[ora:contains(text(), "HURRY") > 0]'
          PASSING doc AS "d");
```

Example 12–43 creates a new lexer preference `my_lexer`, with the attribute `mixed_case` set to `TRUE`. It also sets `printjoin` characters to "`-`", "`!`" and "`,`". You can use the same preferences for building `CONTEXT` indexes and for building policies.

See Also: *Oracle Text Reference* for a full list of lexer attributes

Example 12–43 Create a Preference for Mixed Case

```
BEGIN
  CTX_DDL.create_preference(PREFERENCE_NAME  =>  'my_lexer',
                            OBJECT_NAME      =>  'BASIC_LEXER');
```

```
  CTX_DDL.set_attribute(PREFERENCE_NAME  =>  'MY_LEXER',
                        ATTRIBUTE_NAME   =>  'MIXED_CASE',
                        ATTRIBUTE_VALUE  =>  'TRUE');
  CTX_DDL.set_attribute(PREFERENCE_NAME  =>  'my_lexer',
                        ATTRIBUTE_NAME   =>  'printjoins',
                        ATTRIBUTE_VALUE  =>  '-,!');
END;
/
```

Example 12–44 creates a new policy `my_policy` and specifies only the lexer. All other preferences are defaulted. Example 12–44 uses `preference-case-mixed`.

Example 12–44 Create a Policy with Mixed Case (Case-Insensitive)

```
BEGIN
  CTX_DDL.create_policy(POLICY_NAME  => 'my_policy',
                        LEXER        => 'my_lexer');
END;
/
```

Example 12–45 uses the new policy in a query. It returns no rows, because "`HURRY`" in `text_query` no longer matches "`Hurry`" in the `purchaseOrder`.

Example 12–45 ora:contains, Case-Sensitive (1)

```
SELECT id FROM purchase_orders_xmltype
  WHERE XMLExists(
          'declare namespace ora = "http://xmlns.oracle.com/xdb"; (: :)
           $d/purchaseOrder/comment
             [ora:contains(text(), "HURRY", "my_policy") > 0]'
          PASSING doc AS "d");
```

Example 12–46 returns one row, because the `text_query` term "`Hurry`" exactly matches the text "`Hurry`" in the `comment` element.

Example 12–46 ora:contains, Case-Sensitive (2)

```
SELECT id FROM purchase_orders_xmltype
  WHERE XMLExists(
          'declare namespace ora = "http://xmlns.oracle.com/xdb"; (: :)
           $d/purchaseOrder/comment[ora:contains(text(), "Hurry") > 0]'
          PASSING doc AS "d");
```

Policy Defaults

The policy argument to `ora:contains` is optional. If it is omitted, then the query uses the default policy `CTXSYS.DEFAULT_POLICY_ORACONTAINS`.

When you create a policy for use with `ora:contains`, you do not need to specify every preference. In Example 12–44 on page 12-21, for example, only the lexer preference was specified. For the preferences that are not specified, `CREATE_POLICY` uses the default preferences:

- `CTXSYS.DEFAULT_LEXER`
- `CTXSYS.DEFAULT_STOPLIST`
- `CTXSYS.DEFAULT_ WORDLIST`

Creating a policy follows copy semantics for preferences and their attributes, just as creating a `CONTEXT` index follows copy semantics for index metadata.

Performance of ora:contains

The `ora:contains` XPath function does not depend on a supporting index. `ora:contains` is very flexible. But if you use it to search across large amounts of data without an index, then it can also be resource-intensive. This section shows how to get the best performance from queries that include XPath expressions with XPath function `ora:contains`.

> **Note:** Function-based indexes can also be very effective in speeding up XML queries, but they are not generally applicable to Text queries.

The examples in this section use table `purchase_orders_xmltype_big`. This has the same table structure and XML schema as `purchase_orders_xmltype`, but it has around 1,000 rows. Each row has a unique ID (in column `id`), and some different text in `/purchaseOrder/items/item/comment`. Where an execution plan is shown, it was produced using the SQL*Plus command `AUTOTRACE`. Execution plans can also be produced using SQL commands `TRACE` and `TKPROF`. A description of commands `AUTOTRACE`, `trace` and `tkprof` is outside the scope of this chapter.

This section contains these topics:

- Use a Primary Filter in the Query
- XPath Rewrite and CONTEXT Indexes

Use a Primary Filter in the Query

Because `ora:contains` is relatively costly to process, Oracle recommends that you write queries that include a primary filter wherever possible. This minimizes the number of rows processed by `ora:contains`.

Example 12–47 examines each row in a table (a full table scan), as shown by the execution plan. In this example, `ora:contains` is evaluated for each row.

Example 12–47 ora:contains in Large Table

```
SELECT id FROM purchase_orders_xmltype_big
  WHERE XMLExists(
           'declare namespace ora = "http://xmlns.oracle.com/xdb"; (: :)
            $d/purchaseOrder/comment[ora:contains(text(), "constitution") > 0]'
           PASSING doc AS "d");
```

```
Execution Plan
----------------------------------------------------------------------------------------------
| Id | Operation                      | Name                      |Rows|Bytes|Cost(%CPU)| Time|
----------------------------------------------------------------------------------------------
|   0| SELECT STATEMENT               |                           |  32|64480|686(38)|00:00:09|
|*  1| FILTER                         |                           |    |     |       |        |
|   2|TABLE ACCESS FULL               |PURCHASE_ORDERS_XMLTYPE_BIG|1161|2284K| 140(3)|00:00:02|
|*  3|COLLECTION ITERATOR PICKLER FETCH| XMLSEQUENCEFROMXMLTYPE    |    |     |       |        |
----------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter( EXISTS (SELECT 0 FROM TABLE() "KOKBF$" WHERE
              SYS_XMLCONTAINS(SYS_XQ_UPKXML2SQL(SYS_XQEXVAL(SYS_XQEXTRACT(SYS_XQCON2SEQ(VALUE(KOKBF$)),
              '/comment/text()'),1,50),50,1,0),'constitution')>0))
   3 - filter(SYS_XMLCONTAINS(SYS_XQ_UPKXML2SQL(SYS_XQEXVAL(SYS_XQEXTRACT(SYS_XQCON2SEQ(VALUE(KOKBF$)),
              '/comment/text()'),1,50),50,1,0),'constitution')>0)

Note
```

```
-----
   - dynamic sampling used for this statement
```

If you create an index on column `id`, as shown in Example 12–48, and you add a selective predicate `id` to the query, as shown in Example 12–49, then index `id` drives the execution, as shown in the execution plan. Function `ora:contains` is then executed only for the rows where `id` is less than 5.

Example 12–48 B-tree Index on ID

```
CREATE INDEX id_index ON purchase_orders_xmltype_big(id);
```

Example 12–49 ora:contains in Large Table, with Additional Predicate

```
SELECT id FROM purchase_orders_xmltype_big
  WHERE XMLExists(
          'declare namespace ora = "http://xmlns.oracle.com/xdb"; (: :)
           $d/purchaseOrder/comment[ora:contains(text(), "constitution") > 0]'
          PASSING doc AS "d")
    AND id > 5;
```

```
Execution Plan
----------------------------------------------------------------------------------------------
| Id  | Operation                   | Name                      |Rows| Bytes |Cost(%CPU)| Time|
----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |                           |  1 |  2015 |8  (13)|00:00:01|
|*  1 |  TABLE ACCESS BY INDEX ROWID|PURCHASE_ORDERS_XMLTYPE_BIG|  1 |  2015 |8  (13)|00:00:01|
|*  2 |   INDEX RANGE SCAN          |ID_INDEX                   | 10 |       |2   (0)|00:00:01|
----------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter(EXISTSNODE(SYS_MAKEXML("PURCHASE_ORDERS_XMLTYPE_BIG"."SYS_NC00003$
       ")','/purchaseOrder/items/item/comment[ora:contains(text(), "constitution") >
              0]','xmlns:ora="http://xmlns.oracle.com/xdb"')=1)
   2 - access("ID">5)

Note
-----
   - dynamic sampling used for this statement
```

XPath Rewrite and CONTEXT Indexes

`ora:contains` does not rely on a supporting index. But under some circumstances an `ora:contains` may use an existing `CONTEXT` index for better performance.

Benefits of XPath Rewrite In some circumstances, Oracle Database rewrites a SQL/XML query into an object-relational query. This is done as part of query optimization and is transparent to the user. Two of the benefits of XPath rewrite are:

- The rewritten query can directly access the underlying object-relational tables instead of processing the whole XML document.
- The rewritten query can make use of any available indexes.

XPath rewrite is a performance optimization. XPath rewrite is performed only when XML data is stored object-relationally, which in turn means that the data must be XML schema-based.

See Also:

- "Automatic Rewriting of XQuery and XPath Expressions" on page 1-20 for more on the benefits of XPath rewrite
- Chapter 8, "XPath Rewrite for Structured Storage" for a full discussion of XPath rewrite for object-relational storage

From Documents to Nodes Consider Example 12–50, a simple `ora:contains` query. To naively process the XPath expression in this query, each cell in the `doc` column must be considered, and each cell must be tested to see if it matches that XPath expression

Example 12–50 ora:contains Search for "electric"

```
SELECT id FROM purchase_orders_xmltype
  WHERE XMLExists(
          'declare namespace ora = "http://xmlns.oracle.com/xdb"; (: :)
           $d/purchaseOrder/items/item/comment
             [ora:contains(text(), "electric") > 0]'
          PASSING doc AS "d");
```

But if `doc` is XML schema-based, and the `purchaseOrder` documents are physically stored in object-relational tables, then it makes sense to go straight to column `comment` (if such a column exists) and test each cell there to see if it matches "`electric`".

This is the first XPath-rewrite step. If the first argument to `ora:contains` maps to a single relational column, then `ora:contains` executes against that column. Even if there are no indexes involved, this can significantly improve query performance.

From ora:contains to contains As noted in "From Documents to Nodes" on page 12-24, Oracle XML DB might rewrite a query so that an XPath expression passed to SQL/XML function `XMLExists` can be resolved by applying `ora:contains` to an underlying column, instead of applying the complete XPath expression to the entire XML document. This section shows how such query can make use of a `CONTEXT` index on the underlying column.

If you are running `ora:contains` against a text node or an attribute that maps to a column that has a `CONTEXT` index, why might you not use that index? One reason is that a rewritten query should give the same results as the original query. To ensure consistent results, the following conditions must both be true, in order for a `CONTEXT` index to be used.

- The `ora:contains` target (`input_text`) must be either a single text node whose parent node maps to a column or an attribute that maps to a column. The column must be a single relational column (possibly in an ordered collection table).
- As noted in "Policies for ora:contains Queries" on page 12-18, the indexing choices (for `contains`) and policy choices (for `ora:contains`) affect the semantics of queries. A simple mismatch might be that the index-based `contains` would do a *case-sensitive* search, while `ora:contains` specifies a *case-insensitive* search. To ensure that the `ora:contains` and the rewritten `contains` have the same semantics, the `ora:contains` policy must exactly match the index choices of the `CONTEXT` index.

Both the `ora:contains` policy and the `CONTEXT` index must also use the `NULL_SECTION_GROUP` section group type. The default section group for an `ora:contains` policy is `ctxsys.NULL_SECTION_GROUP`.

Third, the `CONTEXT` index is generally asynchronous. If you add a new document that contains the word "`dog`", but do not synchronize the `CONTEXT` index, then a

contains query for "dog" does not return that document. But an ora:contains query against the same data does. To ensure that the ora:contains and the rewritten contains always return the same results, build the CONTEXT index with the TRANSACTIONAL keyword in the PARAMETERS string.

See Also: *Oracle Text Reference*

Summary of Using XPath Rewrite with ora:contains A query with XMLQuery, XMLTable or XMLExists, where the XPath includes ora:contains, can be considered for XPath rewrite if:

- The XML data is XML schema-based
- The first argument to ora:contains, input_text, is either a single text node whose parent node maps to a column, or an attribute that maps to a column. The column must be a single relational column (possibly in an ordered collection table).

The rewritten query uses a CONTEXT index if:

- There is a CONTEXT index on the column that the parent node (or attribute node) of input_text maps to.
- The ora:contains policy exactly matches the index choices of the CONTEXT index.
- The CONTEXT index was built with the TRANSACTIONAL keyword in the PARAMETERS string.

XPath rewrite can speed up queries significantly, especially if there is a suitable CONTEXT index.

Text Path BNF Specification

```
HasPathArg           ::=   LocationPath
                         | EqualityExpr
InPathArg            ::=   LocationPath
LocationPath         ::=   RelativeLocationPath
                         | AbsoluteLocationPath
AbsoluteLocationPath ::=   ("/" RelativeLocationPath)
                         | ("//" RelativeLocationPath)
RelativeLocationPath ::=   Step
                         | (RelativeLocationPath "/" Step)
                         | (RelativeLocationPath "//" Step)
Step                 ::=   ("@" NCName)
                         | NCName
                         | (NCName Predicate)
                         | Dot
                         | "*"
Predicate            ::=   ("[" OrExp "]")
                         | ("[" Digit+ "]")
OrExpr               ::=   AndExpr
                         | (OrExpr "or" AndExpr)
AndExpr              ::=   BooleanExpr
                         | (AndExpr "and" BooleanExpr)
BooleanExpr          ::=   RelativeLocationPath
                         | EqualityExpr
                         | ("(" OrExpr ")")
                         | ("not" "(" OrExpr ")")
EqualityExpr         ::=   (RelativeLocationPath "=" Literal)
                         | (Literal "=" RelativeLocationPath)
```

```
                          |  (RelativeLocationPath "=" Literal)
                          |  (Literal "!=" RelativeLocationPath)
                          |  (RelativeLocationPath "=" Literal)
                          |  (Literal "!=" RelativeLocationPath)
Literal              ::=     (DoubleQuote [~"]* DoubleQuote)
                          |  (SingleQuote [~']* SingleQuote)
NCName               ::=     (Letter |  Underscore) NCNameChar*
NCNameChar           ::=    Letter
                          |  Digit
                          |  Dot
                          |  Dash
                          |  Underscore
Letter               ::=     ([a-z] | [A-Z])
Digit                ::=     [0-9]
Dot                  ::=     "."
Dash                 ::=     "-"
Underscore           ::=     "_"
```

Support for Full-Text XML Examples

This section contains these topics:

- Purchase-Order XML Document, po001.xml
- CREATE TABLE Statements
- Purchase-Order XML Schema for Full-Text Search Examples

Purchase-Order XML Document, po001.xml

Example 12–51 Purchase Order XML Document, po001.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<purchaseOrder xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xsi:noNamespaceSchemaLocation="xmlschema/po.xsd"
               orderDate="1999-10-20">
  <shipTo country="US">
    <name>Alice Smith</name>
    <street>123 Maple Street</street>
    <city>Mill Valley</city>
    <state>CA</state>
    <zip>90952</zip>
  </shipTo>
  <billTo country="US">
    <name>Robert Smith</name>
    <street>8 Oak Avenue</street>
    <city>Old Town</city>
    <state>PA</state>
    <zip>95819</zip>
  </billTo>
  <comment>Hurry, my lawn is going wild!</comment>
  <items>
    <item partNum="872-AA">
      <productName>Lawnmower</productName>
      <quantity>1</quantity>
      <USPrice>148.95</USPrice>
      <comment>Confirm this is electric</comment>
    </item>
    <item partNum="926-AA">
      <productName>Baby Monitor</productName>
```

```
      <quantity>1</quantity>
      <USPrice>39.98</USPrice>
      <shipDate>1999-05-21</shipDate>
    </item>
  </items>
</purchaseOrder>
```

CREATE TABLE Statements

Example 12–52 Create Table PURCHASE_ORDERS

```
CREATE TABLE purchase_orders (id NUMBER, doc VARCHAR2(4000));

INSERT INTO purchase_orders (id, doc)
  VALUES (1,
          '<?xml version="1.0" encoding="UTF-8"?>
           <purchaseOrder xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                          xsi:noNamespaceSchemaLocation="xmlschema/po.xsd"
                          orderDate="1999-10-20">
             <shipTo country="US">
               <name>Alice Smith</name>
               <street>123 Maple Street</street>
               <city>Mill Valley</city>
               <state>CA</state>
               <zip>90952</zip>
             </shipTo>
             <billTo country="US">
               <name>Robert Smith</name>
               <street>8 Oak Avenue</street>
               <city>Old Town</city>
               <state>PA</state>
               <zip>95819</zip>
             </billTo>
             <comment>Hurry, my lawn is going wild!</comment>
             <items>
               <item partNum="872-AA">
                 <productName>Lawnmower</productName>
                 <quantity>1</quantity>
                 <USPrice>148.95</USPrice>
                 <comment>Confirm this is electric</comment>
               </item>
               <item partNum="926-AA">
                 <productName>Baby Monitor</productName>
                 <quantity>1</quantity>
                 <USPrice>39.98</USPrice>
                 <shipDate>1999-05-21</shipDate>
               </item>
             </items>
           </purchaseOrder>');
COMMIT;
```

Example 12–53 Create Table PURCHASE_ORDERS_XMLTYPE

```
CREATE TABLE purchase_orders_xmltype (id  NUMBER, doc XMLType);

INSERT INTO purchase_orders_xmltype (id, doc)
  VALUES (1,
          XMLTYPE ('<?xml version="1.0" encoding="UTF-8"?>
                     <purchaseOrder
                       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
                        xsi:noNamespaceSchemaLocation="po.xsd"
                        orderDate="1999-10-20">
                        <shipTo country="US">
                          <name>Alice Smith</name>
                          <street>123 Maple Street</street>
                          <city>Mill Valley</city>
                          <state>CA</state>
                          <zip>90952</zip>
                        </shipTo>
                        <billTo country="US">
                          <name>Robert Smith</name>
                          <street>8 Oak Avenue</street>
                          <city>Old Town</city>
                          <state>PA</state>
                          <zip>95819</zip>
                        </billTo>
                        <comment>Hurry, my lawn is going wild!</comment>
                        <items>
                          <item partNum="872-AA">
                            <productName>Lawnmower</productName>
                            <quantity>1</quantity>
                            <USPrice>148.95</USPrice>
                            <comment>Confirm this is electric</comment>
                          </item>
                          <item partNum="926-AA">
                            <productName>Baby Monitor</productName>
                            <quantity>1</quantity>
                            <USPrice>39.98</USPrice>
                            <shipDate>1999-05-21</shipDate>
                          </item>
                        </items>
                    </purchaseOrder>'));
COMMIT;
```

Example 12–54 Create Table PURCHASE_ORDERS_XMLTYPE_TABLE

```
CREATE TABLE purchase_orders_xmltype_table OF XMLType;

INSERT INTO purchase_orders_xmltype_table
  VALUES (
    XMLType ('<?xml version="1.0" encoding="UTF-8"?>
              <purchaseOrder
                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                xsi:noNamespaceSchemaLocation="xmlschema/po.xsd"
                orderDate="1999-10-20">
                <shipTo country="US">
                  <name>Alice Smith</name>
                  <street>123 Maple Street</street>
                  <city>Mill Valley</city>
                  <state>CA</state>
                  <zip>90952</zip>
                </shipTo>
                <billTo country="US">
                  <name>Robert Smith</name>
                  <street>8 Oak Avenue</street>
                  <city>Old Town</city>
                  <state>PA</state>
                  <zip>95819</zip>
                </billTo>
                <comment>Hurry, my lawn is going wild!</comment>
```

```
                <items>
                  <item partNum="872-AA">
                    <productName>Lawnmower</productName>
                    <quantity>1</quantity>
                    <USPrice>148.95</USPrice>
                    <comment>Confirm this is electric</comment>
                  </item>
                  <item partNum="926-AA">
                    <productName>Baby Monitor</productName>
                    <quantity>1</quantity>
                    <USPrice>39.98</USPrice>
                    <shipDate>1999-05-21</shipDate>
                  </item>
                </items>
              </purchaseOrder>'));
COMMIT;
```

Purchase-Order XML Schema for Full-Text Search Examples

Example 12–55 Purchase-Order XML Schema for Full-Text Search Examples

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:annotation>
    <xsd:documentation xml:lang="en">
      Purchase order schema for Example.com.
      Copyright 2000 Example.com. All rights reserved.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:element name="purchaseOrder" type="PurchaseOrderType"/>
  <xsd:element name="comment" type="xsd:string"/>
  <xsd:complexType name="PurchaseOrderType">
    <xsd:sequence>
      <xsd:element name="shipTo" type="USAddress"/>
      <xsd:element name="billTo" type="USAddress"/>
      <xsd:element ref="comment" minOccurs="0"/>
      <xsd:element name="items" type="Items"/>
    </xsd:sequence>
    <xsd:attribute name="orderDate" type="xsd:date"/>
  </xsd:complexType>
  <xsd:complexType name="USAddress">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element name="street" type="xsd:string"/>
      <xsd:element name="city" type="xsd:string"/>
      <xsd:element name="state" type="xsd:string"/>
      <xsd:element name="zip" type="xsd:decimal"/>
    </xsd:sequence>
    <xsd:attribute name="country" type="xsd:NMTOKEN" fixed="US"/>
  </xsd:complexType>
  <xsd:complexType name="Items">
    <xsd:sequence>
      <xsd:element name="item" minOccurs="0" maxOccurs="unbounded">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="productName" type="xsd:string"/>
            <xsd:element name="quantity">
              <xsd:simpleType>
                <xsd:restriction base="xsd:positiveInteger">
```

```
                <xsd:maxExclusive value="100"/>
              </xsd:restriction>
            </xsd:simpleType>
          </xsd:element>
          <xsd:element name="USPrice" type="xsd:decimal"/>
          <xsd:element ref="comment" minOccurs="0"/>
          <xsd:element name="shipDate" type="xsd:date" minOccurs="0"/>
        </xsd:sequence>
        <xsd:attribute name="partNum" type="SKU" use="required"/>
      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>
<!-- Stock Keeping Unit, a code for identifying products -->
<xsd:simpleType name="SKU">
  <xsd:restriction base="xsd:string">
    <xsd:pattern value="\d{3}-[A-Z]{2}"/>
  </xsd:restriction>
</xsd:simpleType>
</xsd:schema>
```

Part III

Using XMLType APIs

Part III of this manual introduces you to ways you can use Oracle XML DB XMLType PL/SQL, Java, C APIs, and Oracle Data Provider for .NET (ODP.NET) to access and manipulate XML data. It contains the following chapters:

- Chapter 13, "PL/SQL APIs for XMLType"
- Chapter 14, "PL/SQL Package DBMS_XMLSTORE"
- Chapter 15, "Java DOM API for XMLType"
- Chapter 16, "Using the C API for XML"
- Chapter 17, "Using Oracle Data Provider for .NET with Oracle XML DB"

13

PL/SQL APIs for XMLType

This chapter describes the use of the APIs for `XMLType` in PL/SQL.

This chapter contains these topics:

- Overview of PL/SQL APIs for XMLType
- PL/SQL DOM API for XMLType (DBMS_XMLDOM)
- PL/SQL Parser API for XMLType (DBMS_XMLPARSER)
- PL/SQL XSLT Processor for XMLType (DBMS_XSLPROCESSOR)
- PL/SQL Translation API for XMLType (DBMS_XMLTRANSLATIONS)

Overview of PL/SQL APIs for XMLType

This chapter describes the PL/SQL Application Program Interfaces (APIs) for `XMLType`. These include the following:

- PL/SQL Document Object Model (DOM) API for `XMLType` (package `DBMS_XMLDOM`): For accessing `XMLType` objects. You can access both XML schema-based and non-schema-based documents. Before database startup, you must specify the read-from and write-to directories in file `initialization.ORA`. For example:

  ```
  UTL_FILE_DIR=/mypath/insidemypath
  ```

  The read-from and write-to files must be on the server file system.

  A **DOM** is a tree-based object representation of an XML document in dynamic memory. It enables programmatic access to its elements and attributes. The DOM object and its interface is a W3C recommendation. It specifies the Document Object Model of an XML document including APIs for programmatic access. DOM views the parsed document as a tree of objects.
- PL/SQL XML Parser API for `XMLType` (package `DBMS_XMLPARSER`): For accessing the content and structure of XML documents.
- PL/SQL XSLT Processor for `XMLType` (package `DBMS_XSLPROCESSOR`): For transforming XML documents to other formats using XSLT.

API Features

The PL/SQL APIs for `XMLType` allow you to perform the following tasks:

- Create `XMLType` tables, columns, and views
- Construct `XMLType` instances from data encoded in different character sets.

- Access `XMLType` data
- Manipulate `XMLType` data

See Also:

- "Oracle XML DB Features", for an overview of the Oracle XML DB architecture and new features.
- Chapter 4, "XMLType Operations"
- *Oracle Database PL/SQL Packages and Types Reference*

Lazy Loading of XML Data (Lazy Manifestation)

Because `XMLType` provides a dynamic memory or virtual Document Object Model (DOM), it can use a memory conserving process called **lazy XML loading**, also sometimes referred to as **lazy manifestation**. This process optimizes memory usage by only loading rows of data when they are requested. It throws away previously-referenced sections of the document if memory usage grows too large. Lazy XML loading supports highly scalable applications that have many concurrent users needing to access large XML documents.

XMLType Data Type Supports XML Schema

The `XMLType` data type includes support for XML schemas. You can create an XML schema and annotate it with mappings from XML to object-relational storage. To take advantage of the PL/SQL DOM API, first create an XML schema and register it. Then, when you create `XMLType` tables and columns, you can specify that these conform to the registered XML schema.

XMLType Supports Data in Different Character Sets

`XMLType` instances can be created from data encoded in any Oracle-supported character set by using the PL/SQL `XMLType` constructor or `XMLType` method `createXML()`. The source XML data must be supplied using data type `BFILE` or `BLOB`. The encoding of the data is specified through argument `csid`. When this argument is zero (`0`), the encoding of the source data is determined from the XML prolog, as specified in Appendix F of the XML 1.0 Reference.

Caution: *AL32UTF8* is the Oracle Database character set that is appropriate for `XMLType` data. It is equivalent to the IANA registered standard UTF-8 encoding, which supports all valid XML characters.

Do not confuse Oracle Database database character set UTF8 (no hyphen) with database character set AL32UTF8 or with character *encoding* UTF-8. Database character set UTF8 has been *superseded* by AL32UTF8. Do *not* use UTF8 for XML data. Character set UTF8 supports only Unicode version 3.1 and earlier. It does not support all valid XML characters. AL32UTF8 has no such limitation.

Using database character set UTF8 for XML data could potentially *stop a system or affect security negatively*. If a character that is not supported by the database character set appears in an input-document element name, a replacement character (usually "?") is substituted for it. This terminates parsing and raises an exception. It could cause an irrecoverable error.

PL/SQL DOM API for XMLType (DBMS_XMLDOM)

This section describes the PL/SQL DOM API for `XMLType`, `DBMS_XMLDOM`.

> **See Also:** *Oracle Database PL/SQL Packages and Types Reference* for descriptions of the individual `DBMS_XMLDOM` methods

Overview of the W3C Document Object Model (DOM) Recommendation

Skip this section if you are familiar with the generic DOM specifications recommended by the World Wide Web Consortium (W3C).

The Document Object Model (DOM) recommended by the W3C is a universal API for accessing the structure of XML documents. It was originally developed to formalize Dynamic HTML, which is used for animation, interaction, and dynamic updating of Web pages. DOM provides a language-neutral and platform-neutral object model for Web pages and XML documents. DOM describes language-independent and platform-independent interfaces to access and operate on XML components and elements. It expresses the structure of an XML document in a universal, content-neutral way. Applications can be written to dynamically delete, add, and edit the content, attributes, and style of XML documents. DOM makes it possible to create applications that work properly on all browsers, servers, and platforms.

Oracle XDK Extensions to the W3C DOM Standard

Oracle XML Developer's Kit (Oracle XDK) extends the W3C DOM API in various ways. All of these extensions are supported by Oracle XML DB except those relating to client-side operations that are not applicable in the database. This type of procedural processing is available through the Simple API for XML (SAX) interface in the Oracle XDK Java and C components.

> **See Also:** *Oracle XML Developer's Kit Programmer's Guide*

Supported W3C DOM Recommendations

All Oracle XML DB APIs for accessing and manipulating XML comply with standard XML processing requirements as approved by the W3C. The PL/SQL DOM supports Levels 1 and 2 from the W3C DOM specifications.

- In Oracle9*i* release 1 (9.0.1), Oracle XDK for PL/SQL implemented DOM Level 1.0 and parts of DOM Level 2.0.
- In Oracle9*i* release 2 (9.2) and Oracle Database 10*g* release 1 (10.1), the PL/SQL API for `XMLType` implements DOM Levels 1.0 and Level 2.0 Core, and is fully integrated in the database through extensions to the `XMLType` API.

The following briefly describes each level:

- **DOM Level 1.0 –** The first formal Level of the DOM specifications, completed in October 1998. Level 1.0 defines support for XML 1.0 and HTML.
- **DOM Level 2.0 –** Completed in November 2000, Level 2.0 extends Level 1.0 with support for XML 1.0 with namespaces and adds support for Cascading Style Sheets (CSS) and events (user-interface events and tree manipulation events), and enhances tree manipulations (tree ranges and traversal mechanisms). CSS are a simple mechanism for adding style (fonts, colors, spacing, and so on) to Web documents.

Difference Between DOM and SAX

The generic APIs for XML can be classified in two main categories:

- Tree-based. DOM is the primary generic tree-based API for XML.
- Event-based. SAX (Simple API for XML) is the primary generic event-based programming interface between an XML parser and an XML application.

DOM works by creating objects. These objects have child objects and properties, and the child objects have child objects and properties, and so on. Objects are referenced either by moving down the object hierarchy or by explicitly giving an HTML element an ID attribute. For example:

```
<img src="employee_jdoe.gif" ID="0123jdoe">
```

Examples of structural manipulations are:

- Reordering elements
- Adding or deleting elements
- Adding or deleting attributes
- Renaming elements

See Also:

- http://www.w3.org/DOM/ for information about DOM
- http://www.saxproject.org/ for information about SAX

PL/SQL DOM API for XMLType (DBMS_XMLDOM): Features

Oracle XML DB extends the Oracle Database XML development platform beyond SQL support for storage and retrieval of XML data. It lets you operate on `XMLType` instances using DOM in PL/SQL, Java, and C.

The *default* action for the PL/SQL DOM API for `XMLType` (`DBMS_XMLDOM`) is to do the following:

- Produce a parse tree that can be accessed by DOM APIs.
- Validate, if a DTD is found. Otherwise, do not validate.
- Raise an application error if parsing fails.

DTD validation occurs when the object document is manifested. If lazy manifestation is employed, then the document is validated when it is used.

The PL/SQL DOM API exploits a C-based representation of XML in the server and operates on XML schema-based XML instances. The PL/SQL, Java, and C DOM APIs for `XMLType` comply with the W3C DOM Recommendations to define and implement structured storage of XML data in relational or object-relational columns and as dynamic memory instances of `XMLType`. See "Preparing XML Data to Use the PL/SQL DOM API for XMLType" on page 13-6, for a description of W3C DOM Recommendations.

XML Schema Support

The PL/SQL DOM API for `XMLType` supports XML schema. Oracle XML DB uses annotations within an XML schema as metadata to determine the structure of an XML document and the mapping of the document to a database schema.

Note: For backward compatibility and flexibility, the PL/SQL DOM supports both XML schema-based documents and non-schema-based documents.

After an XML schema is registered with Oracle XML DB, the PL/SQL DOM API for `XMLType` builds a tree representation of an associated XML document in dynamic memory as a hierarchy of node objects, each with its own specialized interfaces. Most node object types can have child node types, which in turn implement additional, more specialized interfaces. Nodes of some node types can have child nodes of various types, while nodes of other node types must be leaf nodes, which do not have child nodes.

Enhanced Performance

Oracle XML DB uses DOM to provide a standard way to translate data between XML and multiple back-end data sources. This eliminates the need to use separate XML translation techniques for the different data sources in your environment. Applications needing to exchange XML data can use a single native XML database to cache XML documents. Oracle XML DB can thus speed up application performance by acting as an intermediate cache between your Web applications and your back-end data sources, whether they are in relational databases or file systems.

See Also: Chapter 15, "Java DOM API for XMLType"

Designing End-to-End Applications using Oracle XDK and Oracle XML DB

When you build applications based on Oracle XML DB, you do not need the additional components in Oracle XDK. However, you can use Oracle XDK components with Oracle XML DB to deploy a full suite of XML-enabled applications that run end-to-end. You can use features in Oracle XDK for:

- Simple API for XML (SAX) interface processing. SAX is an XML standard interface provided by XML parsers and used by procedural and event-based applications.
- DOM interface processing, for structural and recursive object-based processing.

Oracle XDK contain the basic building blocks for creating applications that run on a client, in a browser or a plug-in. Such applications typically read, manipulate, transform and view XML documents. To provide a broad variety of deployment options, Oracle XDK is available for Java, C, and C++. Oracle XDK is fully supported and comes with a commercial redistribution license.

Oracle XDK for Java consists of these components:

- **XML Parsers –** Creates and parses XML using industry standard DOM and SAX interfaces. Supports Java, C, C++, and the Java API for XML Processing (JAXP).
- **XSL Processor –** Transforms or renders XML into other text-based formats such as HTML. Supports Java, C, and C++.
- **XML Schema Processor –** Uses XML simple and complex data types. Supports Java, C, and C++.
- **XML Class Generator, Oracle JAXB Class Generator –** Automatically generate C++ and Java classes, respectively, from DTDs and XML schemas, to send XML data from Web forms or applications. Class generators accept an input file and create a set of output classes that have corresponding functionality. For the XML

Class Generator, the input file is a DTD, and the output is a series of classes that can be used to create XML documents conforming with the DTD.

- **XML SQL Utility –** Generates XML documents, DTDs, and XML schemas from SQL queries. Supports Java.
- **TransX Utility –** Loads data encapsulated in XML into the database. Has additional functionality useful for installations.
- **XSQL Servlet –** Combines XML, SQL, and XSLT in the server to deliver dynamic Web content.
- **XML Pipeline Processor –** Invokes Java processes through XML control files.
- **XSLT VM and Compiler –** Provides a high-performance C-based XSLT transformation engine that uses compiled style sheets.
- **XML Java Beans –** Parses, transforms, compares, retrieves, and compresses XML documents using Java components.

See Also: *Oracle XML Developer's Kit Programmer's Guide*

Preparing XML Data to Use the PL/SQL DOM API for XMLType

To prepare data for using PL/SQL DOM APIs in Oracle XML DB:

1. Create a standard XML schema.
2. Annotate the XML schema with definitions for the SQL objects you use.
3. Register the XML schema, to generate the necessary database mappings.

You can then do any of the following:

- Use `XMLType` views to wrap existing relational or object-relational data in XML formats, making it available to your applications in XML form. See "Wrapping Existing Data into XML with XMLType Views" on page 13-7.
- Insert XML data into `XMLType` columns.
- Use Oracle XML DB PL/SQL and Java DOM APIs to manipulate XML data stored in `XMLType` columns and tables.

Defining an XML Schema Mapping to SQL Object Types

An XML schema must be registered before it can be referenced by an XML document. When you register an XML schema, elements and attributes it declares are mapped to attributes of corresponding SQL object types within the database.

After XML schema registration, XML documents that conform to the XML schema and reference it can be managed by Oracle XML DB. Tables and columns for storing the conforming documents can be created for root elements defined by the XML schema.

See Also: Chapter 7, "XML Schema Storage and Query: Basic"

An XML schema is registered by using PL/SQL package `DBMS_XMLSCHEMA` and by specifying the schema document and its *schema-location URL*. This URL is a name that uniquely identifies the registered schema within the databas. It need not correspond to any real location—in particular, it need not indicate where the schema document is located.

The *target namespace* of the schema is another URL used in the XML schema. It specifies a namespace for the XML-schema elements and types. An XML document

should specify both the namespace of the root element and the schema-location URL identifying the schema that defines this element.

When documents are inserted into Oracle XML DB using path-based protocols such as HTTP(S) and FTP, the XML schema to which the document conforms is *registered implicitly*, provided its name and location are specified and it has not yet been registered.

> **See Also:** *Oracle Database PL/SQL Packages and Types Reference* descriptions of the individual `DBMS_XMLSCHEMA` methods

DOM Fidelity for XML Schema Mapping

Elements and attributes declared within the XML schema get mapped to separate attributes of the corresponding SQL object type. Other information encoded in an XML document, such as comments, processing instructions, namespace declarations and prefix definitions, and whitespace, is not represented directly.

To store this additional information, binary attribute `SYS_XDBPD$` is present in all generated SQL object types. This database attribute stores all information in the original XML document that is not stored using the other database attributes. Retaining this accessory information ensures *DOM fidelity* for XML documents stored in Oracle XML DB: an XML document retrieved from the database is identical to the original document that was stored.

> **Note:** In this book, the `SYS_XDBPD$` attribute has been omitted from most examples, for simplicity. However, the attribute is always present in SQL object types generated by schema registration.

Wrapping Existing Data into XML with XMLType Views

To make existing relational and object-relational data available to your XML applications, you can create `XMLType` views, wrapping the data in an XML format. You can then access this XML data using the PL/SQL DOM API.

After you register an XML schema containing annotations that represent the mapping between XML types and SQL object types, you can create an `XMLType` view that conforms to the XML schema.

> **See Also:** Chapter 19, "XMLType Views"

DBMS_XMLDOM Methods Supported

All `DBMS_XMLDOM` methods are supported by Oracle XML DB, with the *exception* of the following:

- `writeExternalDTDToFile()`
- `writeExternalDTDToBuffer()`
- `writeExternalDTDToClob()`

> **See Also:** *Oracle Database PL/SQL Packages and Types Reference* for descriptions of the individual `DBMS_XMLDOM` methods

PL/SQL DOM API for XMLType: Node Types

In the DOM specification, the term "**document**" is used to describe a container for many different kinds of information or data, which the DOM objectifies. The DOM specifies the way elements within an XML document container are used to create an object-based tree structure and to define and expose interfaces to manage and use the objects stored in XML documents. Additionally, the DOM supports storage of documents in diverse systems.

When a request such as `getNodeType(myNode)` is given, it returns `myNodeType`, which is the node type supported by the parent node. These constants represent the different types that a node can adopt:

- `ELEMENT_NODE`
- `ATTRIBUTE_NODE`
- `TEXT_NODE`
- `CDATA_SECTION_NODE`
- `ENTITY_REFERENCE_NODE`
- `ENTITY_NODE`
- `PROCESSING_INSTRUCTION_NODE`
- `COMMENT_NODE`
- `DOCUMENT_NODE`
- `DOCUMENT_TYPE_NODE`
- `DOCUMENT_FRAGMENT_NODE`
- `NOTATION_NODE`

Table 13–1 shows the node types for XML and HTML and the allowed corresponding children node types.

Table 13–1 XML and HTML DOM Node Types and Their Child Node Types

Node Type	Children Node Types
`Document`	`Element` (maximum of one), `ProcessingInstruction`, `Comment`, `DocumentType` (maximum of one)
`DocumentFragment`	`Element`, `ProcessingInstruction`, `Comment`, `Text`, `CDATASection`, `EntityReference`
`DocumentType`	No children
`EntityReference`	`Element`, `ProcessingInstruction`, `Comment`, `Text`, `CDATASection`, `EntityReference`
`Element`	`Element`, `Text`, `Comment`, `ProcessingInstruction`, `CDATASection`, `EntityReference`
`Attr`	`Text`, `EntityReference`
`ProcessingInstruction`	No children
`Comment`	No children
`Text`	No children

Table 13–1 (Cont.) XML and HTML DOM Node Types and Their Child Node Types

Node Type	Children Node Types
CDATASection	No children
Entity	Element, ProcessingInstruction, Comment, Text, CDATASection, EntityReference
Notation	No children

Oracle XML DB DOM API for XMLType also specifies these interfaces:

- **A NodeList interface** to handle ordered lists of Nodes, for example:
  - The children of a Node
  - Elements returned by method getElementsByTagName() of the element interface
- **A NamedNodeMap interface** to handle unordered sets of nodes, referenced by their name attribute, such as the attributes of an element.

Working with XML Schema-Based Data

Oracle Database has several extensions for character-set conversion and input and output to and from a file system. PL/SQL API for XMLType is optimized to operate on XML schema-based XML instances. Function newDOMDocument constructs a DOM document handle, given an XMLType value.

A typical usage scenario would be for a PL/SQL application to:

1. Fetch or construct an XMLType instance
2. Construct a DOMDocument node over the XMLType instance
3. Use the DOM API to access and manipulate the XML data

> **Note:** For DOMDocument, node types represent handles to XML fragments but do not represent the data itself.
>
> For example, if you copy a node value, DOMDocument clones the handle to the same underlying data. Any data modified by one of the handles is visible when accessed by the other handle. The XMLType value from which the DOMDocument handle is constructed is the data, and reflects the results of all DOM operations on it.

DOM NodeList and NamedNodeMap Objects

Changes to the underlying document structure are reflected in all relevant NodeList and NamedNodeMap objects.

For example, if a DOM user gets a NodeList object containing the children of an element, and then subsequently adds more children to that element (or removes children, or modifies them), then those changes are automatically propagated in the NodeList, without additional action from the user. Likewise, changes to a node in the tree are propagated throughout all references to that node in NodeList and NamedNodeMap objects.

The interfaces: Text, Comment, and CDATASection, all inherit from the CharacterData interface.

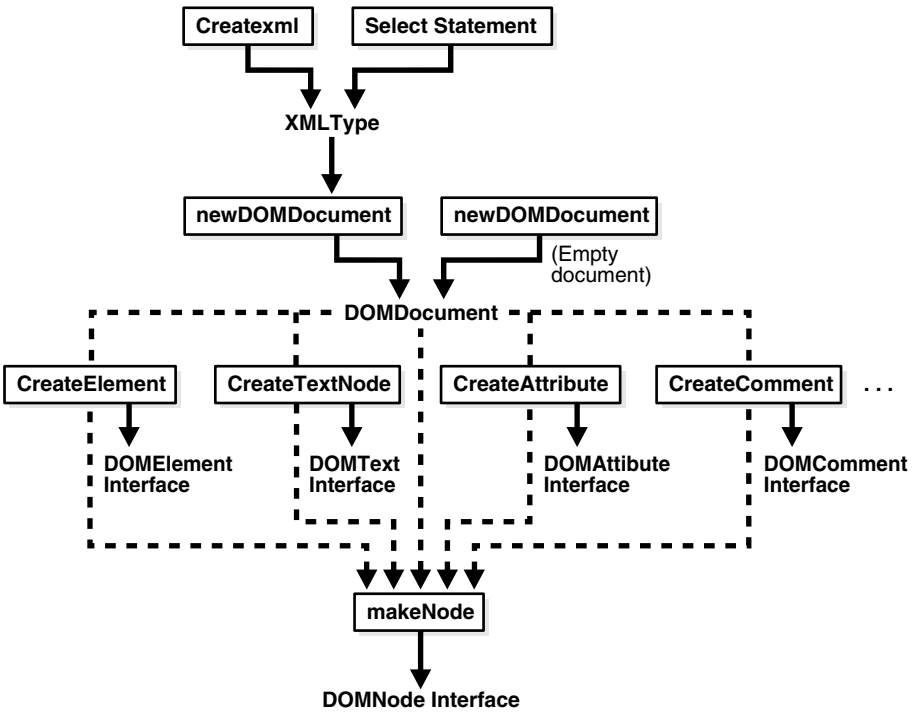
Using the PL/SQL DOM API for XMLType (DBMS_XMLDOM)

Figure 13–1 illustrates the use of PL/SQL DOM API for `XMLType` (`DBMS_XMLDOM`).

You can create a DOM document (`DOMDocument`) from an existing `XMLType` or as an empty document.

1. The `newDOMDocument` procedure processes the `XMLType` instance or empty document. This creates a `DOMDocument` instance.
2. You can use DOM API PL/SQL methods such as `createElement()`, `createText()`, `createAttribute()`, and `createComment()` to traverse and extend the DOM tree.
3. The results of PL/SQL methods such as `DOMElement()` and `DOMText()` can also be passed to PL/SQL function `makeNode` to obtain the `DOMNode` interface.

Figure 13–1 Using the PL/SQL DOM API for XMLType



PL/SQL DOM API for XMLType – Examples

This section presents examples of using the PL/SQL DOM API for `XMLType`.

Remember to call procedure `freeDocument` for *each* `DOMDocument` instance, when you are through with the instance. This procedure frees the document and all of its nodes. You can still access `XMLType` instances on which `DOMDocument` instances were built, even after the `DOMDocument` instances have been freed.

Example 13–1 creates a hierarchical, representation of an XML document in dynamic memory: a DOM document.

Example 13–1 Creating and Manipulating a DOM Document

```
CREATE TABLE person OF XMLType;

DECLARE
```

```
  var       XMLType;
  doc       DBMS_XMLDOM.DOMDocument;
  ndoc      DBMS_XMLDOM.DOMNode;
  docelem   DBMS_XMLDOM.DOMElement;
  node      DBMS_XMLDOM.DOMNode;
  childnode DBMS_XMLDOM.DOMNode;
  nodelist  DBMS_XMLDOM.DOMNodelist;
  buf       VARCHAR2(2000);
BEGIN
  var := XMLType('<PERSON><NAME>ramesh</NAME></PERSON>');

  -- Create DOMDocument handle
  doc     := DBMS_XMLDOM.newDOMDocument(var);
  ndoc    := DBMS_XMLDOM.makeNode(doc);

  DBMS_XMLDOM.writeToBuffer(ndoc, buf);
  DBMS_OUTPUT.put_line('Before:'||buf);

  docelem := DBMS_XMLDOM.getDocumentElement(doc);

  -- Access element
  nodelist := DBMS_XMLDOM.getElementsByTagName(docelem, 'NAME');
  node := DBMS_XMLDOM.item(nodelist, 0);
  childnode := DBMS_XMLDOM.getFirstChild(node);

  -- Manipulate element
  DBMS_XMLDOM.setNodeValue(childnode, 'raj');
  DBMS_XMLDOM.writeToBuffer(ndoc, buf);
  DBMS_OUTPUT.put_line('After:'||buf);
  DBMS_XMLDOM.freeDocument(doc);
  INSERT INTO person VALUES (var);
END;
/
```

This produces the following output:

```
Before:<PERSON>
  <NAME>ramesh</NAME>
</PERSON>

After:<PERSON>
  <NAME>raj</NAME>
</PERSON>
```

This query confirms that the data has changed:

```
SELECT * FROM person;
SYS_NC_ROWINFO$
---------------
<PERSON>
  <NAME>raj</NAME>
</PERSON>

1 row selected.
```

Example 13–1 uses a *handle* to the DOM document to manipulate it: print it, change part of it, and print it again after the change. Manipulating the DOM document by its handle also indirectly affects the XML data represented by the document, so that querying that data after the change shows the changed result.

The DOM document is created from an XMLType variable using PL/SQL function newDOMDocument. The handle to this document is created using function makeNode. The document is written to a VARCHAR2 buffer using function writeToBuffer, and the buffer is printed using DBMS_OUTPUT.put_line.

After manipulating the document using various DBMS_XMLDOM procedures, the (changed) data in the XMLType variable is inserted into a table and queried, showing the change. It is only when the data is inserted into a database table that it becomes persistent. Until then, it exists in memory only. This persistence is demonstrated by the fact that the database query is made after the document (DOMDocument instance) has been freed from dynamic memory.

Example 13–2 creates an empty DOM document, and then adds an element node (<ELEM>) to the document. DBMS_XMLDOM API node procedures are used to obtain the name (<ELEM>), value (NULL), and type (1 = element node) of the element node.

Example 13–2 Creating an Element Node and Obtaining Information About It

```
DECLARE
  doc   DBMS_XMLDOM.DOMDocument;
  elem  DBMS_XMLDOM.DOMElement;
  nelem DBMS_XMLDOM.DOMNode;
BEGIN
  doc := DBMS_XMLDOM.newDOMDocument;
  elem := DBMS_XMLDOM.createElement(doc, 'ELEM');
  nelem := DBMS_XMLDOM.makeNode(elem);
  DBMS_OUTPUT.put_line('Node name = ' || DBMS_XMLDOM.getNodeName(nelem));
  DBMS_OUTPUT.put_line('Node value = '|| DBMS_XMLDOM.getNodeValue(nelem));
  DBMS_OUTPUT.put_line('Node type = ' || DBMS_XMLDOM.getNodeType(nelem));
  DBMS_XMLDOM.freeDocument(doc);
END;
/
```

This produces the following output:

```
Node name = ELEM
Node value =
Node type = 1
```

Large Node Handling using DBMS_XMLDOM

Prior to Oracle Database 11g Release 1 (11.1), each text node or attribute value processed by Oracle XML DB was limited in size to 64 K bytes. Starting with release 11.1, this restriction no longer applies.

To overcome this size limitation and allow nodes to contain graphics files, PDF files, and multibyte character encodings, the following abstract streams are available. These abstract PL/SQL streams are analogous to the corresponding Java streams. Each input stream has an associated writer, or data producer, and each output stream has an associated reader, or data consumer.

1. **Binary Input Stream:** This provides the data consumer with read-only access to source data, as a sequential (non-array) linear space of bytes. The consumer has iterative read access to underlying source data (whatever representation) in binary format, that is, read access to source data in unconverted, "raw" format. The consumer sees a sequence of bytes as they exist in the node. There is no specification of the format or representation of the source data. In particular, there is no associated character set.

2. **Binary Output Stream:** This provides the data producer with write-only access to target data as a sequential (non-array) linear space of bytes. The producer has iterative write access to target data in binary format, that is, write access to target data in pure binary format with no data semantics at all. The producer passes a sequence of bytes and the target data is replaced by these bytes. No data conversion occurs.
3. **Character Input Stream:** This provides the data consumer iterative read-only access to source data as a sequential (non-array) linear space of characters, independent of the representation and format of the source data. Conversion of the source data may or may not occur.
4. **Character Output Stream:** This provides the data producer with iterative write-only access to target data as a sequential (non-array) linear space of characters. The producer passes a sequence of characters and the target data is replaced by this sequence of characters. Conversion of the passed data may or may not occur.

Each of the input streams has the following abstract methods: open, read, and close. Each of the output streams has the following abstract methods: open, write, flush, and close. For output streams, you must close the stream before any nodes are physically written.

There are four general node access models, for reading and writing. Each access model has both binary and character versions. Binary and character stream methods defined on data type `DOMNode` realize these access models. Each access model is described in a separate section, with an explanation of the PL/SQL functions and procedures in package `DBMS_XMLDOM` that operate on large nodes.

- Get-Push Model
- Get-Pull Model
- Set-Pull Model
- Set-Push Model

For all except the get-push and set-pull access models (whether binary or character), Oracle supplies a concrete stream that you can use (implicitly). For get-push and set-pull, you must define a subtype of the abstract stream type that Oracle provides, and you must implement its access methods (open, close, and so on). For get-push and set-pull, you then instantiate your stream type and supply your stream as an argument to the access method. So, for example, you would use `my_node.getNodeValueAsCharacterStream(my-stream)` for get-push, but just `my_node.getNodeValueAsCharacterStream()` for get-pull. The latter requires no explicit stream argument, because the concrete stream supplied by Oracle is used.

Note: When you access a character-data stream, the access method you use determines the apparent character set of the nodes accessed. If you use Java to access the stream, then the character set seen by your Java program is UCS2 (or an application-specified character set). If you use PL/SQL to access the stream, then the character set seen by your PL/SQL program is the database-session character set (or an application-specified character set). In all cases, however, the XML data is stored in the database in the database character set.

In the following descriptions, C1 is the character set of the node as stored in the database, and C2 is the character set of the node as seen by your program.

See Also:

- "Handling Large Nodes using Java" on page 15-16 for information on using Java with large nodes
- *Oracle Database PL/SQL Packages and Types Reference*
- *Oracle Database XML Java API Reference* for information about Java functions for handling large nodes
- *Oracle Database XML C API Reference* for information about C functions for handling large nodes

Get-Push Model

To read a node value in this model, the application creates a binary output stream or character output stream and passes this to Oracle XML DB. In this case, the source data is the node value. Oracle XML DB populates the output stream by pushing node data into the stream. If the stream is a character output stream, then the character set, C2, is the session character set, and node data is converted, if necessary, from C1 to C2. Additionally, the data type of the node may be any supported by Oracle XML DB and, if the node data type is not character data then the node data is first converted to character data in C2. If a binary output stream, the data type of the node must be `RAW` or `BLOB`.

The procedures of the `DBMS_XMLDOM` package to be used for this case are:

```
PROCEDURE getNodeValueAsBinaryStream (n     IN DBMS_XMLDOM.domnode,
                                      value IN SYS.utl_BinaryOutputStream);
```

The application passes an implementation of `SYS.utl_BinaryOutputStream` into which Oracle XML DB writes the contents of the node. The data type of the node must be `RAW` or `CLOB` or else an exception is raised.

```
PROCEDURE getNodeValueAsCharacterStream (n     IN DBMS_XMLDOM.domnode,
                                         value IN SYS.utl_CharacterOutputStream);
```

The node data is converted, as necessary, to the session character set and then "pushed" into the `SYS.utl_CharacterOutputStream`.

The following example fragments illustrate reading the node value as binary data and driving the write methods in a user-defined subtype of `SYS.utl_BinaryOutPutStream`, which is called `MyBinaryOutputStream`:

Example 13–3 Creating a User-Defined Subtype of SYS.util_BinaryOutputStream()

```
CREATE TYPE MyBinaryOutputStream UNDER SYS.utl_BinaryOutputStream (
    CONSTRUCTOR FUNCTION MyBinaryOutputStream ()
    RETURN SELF AS RESULT,
    MEMBER FUNCTION  write (bytes IN RAW) RETURN INTEGER,
    MEMBER PROCEDURE write (bytes IN RAW, offset IN INTEGER, length IN OUT
           INTEGER),
    MEMBER FUNCTION flush () RETURN BOOLEAN,
    MEMBER FUNCTION close () RETURN BOOLEAN);
);

-- Put code here that implements these methods
...
```

Example 13–4 Retrieving Node Value with a User-Defined Stream

```
DECLARE
  ostream     MyBinaryOutputStream = MyBinaryOutputStream ();
  node        DBMS_XMLDOM.domnode;
  ...
BEGIN
  ...
  -- This drives the write methods in MyBinaryOutputStream,
  -- flushes the data, and closes the stream after the value has been
  -- completely written.
  DBMS_XMLDOM.getNodeValueAsBinaryStream (node, ostream);
  ...
END;
```

Get-Pull Model

To read the value of a node in this model, Oracle XML DB creates a binary input stream or character input stream and returns this to the caller. The character set, C2, of the character input stream is the current session character set. Oracle XML DB populates the input stream as the caller pulls the node data from the stream so Oracle XML DB is again the producer of the data. If the stream is a character input stream, then the node data type may be any supported by Oracle XML DB and node data, if character, is converted, if necessary, from C1 to C2. If the node data is non-character, it is converted to character in C2. If a binary input stream, the data type of the node must be `RAW` or `BLOB`.

The functions of the `DBMS_XMLDOM` package to be used for this case are `getNodeValueAsBinaryStream` and `getNodeValueAsCharacterStream`.

```
FUNCTION getNodeValueAsBinaryStream(n IN DBMS_XMLDOM.domnode)
         RETURN SYS.utl_BinaryInputStream;
```

This function returns an instance of the new PL/SQL `SYS.utl_BinaryInputStream` that can be read using defined methods as described in the section "Set-Pull Model" on page 13-16. The node data type must be `RAW` or `BLOB` or else an exception is raised.

```
FUNCTION getNodeValueAsCharacterStream (n IN DBMS_XMLDOM.domnode)
         RETURN SYS.utl_CharacterInputStream;
```

This function returns an instance of the new PL/SQL `SYS.utl_CharacterInputStream` that can be read using defined methods. If the node data is

character it is converted to the current session character set. If the node data is not character data, it is first converted to character data.

Example 13–5 illustrates reading a node value as binary data in 50-byte increments:

Example 13–5 Get-Pull of Binary Data

```
DECLARE
  istream      SYS.utl_BinaryInputStream;
  node         DBMS_XMLDOM.domnode;
  buffer       raw(50);
  numBytes     pls_integer;
  ...
BEGIN
  ...
  istream := DBMS_XMLDOM.getNodeValueAsBinaryStream (node);
  -- Read stream in 50-byte chunks
  LOOP
    numBytes := 50;
    istream.read ( buffer, numBytes);
    if numBytes <= 0 then
       exit;
    end if;
-- Process next 50 bytes of node value in buffer
END LOOP
...
END;
```

Example 13–6 illustrates reading a node value as character data in 50-character increments:

Example 13–6 Get-Pull of Character Data

```
DECLARE
  istream      SYS.utl_CharacterInputStream;
  node         DBMS_XMLDOM.domnode;
  buffer       varchar2(50);
  numChars     pls_integer;
  ...
BEGIN
  ...
  istream := DBMS_XMLDOM.getNodeValueAsCharacterStream (node);
-- Read stream in 50-character chunks
LOOP
   numChars := 50;
   istream.read ( buffer, numChars);
   IF numChars <= 0 then
      exit;
   END IF;
-- Process next 50 characters of node value in buffer
END LOOP
...
END;
```

Set-Pull Model

To write a node value in this mode, the application creates a binary input stream or character input stream and passes this to Oracle XML DB. The character set of the character input stream, C2, is the session character set. Oracle XML DB pulls the data from the input stream and populates the node. If the stream is a character input

stream, then the data type of the node may be any supported by Oracle XML DB. If the data type of the node is not character, the stream data is first converted to the node data type. If the node data type is character, then no conversion occurs, so the node data remains in character set C2. If the stream is a `binary input stream`, then the data type of the node must be `RAW` or `BLOB` and no conversion occurs.

The procedures of the `DBMS_XMLDOM` package to be used for this case are `setNodeValueAsBinaryStream` and `setNodeValueAsCharacterStream`.

```
PROCEDURE setNodeValueAsBinaryStream(n IN DBMS_XMLDOM.domnode,
          value IN SYS.utl_BinaryInputStream);
```

The application passes in an implementation of `SYS.utl_BinaryInputStream` from which Oracle XML DB reads data to populate the node. The data type of the node must be `RAW` or `BLOB` or else an exception is raised.

```
PROCEDURE setNodeValueAsCharacterStream (n IN DBMS_XMLDOM.domnode,
          value IN SYS.utl_CharacterInputStream);
```

The application passes in an implementation of `SYS.utl_CharacterInputStream` from which Oracle XML DB reads to populate the node. The data type of the node may be any valid type supported by Oracle XML DB. If it is a non-character data type, the character data read from the stream is converted to the data type of the node. If the data type of the node is either character or `CLOB`, then no conversion occurs and the character set of the node becomes the character set of the PL/SQL session.

Example 13–7 illustrates setting the node value to binary data produced by the read methods defined in a user-defined subtype of `SYS.utl_BinaryInputStream`, which is called `MyBinaryInputStream`:

Example 13–7 Set-Pull of Binary Data

```
CREATE TYPE MyBinaryInputStream UNDER SYS.utl_BinaryInputStream (
    CONSTRUCTOR FUNCTION MyBinaryInputStream ()
    RETURN SELF AS RESULT,
  MEMBER FUNCTION read () RETURN RAW,
  MEMBER PROCEDURE read (bytes IN OUT RAW, numbytes IN OUT INTEGER),
  MEMBER PROCEDURE read (bytes IN OUT RAW,
                         offset IN INTEGER,
                         length IN OUT INTEGER),
  MEMBER FUNCTION close () RETURN BOOLEAN);
```

You can use an object of type `MyBinaryInputStream` to set the value of a node as follows:

```
DECLARE
  istream     MyBinaryInputStream = MyBinaryInputStream ();
  node        DBMS_XMLDOM.domnode;
  ...
BEGIN
  ...
  -- This drives the read methods in MyBinaryInputStream
  DBMS_XMLDOM.setNodeValueAsBinaryStream (node, istream);
  ...
END;
```

Set-Push Model

To write a new node value in this mode, Oracle XML DB creates a binary output stream or character output stream and returns this to the caller. The character set of the

character output stream, C2, is the current session character set. The caller pushes data into the output stream and Oracle XML DB then writes this to the Oracle XML DB Node. If the stream is a character output stream, then the data type of the node may be any type supported by Oracle XML DB. In this case, the character data is converted to the node data type. If the node data type is character, then the character set, C1, is changed to C2. No data conversion occurs. If the stream is a binary input stream, and the data type of the node must be `RAW` or `BLOB`. In this case, the stream is read without data conversion.

The procedures of the `DBMS_XMLDOM` package to be used for this case are `setNodeValueAsBinaryStream` and `setNodeValueAsCharacterStream`.

```
FUNCTION setNodeValueAsBinaryStream(n IN DBMS_XMLDOM.domnode)
         RETURN SYS.utl_BinaryOutputStream;
```

This function returns an instance of `SYS.utl_BinaryOutputStream` into which the caller can write the node value. The data type of the node must be `RAW` or `BLOB` or else an exception is raised.

```
FUNCTION setNodeValueAsCharacterStream (n IN DBMS_XMLDOM.domnode)
         RETURN SYS.utl_CharacterOutputStream;
```

This function returns an instance of the PL/SQL `SYS.utl_CharacterOutputStream` type into which the caller can write the node value. The data type of the node can be any valid Oracle XML DB data type. If the type is not character or CLOB, the character data written to the stream is converted to the node data type. If the data type of the node is character or CLOB, then the character data written to the stream is converted from PL/SQL session character set to the character set of the node

Example 13–8 illustrates setting the value of a node to binary data by writing 50-byte segments into the `SYS.utl_BinaryOutputStream`:

Example 13–8 Set-Push of Binary Data

```
DECLARE
  ostream      SYS.utl_BinaryOutputStream;
  node         DBMS_XMLDOM.domnode;
  buffer       raw(500);
  segment      raw(50);
  numBytes     pls_integer;
  offset       pls_integer;
  ...
BEGIN
  ...
  ostream := DBMS_XMLDOM.setNodeValueAsBinaryStream (node);
  offset := 0;
  length := 500;
  -- Write to stream in 50-byte chunks
  LOOP
    numBytes := 50;
    -- Get next 50 bytes of buffer
    ostream.write ( segment, offset, numBytes);
    length := length - numBytes;
    IF length  <= 0 then
       exit;
    END IF;
  END LOOP
  ostream.close();
  ...
```

```
END;
```

Determining Binary Stream or Character Stream

To determine whether to use a character stream or a binary stream to access the node value use the following method which is also included as part of the DBMS_XMLDOM package:

```
FUNCTION useBinaryStream (n IN DBMS_XMLDOM.domnode) RETURN BOOLEAN;
```

This function returns TRUE if the data type of the node is RAW or BLOB, so that the node value may be read or written using either a SYS.utl_BinaryInputStream or a SYS.utl_BinaryOutputStream. If a value of FALSE is returned, the node value can be accessed only using a SYS.utl_CharacterInputStream or a SYS.utl_CharacterOutputStream.

PL/SQL Parser API for XMLType (DBMS_XMLPARSER)

XML documents are made up of storage units, called *entities*, that contain either parsed or unparsed data. Parsed data is made up of characters, some of which form character data and some of which form markup. Markup encodes a description of the document storage layout and logical structure. XML provides a mechanism for imposing constraints on the storage layout and logical structure.

A software module called an XML parser or processor reads XML documents and provides access to their content and structure. An XML parser usually does its work on behalf of another module, typically the application.

Features of the PL/SQL Parser API for XMLType

The PL/SQL Parser API for XMLType (DBMS_XMLPARSER) builds a result tree that can be accessed by PL/SQL APIs. If parsing fails, it raises an error.

Method DBMS_XMLPARSER.setErrorLog() is not supported.

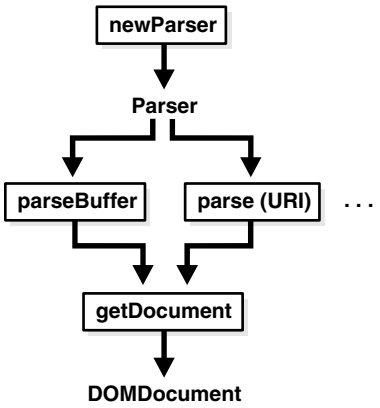
> **See Also:** *Oracle Database PL/SQL Packages and Types Reference* for descriptions of individual DBMS_XMLPARSER methods

Using the PL/SQL Parser API for XMLType (DBMS_XMLPARSER)

Figure 13–2 illustrates how to use the PL/SQL Parser for XMLType (DBMS_XMLPARSER). These are the steps:

1. Construct a parser instance using PL/SQL method newParser().
2. Parse XML documents using PL/SQL methods such as parseBuffer(), parseClob(), and parse(URI). An error is raised if the input is not a valid XML document.
3. Call PL/SQL function getDocument on the parser to obtain a DOMDocument interface.

Figure 13–2 Using the PL/SQL Parser API for XMLType



Example 13–9 parses a simple XML document. It creates an XML parser (instance of `DBMS_XMLPARSER.parser`) and uses it to parse the XML document (text) in variable `indoc`. Parsing creates a DOM document, which is retrieved from the parser using `DBMS_XMLPARSER.getDocument`. A DOM node is created that contains the entire document, and the node is printed. After freeing (destroying) the DOM document, the parser instance is freed using `DBMS_XMLPARSER.freeParser`.

Example 13–9 Parsing an XML Document

```
DECLARE
  indoc    VARCHAR2(2000);
  indomdoc DBMS_XMLDOM.DOMDocument;
  innode   DBMS_XMLDOM.DOMNode;
  myparser DBMS_XMLPARSER.parser;
  buf      VARCHAR2(2000);
BEGIN
  indoc := '<emp><name>De Selby</name></emp>';
  myParser := DBMS_XMLPARSER.newParser;
  DBMS_XMLPARSER.parseBuffer(myParser, indoc);
  indomdoc := DBMS_XMLPARSER.getDocument(myParser);
  innode := DBMS_XMLDOM.makeNode(indomdoc);
  DBMS_XMLDOM.writeToBuffer(innode, buf);
  DBMS_OUTPUT.put_line(buf);
  DBMS_XMLDOM.freeDocument(indomdoc);
  DBMS_XMLPARSER.freeParser(myParser);
END;
/
```

This produces the following output:

```
<emp><name>De Selby</name></emp>
```

PL/SQL XSLT Processor for XMLType (DBMS_XSLPROCESSOR)

The W3C XSL Recommendation describes rules for transforming a source tree into a result tree. A transformation expressed in Extensible Stylesheet Language Transformation (XSLT) language is called an **XSL style sheet**. The transformation specified is achieved by associating patterns with templates defined in the XSLT style sheet. A template is instantiated to create part of the result tree.

Enabling Transformations and Conversions with XSLT

The Oracle XML DB PL/SQL DOM API for `XMLType` also supports XSLT. This enables transformation from one XML document to another, or conversion into HTML, PDF, or other formats. XSLT is also widely used to convert XML to HTML for browser display.

The embedded XSLT processor follows Extensible Stylesheet Language (XSL) statements and traverses the DOM tree structure for XML data residing in `XMLType`. Oracle XML DB applications do not require a separate parser as did the prior release XML Parser for PL/SQL. However, applications requiring external processing can still use the XML Parser for PL/SQL first to expose the document structure.

PL/SQL package `DBMS_XSLPROCESSOR` provides a convenient and efficient way of applying a single style sheet to multiple documents. The performance of this package is better than that of Oracle SQL function `XMLtransform` and `XMLType` method `transform()`, because the style sheet is parsed only once.

Note: The XML Parser for PL/SQL in Oracle XDK parses an XML document (or a standalone DTD) so that the XML document can be processed by an application, typically running on the client. PL/SQL APIs for `XMLType` are used for applications that run on the server and are natively integrated in the database. Benefits include performance improvements and enhanced access and manipulation options.

See Also: Chapter 11, "Transforming and Validating XMLType Data"

PL/SQL XSLT Processor for XMLType: Features

PL/SQL XSLT Processor for `XMLType` (`DBMS_XSLPROCESSOR`) is the Oracle XML DB implementation of the XSL processor. This follows the W3C XSLT final recommendation (REC-xslt-19991116). It includes the required action of an XSL processor in terms of how it must read XSLT style sheets and the transformations it must achieve.

The types and methods of the PL/SQL XSLT Processor API are made available by PL/SQL package `DBMS_XSLPROCESSOR`. The methods in this package use PL/SQL data types `PROCESSOR` and `STYLESHEET`, which are specific to the XSL Processor implementation.

All `DBMS_XSLPROCESSOR` methods are supported by Oracle XML DB, with the *exception* of method `setErrorLog()`.

See Also: *Oracle Database PL/SQL Packages and Types Reference* for descriptions of the individual `DBMS_XSLPROCESSOR` methods

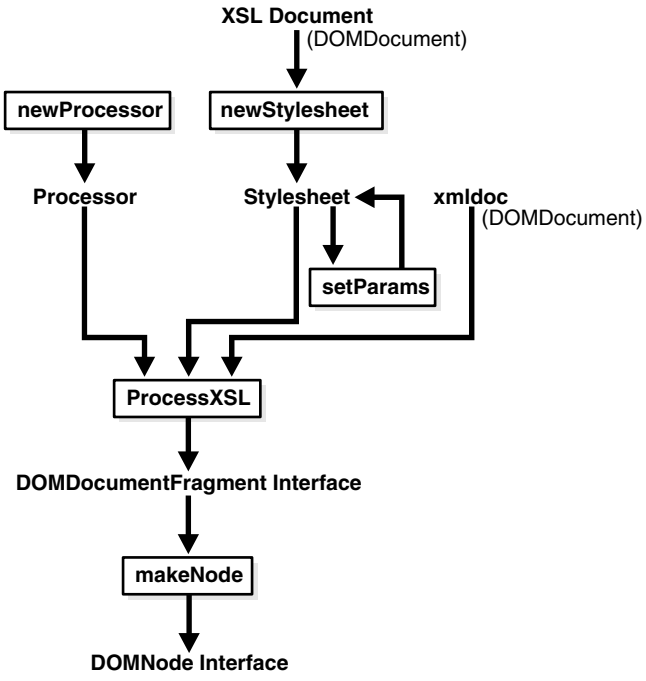
Using the PL/SQL XSLT Processor API for XMLType (DBMS_XSLPROCESSOR)

Figure 13–3 illustrates how to use XSLT Processor for `XMLType` (`DBMS_XSLPROCESSOR`). These are the steps:

1. Construct an XSLT processor using `newProcessor`.
2. Use `newStylesheet` to build a `STYLESHEET` object from a DOM document.
3. Optionally, you can set parameters for the `STYLESHEET` object using `setParams`.

4. Use `processXSL` to transform a DOM document using the processor and `STYLESHEET` object.
5. Use the PL/SQL DOM API for `XMLType` to manipulate the result of XSLT processing.

Figure 13–3 Using the PL/SQL XSLT Processor for XMLType



Example 13–10 transforms an XML document using procedure `processXSL`. It uses the same parser instance to create two different DOM documents: the XML text to transform and the XSLT style sheet. An XSL processor instance is created, which applies the style sheet to the source XML to produce a new DOM fragment. A DOM node (`outnode`) is created from this fragment, and the node content is printed. The output DOM fragment, parser, and XSLT processor instances are freed using procedures `freeDocFrag`, `freeParser`, and `freeProcessor`, respectively.

Example 13–10 Transforming an XML Document using an XSL Style Sheet

```
DECLARE
  indoc      VARCHAR2(2000);
  xsldoc     VARCHAR2(2000);
  myParser   DBMS_XMLPARSER.parser;
  indomdoc   DBMS_XMLDOM.DOMDocument;
  xsltdomdoc DBMS_XMLDOM.DOMDocument;
  xsl        DBMS_XSLPROCESSOR.stylesheet;
  outdomdocf DBMS_XMLDOM.DOMDocumentFragment;
  outnode    DBMS_XMLDOM.DOMNode;
  proc       DBMS_XSLPROCESSOR.processor;
  buf        VARCHAR2(2000);
BEGIN
  indoc := '<emp><empno>1</empno>
             <fname>robert</fname>
             <lname>smith</lname>
             <sal>1000</sal>
             <job>engineer</job>
```

```
            </emp>';
  xsldoc := '<?xml version="1.0"?>
             <xsl:stylesheet version="1.0"
                             xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
               <xsl:output encoding="utf-8"/>
               <!-- alphabetizes an xml tree -->
               <xsl:template match="*">
                 <xsl:copy>
                   <xsl:apply-templates select="*|text()">
                     <xsl:sort select="name(.)" data-type="text"
                               order="ascending"/>
                   </xsl:apply-templates>
                 </xsl:copy>
               </xsl:template>
               <xsl:template match="text()">
                 <xsl:value-of select="normalize-space(.)"/>
               </xsl:template>
             </xsl:stylesheet>';
  myParser := DBMS_XMLPARSER.newParser;
  DBMS_XMLPARSER.parseBuffer(myParser, indoc);
  indomdoc   := DBMS_XMLPARSER.getDocument(myParser);
  DBMS_XMLPARSER.parseBuffer(myParser, xsldoc);
  xsltdomdoc := DBMS_XMLPARSER.getDocument(myParser);
  xsl        := DBMS_XSLPROCESSOR.newStyleSheet(xsltdomdoc, '');
  proc       := DBMS_XSLPROCESSOR.newProcessor;
  --apply stylesheet to DOM document
  outdomdocf := DBMS_XSLPROCESSOR.processXSL(proc, xsl, indomdoc);
  outnode    := DBMS_XMLDOM.makeNode(outdomdocf);
  -- PL/SQL DOM API for XMLType can be used here
  DBMS_XMLDOM.writeToBuffer(outnode, buf);
  DBMS_OUTPUT.put_line(buf);
  DBMS_XMLDOM.freeDocument(indomdoc);
  DBMS_XMLDOM.freeDocument(xsltdomdoc);
  DBMS_XMLDOM.freeDocFrag(outdomdocf);
  DBMS_XMLPARSER.freeParser(myParser);
  DBMS_XSLPROCESSOR.freeProcessor(proc);
END;
/
```

This produces the following output:

```
<emp>
<empno>1</empno>
<fname>robert</fname>
<job>engineer</job>
<lname>smith</lname>
<sal>1000</sal>
</emp>
```

PL/SQL Translation API for XMLType (DBMS_XMLTRANSLATIONS)

When you store security objects in the Oracle XML DB Repository, you store them as `XMLType` instances. The security objects also contain some strings that must be translated, so that you can search for or display them in various languages. The translations for these strings are also stored in the Oracle XML DB Repository, along with the original strings, because they must be associated with the original document. You can retrieve and operate on these strings, depending on your language settings.

Oracle XML DB provides translation support through the `DBMS_XMLTRANSLATIONS` package, which provides an interface to perform translations so that strings can be searched or displayed in various languages.

> **See Also:** Chapter 7, "XML Schema Storage and Query: Basic" for an overview of XML translations

DBMS_XMLTRANSLATIONS Methods

PL/SQL package `DBMS_XMLTRANSLATIONS` provides the following methods:

- `updateTranslation()`: Updates the translation in a particular language at the specified `XPATH`. If the translation in that language is not present, then it is inserted.
- `setSourceLang()`: Sets the source language to a particular language at the specified `XPATH`.
- `translateXML()`: Returns the document in the specified language.
- `getBaseDocument()`: Returns the base document with all the translations.
- `extractXLiff()`: Extracts the translations in `XLIFF` format from either an `XMLTYPE` instance or a resource in Oracle XML DB Repository.
- `mergeXLiff()`: Merges the translations in `XLIFF` format into either an `XMLTYPE` or a resource in Oracle XML DB Repository.
- `disableTranslation()`: Disables translations in the current session so that query or retrieval takes place on the base document, ignoring session language values.
- `enableTranslation()`: Enables translations in the current session.

> **See Also:** *Oracle Database PL/SQL Packages and Type References* for a description of the individual `DBMS_XMLTRANSLATIONS` methods.

14

PL/SQL Package DBMS_XMLSTORE

This chapter introduces you to the PL/SQL package DBMS_XMLSTORE. This package is used to insert, update, and delete data from XML documents in object-relational tables.

This chapter contains these topics:

- Overview of PL/SQL Package DBMS_XMLSTORE
- Using Package DBMS_XMLSTORE
- Inserting with DBMS_XMLSTORE
- Updating with DBMS_XMLSTORE
- Deleting with DBMS_XMLSTORE

Overview of PL/SQL Package DBMS_XMLSTORE

PL/SQL package DBMS_XMLSTORE enables DML operations to be performed on relational tables using XML. It takes a canonical XML mapping similar to that produced by package DBMS_XMLGEN; converts it to object-relational constructs; and then inserts, updates or deletes the corresponding value from relational tables.

The functionality of package DBMS_XMLSTORE is similar to that of package DBMS_XMLSAVE, which is part of the Oracle XML SQL Utility (XSU). There are, however, some important differences:

- DBMS_XMLSTORE is written in C and compiled into the kernel, so it provides higher performance.
- DBMS_XMLSTORE uses the Simple API for XML (SAX) to parse the input XML document, so it has higher scalability and lower memory requirements. DBMS_XMLSTORE lets you input XMLType data, in addition to CLOB and VARCHAR.
- PL/SQL functions insertXML, updateXML, and deleteXML, which are also present in package DBMS_XMLSAVE, have been enhanced in package DBMS_XMLSTORE to take XMLType instances in addition to CLOB values and strings. This provides for better integration with Oracle XML DB functionality.

Using Package DBMS_XMLSTORE

To use PL/SQL package DBMS_XMLSTORE, follow these steps:

1. Create a context handle by calling function DBMS_XMLSTORE.**newContext** and supplying it with the table name to use for the DML operations. For case sensitivity, double-quote the string that is passed to the function.

By default, XML documents are expected to identify rows with the `<ROW>` tag. This is the same default used by package `DBMS_XMLGEN` when generating XML. This may be overridden by calling function `setRowTag`.

2. *For Inserts:* You can set the list of columns to insert calling procedure `DBMS_XMLSTORE.`**`setUpdateColumn`** for each column. This is highly recommended, since it improves performance. The default behavior is to insert values for all of the columns whose corresponding elements are present in the XML document.

3. *For Updates:* You must specify one or more (pseudo-) key columns using function `DBMS_XMLSTORE.`**`setKeyColumn`**. These are used to specify which *rows* are to be updated. In SQL, you would do that using a `WHERE` clause in an `UPDATE` statement, specifying a combination of columns that uniquely identify the rows to be updated. The columns that you use with `setKeyColumn` need *not* be keys of the table—as long as they uniquely specify a row, they can be used.

   For example, in table `employees`, column `employee_id` uniquely identifies rows (it is a key of the table). If the XML document that you use to update the table contains element `<EMPLOYEE_ID>2176</EMPLOYEE_ID>`, then the rows where `employee_id` equals `2176` are updated.

   The list of *update* columns can also be specified, using `DBMS_XMLSTORE.setUpdateColumn`. This is recommended, for better performance. The default behavior is to update *all* of the columns in the row(s) identified by `setKeyColumn` whose corresponding elements are present in the XML document.

4. *For Deletions:* As for updates, you specify (pseudo-) key columns to identify the row(s) to delete.

5. Provide a document to PL/SQL function `insertXML`, `updateXML`, or `deleteXML`.

6. This last step may be repeated multiple times, with several XML documents.

7. Close the context with function `DBMS_XMLSTORE.`**`closeContext`**.

Inserting with DBMS_XMLSTORE

To insert an XML document into a table or view, you supply the table or view name and the document. `DBMS_XMLSTORE` parses the document and then creates an `INSERT` statement into which it binds all the values. By default, `DBMS_XMLSTORE` inserts values into all the columns represented by elements in the XML document.

Example 14–1 uses `DBM_XMLSTORE` to insert the information for two new employees into the `employees` table. The information is provided in the form of XML data.

Example 14–1 Inserting Data with Specified Columns

```
SELECT employee_id AS EMP_ID, salary, hire_date, job_id, email, last_name
  FROM employees WHERE department_id = 30;

EMP_ID     SALARY HIRE_DATE JOB_ID     EMAIL       LAST_NAME
------ ---------- --------- ---------- ---------- ----------
   114      11000 07-DEC-94 PU_MAN     DRAPHEAL     Raphaely
   115       3100 18-MAY-95 PU_CLERK   AKHOO            Khoo
   116       2900 24-DEC-97 PU_CLERK   SBAIDA          Baida
   117       2800 24-JUL-97 PU_CLERK   STOBIAS        Tobias
   118       2600 15-NOV-98 PU_CLERK   GHIMURO        Himuro
   119       2500 10-AUG-99 PU_CLERK   KCOLMENA   Colmenares

6 rows selected.
```

```
DECLARE
  insCtx DBMS_XMLSTORE.ctxType;
  rows NUMBER;
  xmlDoc CLOB :=
    '<ROWSET>
       <ROW num="1">
         <EMPLOYEE_ID>920</EMPLOYEE_ID>
         <SALARY>1800</SALARY>
         <DEPARTMENT_ID>30</DEPARTMENT_ID>
         <HIRE_DATE>17-DEC-2002</HIRE_DATE>
         <LAST_NAME>Strauss</LAST_NAME>
         <EMAIL>JSTRAUSS</EMAIL>
         <JOB_ID>ST_CLERK</JOB_ID>
       </ROW>
       <ROW>
         <EMPLOYEE_ID>921</EMPLOYEE_ID>
         <SALARY>2000</SALARY>
         <DEPARTMENT_ID>30</DEPARTMENT_ID>
         <HIRE_DATE>31-DEC-2004</HIRE_DATE>
         <LAST_NAME>Jones</LAST_NAME>
         <EMAIL>EJONES</EMAIL>
         <JOB_ID>ST_CLERK</JOB_ID>
       </ROW>
     </ROWSET>';
BEGIN
  insCtx := DBMS_XMLSTORE.newContext('HR.EMPLOYEES'); -- Get saved context
  DBMS_XMLSTORE.clearUpdateColumnList(insCtx); -- Clear the update settings

  -- Set the columns to be updated as a list of values
  DBMS_XMLSTORE.setUpdateColumn(insCtx, 'EMPLOYEE_ID');
  DBMS_XMLSTORE.setUpdateColumn(insCtx, 'SALARY');
  DBMS_XMLSTORE.setUpdateColumn(insCtx, 'HIRE_DATE');
  DBMS_XMLSTORE.setUpdateColumn(insCtx, 'DEPARTMENT_ID');
  DBMS_XMLSTORE.setUpdateColumn(insCtx, 'JOB_ID');
  DBMS_XMLSTORE.setUpdateColumn(insCtx, 'EMAIL');
  DBMS_XMLSTORE.setUpdateColumn(insCtx, 'LAST_NAME');

  -- Insert the doc.
  rows := DBMS_XMLSTORE.insertXML(insCtx, xmlDoc);
  DBMS_OUTPUT.put_line(rows || ' rows inserted.');

  -- Close the context
  DBMS_XMLSTORE.closeContext(insCtx);
END;
/

2 rows inserted.

PL/SQL procedure successfully completed.

SELECT employee_id AS EMP_ID, salary, hire_date, job_id, email, last_name
  FROM employees WHERE department_id = 30;

EMP_ID     SALARY HIRE_DATE JOB_ID     EMAIL       LAST_NAME
------ ---------- --------- ---------- ---------- ----------
   114      11000 07-DEC-94 PU_MAN     DRAPHEAL     Raphaely
   115       3100 18-MAY-95 PU_CLERK   AKHOO            Khoo
   116       2900 24-DEC-97 PU_CLERK   SBAIDA          Baida
   117       2800 24-JUL-97 PU_CLERK   STOBIAS        Tobias
```

```
   118       2600 15-NOV-98 PU_CLERK   GHIMURO        Himuro
   119       2500 10-AUG-99 PU_CLERK   KCOLMENA   Colmenares
   920       1800 17-DEC-02 ST_CLERK   STRAUSS       Strauss
   921       2000 31-DEC-04 ST_CLERK   EJONES          Jones

8 rows selected.
```

Updating with DBMS_XMLSTORE

To update (modify) existing data using package DBMS_XMLSTORE, you must specify which rows to update. In SQL, you would do that using a WHERE clause in an UPDATE statement. With DBMS_XMLSTORE, you do it by calling procedure setKeyColumn once for *each* of the columns that are used collectively to identify the row.

You can think of this set of columns as acting like a set of key columns: *together, they specify a unique row* to be updated. However, the columns that you use (with setKeyColumn) need *not* be keys of the table—as long as they uniquely specify a row, they can be used with calls to setKeyColumn.

Example 14–2 uses DBM_XMLSTORE to update information. Assuming that the first name for employee number 188 is incorrectly recorded as Kelly, this example corrects that first name to Pat. Since column employee_id is a primary key for table employees, a single call to setKeyColumn specifying column employee_id is sufficient to identify a unique row for updating.

Example 14–2 Updating Data with Key Columns

```
SELECT employee_id, first_name FROM employees WHERE employee_id = 188;

EMPLOYEE_ID FIRST_NAME
----------- ----------
        188 Kelly

1 row selected.

DECLARE
  updCtx DBMS_XMLSTORE.ctxType;
  rows NUMBER;
  xmlDoc CLOB :=
    '<ROWSET>
       <ROW>
         <EMPLOYEE_ID>188</EMPLOYEE_ID>
         <FIRST_NAME>Pat</FIRST_NAME>
       </ROW>
     </ROWSET>';
BEGIN
   updCtx := DBMS_XMLSTORE.newContext('HR.EMPLOYEES'); -- get the context
   DBMS_XMLSTORE.clearUpdateColumnList(updCtx);        -- clear update settings

   -- Specify that column employee_id is a "key" to identify the row to update.
   DBMS_XMLSTORE.setKeyColumn(updCtx, 'EMPLOYEE_ID');
   rows := DBMS_XMLSTORE.updateXML(updCtx, xmlDoc);    -- update the table
   DBMS_XMLSTORE.closeContext(updCtx);                 -- close the context
END;
/

SELECT employee_id, first_name FROM employees WHERE employee_id = 188;

EMPLOYEE_ID FIRST_NAME
----------- ----------
```

```
        188 Pat

1 row selected.
```

The following UPDATE statement is equivalent to the use of DBM_XMLSTORE in Example 14–2:

```
UPDATE hr.employees SET first_name = 'Pat' WHERE employee_id = 188;
```

Deleting with DBMS_XMLSTORE

Deletions are treated similarly to updates: you specify the key or pseudo-key columns that identify the rows to delete.

Example 14–3 DBMS_XMLSTORE.DELETEXML Example

```
SELECT employee_id FROM employees WHERE employee_id = 188;

EMPLOYEE_ID
-----------
        188

1 row selected.

DECLARE
  delCtx DBMS_XMLSTORE.ctxType;
  rows NUMBER;
  xmlDoc CLOB :=
    '<ROWSET>
       <ROW>
         <EMPLOYEE_ID>188</EMPLOYEE_ID>
         <DEPARTMENT_ID>50</DEPARTMENT_ID>
       </ROW>
     </ROWSET>';
BEGIN
  delCtx  := DBMS_XMLSTORE.newContext('HR.EMPLOYEES');
  DBMS_XMLSTORE.setKeyColumn(delCtx, 'EMPLOYEE_ID');
  rows := DBMS_XMLSTORE.deleteXML(delCtx, xmlDoc);
  DBMS_XMLSTORE.closeContext(delCtx);
END;
/

SELECT employee_id FROM employees WHERE employee_id = 188;

no rows selected.
```

15

Java DOM API for XMLType

This chapter describes how to use `XMLType` in Java, including fetching `XMLType` data through Java Database Connectivity (JDBC).

This chapter contains these topics:

- Overview of Java DOM API for XMLType
- Java DOM API for XMLType
- Loading a Large XML Document into the Database using JDBC
- Java DOM API for XMLType Features
- Java DOM API for XMLType Classes
- Handling Large Nodes using Java
- Using the Java DOM API and JDBC with Binary XML

Overview of Java DOM API for XMLType

Oracle XML DB supports the Java Document Object Model (DOM) Application Program Interface (API) for `XMLType`. This is a generic API for client and server, for both XML schema-based and non-schema-based documents. It is implemented using Java package `oracle.xml.parser.v2`. DOM is a tree-based object representation of XML documents in dynamic memory that enables programmatic access to their elements and attributes. The DOM object and interface are part of a W3C recommendation. DOM views the parsed document as a tree of objects.

To access `XMLType` data using JDBC, use the class `oracle.xdb.XMLType`.

For XML documents that do not conform to any XML schema, use the Java DOM API for `XMLType`, because it can handle *any* valid XML document.

See Also: *Oracle Database XML Java API Reference*

Java DOM API for XMLType

Java DOM API for `XMLType` handles all kinds of valid XML documents, irrespective of how they are stored in Oracle XML DB. It presents to the application a uniform view of the XML document, whether it is XML schema-based or non-schema-based and whatever the underlying storage model. Java DOM API works on both client and server.

As discussed in Chapter 13, "PL/SQL APIs for XMLType", the Oracle XML DB DOM APIs are compliant with the W3C DOM Level 1.0 and Level 2.0 Core Recommendation.

The Java DOM API for `XMLType` can be used to construct an `XMLType` instance from data encoded in different character sets.

You can use the Java DOM API for `XMLType` to access XML documents stored in Oracle XML DB Repository from Java applications. Naming conforms to the Java binding for DOM as specified by the W3C DOM Recommendation. The repository can contain both XML schema-based and non-schema-based documents.

Accessing XMLType Data using JDBC

JDBC is a SQL-based way for Java applications to access any data in Oracle Database, including XML documents in Oracle XML DB. You use Java class `oracle.xdb.XMLType`, method `createXML()` to create XML data.

> **Note:** Use the thick driver with method `XMLType.createXML()` if you pass a stream as input. You cannot use the thin driver in this case.

Using XMLType Data with JDBC

Starting with Oracle Database Release 2 (11.2.0.2), Oracle supports the JDBC 4.0 standard data type for XML data, `java.sql.SQLXML`. By default, method `getObject()` returns an object of type `oracle.xdb.XMLType`, but if you set option `getObjectReturnsXMLType` to `true` when compiling your program with `javac`, then `getObject` returns an object of type `java.sql.SQLXML`. Use this command-line option to make that change in the default behavior:

```
-Doracle.jdbc.getObjectReturnsXMLType="true"
```

How Java Applications Use JDBC to Access XML Documents in Oracle XML DB

JDBC users can query an `XMLType` table to obtain a JDBC `XMLType` interface that supports all SQL/XML functions supported by SQL data type `XMLType`. The Java (JDBC) API for `XMLType` interface can implement the DOM document interface.

Example 15–1 illustrates how to use JDBC to query an `XMLType` table:

Example 15–1 Querying an XMLType Table using JDBC

```
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;
while(orset.next())
  {
  // get the XMLType
  XMLType poxml = (XMLType)orset.getObject(1);
  // get the XMLDocument as a string...
  Document podoc = (Document)poxml.getDOM();
  }
```

You can select `XMLType` data using JDBC in any of these ways:

- Use SQL/XML function `XMLSerialize` in SQL, and obtain the result as an `oracle.sql.CLOB`, `java.lang.String` or `oracle.sql.BLOB` in Java. The Java snippet in Example 15–2 illustrates this.

- Call method `getObject()` in the `PreparedStatement` to obtain the whole `XMLType` instance. The return value of this method is of type `oracle.xdb.XMLType`. Then you can use Java functions on class `XMLType` to access the data. Example 15–3 shows how to do this.

Example 15–2 Selecting XMLType Data using getStringVal() and getCLOB()

```
DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
Connection conn =
  DriverManager.getConnection("jdbc:oracle:oci8:@", "QUINE", "CURRY");
OraclePreparedStatement stmt =
  (OraclePreparedStatement) conn.prepareStatement(
     "SELECT XMLSerialize(DOCUMENT e.poDoc AS CLOB) poDoc, " +
     "XMLSerialize(DOCUMENT e.poDoc AS VARCHAR2(2000)) poString " +
     " FROM po_xml_tab e");
ResultSet rset = stmt.executeQuery();
OracleResultSet orset = (OracleResultSet) rset;
while(orset.next())
  {
  // 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
  }
```

Example 15–3 shows the use of method `getObject()` to directly obtain an `XMLType` instance from `ResultSet`.

Example 15–3 Returning XMLType Data using getObject()

```
import oracle.xdb.XMLType;
...
PreparedStatement stmt =  conn.prepareStatement(
  "SELECT e.poDoc FROM po_xml_tab e");
ResultSet rset = stmt.executeQuery();
while(rset.next())
{
// get the XMLType
XMLType poxml = (XMLType)rset.getObject(1);
// get the XML as a string...
String poString = poxml.getStringVal();
 }
```

Example 15–4 shows how to bind an output parameter of type `XMLType` to a SQL statement. The output parameter is registered as having data type `XMLType`.

Example 15–4 Returning XMLType Data using an Output Parameter

```
public void doCall (String[] args)
  throws Exception
  {
//  CREATE OR REPLACE FUNCTION getPurchaseOrder(reference VARCHAR2)
//  RETURN XMLTYPE
//  AS
//    xml XMLTYPE;
//  BEGIN
//    SELECT OBJECT_VALUE INTO xml
//      FROM purchaseorder
//      WHERE XMLCast(XMLQuery('$p/PurchaseOrder/Reference'
```

```
//                              PASSING OBJECT_VALUE AS "p" RETURNING CONTENT)
//                      AS VARCHAR2(30))
//              = reference;
//      RETURN xml;
//  END;
    String SQLTEXT = "{? = call getPurchaseOrder('BLAKE-2002100912333601PDT')}";
    CallableStatement sqlStatement = null;
    XMLType xml = null;
    super.doSomething(args);
    createConnection();
    try
    {
      System.out.println("SQL := " + SQLTEXT);
      sqlStatement = getConnection().prepareCall(SQLTEXT);
      sqlStatement.registerOutParameter (1, OracleTypes.OPAQUE,"SYS.XMLTYPE");
      sqlStatement.execute();
      xml = (XMLType) sqlStatement.getObject(1);
      System.out.println(xml.getStringVal());
    }
    catch (SQLException SQLe)
    {
      if (sqlStatement != null)
      {
        sqlStatement.close();
        throw SQLe;
      }
    }
```

Manipulating XML Database Documents using JDBC

You can also update, insert, and delete `XMLType` data using Java Database Connectivity (JDBC).

> **Note:** `XMLType` methods `extract()`, `transform()`, and `existsNode()` work only with the OCI driver.
>
> Not all `oracle.xdb.XMLType` functions are supported by the thin JDBC driver. If you do not use `oracle.xdb.XMLType` classes and the OCI driver, you could lose performance benefits associated with the intelligent handling of XML.

You can update, insert, or delete `XMLType` data in either of these ways:

- Bind a `CLOB` instance or a string to an `INSERT`, `UPDATE`, or `DELETE` statement, and use the `XMLType` constructor inside SQL to construct the XML instance. Example 15–5 illustrates this.
- Use `setObject()` in the `PreparedStatement` to set the entire `XMLType` instance. Example 15–6 illustrates this.

Example 15–5 Updating XMLType Data using SQL UPDATE with Constructor XMLType

```
OraclePreparedStatement stmt =
    (OraclePreparedStatement) conn.prepareStatement(
        "UPDATE po_xml_tab SET poDoc = XMLType(?)");

// 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();
```

Example 15–6 Updating XMLType Data using SQL UPDATE with setObject()

```
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();
```

When selecting `XMLType` values, JDBC describes the column as an opaque type. You can select the column type name and compare it with `XMLTYPE` to see whether you are dealing with an `XMLType` instance. Example 15–7 illustrates this.

Example 15–7 Retrieving Metadata about XMLType Data using JDBC

```
import oracle.sql.*;
import oracle.jdbc.*;
...
OraclePreparedStatement stmt =
    (OraclePreparedStatement) conn.prepareStatement(
        "SELECT poDoc FROM po_xml_tab");

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

// 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 XMLTYPE
if (mdata.getColumnType(1) == OracleTypes.OPAQUE &&
    mdata.getColumnTypeName(1).compareTo("SYS.XMLTYPE") == 0)
{
   // It is an XMLtype
}
```

Example 15–8 updates element `discount` inside element `PurchaseOrder` stored in an `XMLType` column. It uses JDBC and class `oracle.xdb.XMLType`. It uses the XML parser to update a DOM tree and write the updated XML value to the `XMLType` column.

Example 15–8 Updating an Element in an XMLType Column using JDBC

```
-- Create po_xml_hist table to store old PurchaseOrders
CREATE TABLE po_xml_hist (xpo XMLType);
/* NOTE: You must have xmlparserv2.jar and xdb.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 = "QUINE";
  static String pass = "CURRY";
  static String qryStr =
  "SELECT x.poDoc from po_xml_tab x " +
  "WHERE XMLCast(XMLQuery('/PO/PONO/text()'" +
  " PASSING x.poDoc RETURNING CONTENT)" +
  " AS NUMBER)" +
  " = 200";
  static String updateXML(String xmlTypeStr)
  Java DOM API for XMLType
  Beta Draft Java DOM API for XMLType 15-7
    {
    System.out.println("\n==============================");
    System.out.println("xmlType.getStringVal():");
    System.out.println(xmlTypeStr);
    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;
    }
  }

public static void main(String args[]) throws Exception
  {
  try {
      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)orset.getObject(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 " +
        "XMLCast(XMLQuery('/PO/PONO/text()'" +
        " PASSING value(xmltab) RETURNING CONTENT)" +
        " AS NUMBER)" +
        "= 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" +
                "XMLCast(XMLQuery('/PurchaseOrder/PONO/text()'" +
                " PASSING value(xmltab) RETURNING CONTENT)" +
                " AS NUMBER)" +
                "= 1001");
      System.out.println("PurchaseOrder 1001 deleted!");
      }
  catch(Exception e)
    {
    e.printStackTrace(System.out);
    }
  }

SELECT x.xpo.getCLOBVal() FROM po_xml x;
```

Example 15–9 shows the updated purchase order that results from Example 15–8.

Example 15–9 Updated Purchase-Order Document

```
<?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>
    <QUANTITY>1</QUANTITY>
    <DISCOUNT>10</DISCOUNT>
   </LINEITEM_TYP>
   <LINEITEM_TYP LineItemNo="2">
    <ITEM StockNo="1011">
     <PRICE>4500.23</PRICE>
     <TAXRATE>2</TAXRATE>
    </ITEM>
    <QUANTITY>2</QUANTITY>
    <DISCOUNT>10</DISCOUNT>
   </LINEITEM_TYP>
  </LINEITEMS>
  <SHIPTOADDR>
   <STREET>55 Madison Ave</STREET>
   <CITY>Madison</CITY>
   <STATE>WI</STATE>
   <ZIP>53715</ZIP>
  </SHIPTOADDR>
</PurchaseOrder>
```

Example 15–10 does all of the following:

- Selects an `XMLType` instance from an `XMLType` table
- Extracts portions of the `XMLType` instance, based on an XPath expression
- Checks for the existence of elements
- Transforms the `XMLType` instance to another XML format based on XSL
- Checks the validity of the `XMLType` document against an XML schema

Example 15–10 Manipulating an XMLType Column using JDBC

```
import java.sql.*;
import java.io.*;
import java.net.*;
import java.util.*;
import oracle.xml.parser.v2.*;
import oracle.xml.parser.schema.*;
import org.xml.sax.*;
import org.w3c.dom.*;
import oracle.xml.sql.dataset.*;
import oracle.xml.sql.query.*;
import oracle.xml.sql.docgen.*;
```

```
import oracle.xml.sql.*;
import oracle.jdbc.driver.*;
import oracle.sql.*;
import oracle.xdb.XMLType;
public class tkxmtpk1
{
static String conStr = "jdbc:oracle:oci8:@";
static String user = "tpjc";
static String pass = "tpjc";
static String qryStr = "select x.resume from t1 x where id<3";
static String xslStr =
"<?xml version='1.0'?> " +
"<xsl:stylesheet version='1.0' xmlns:xsl='http://www.w3.org/1" +
"999/XSL/Transform'> " +
"<xsl:template match='ROOT'> " +
"<xsl:apply-templates/> " +
"</xsl:template> " +
"<xsl:template match='NAME'> " +
"<html> " +
" <body> " +
" This is Test " +
" </body> " +
"</html> " +
"</xsl:template> " +
"</xsl:stylesheet>";
static void parseArg(String args[])
{
conStr = (args.length >= 1 ? args[0]:conStr);
user = (args.length >= 2 ? args[1].substring(0, args[1].indexOf("/")):user);
pass = (args.length >= 2 ? args[1].substring(args[1].indexOf("/")+1):pass);
qryStr = (args.length >= 3 ? args[2]:qryStr);
}
/**
* Print the byte array contents
*/
static void showValue(byte[] bytes) throws SQLException
{
if (bytes == null)
System.out.println("null");
else if (bytes.length == 0)
System.out.println("empty");
else
{
for(int i=0; i<bytes.length; i++)
System.out.print((bytes[i]&0xff)+" ");
System.out.println();
}
}
public static void main(String args[]) throws Exception
{
tkxmjnd1 util = new tkxmjnd1();
try{
if(args != null)
parseArg(args);
// System.out.println("conStr=" + conStr);
System.out.println("user/pass=" + user + "/" +pass );
System.out.println("qryStr=" + qryStr);
DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
Connection conn = DriverManager.getConnection(conStr, user, pass);
Statement s = conn.createStatement();
```

```
ResultSet rset = s.executeQuery(qryStr);
OracleResultSet orset = (OracleResultSet) rset;
OPAQUE xml;
while(orset.next()){
oracle.xdb.XMLType xt = (oracle.xdb.XMLType)(orset.getObject(1));
System.out.println("Testing getDOM() ...");
Document doc = xt.getDOM();
util.printDocument(doc);
System.out.println("Testing getBytesValue() ...");
showValue(xt.getBytesValue());
System.out.println("Testing existsNode() ...");
try {
System.out.println("existsNode(/)" + xt.existsNode("/", null));
}
Java DOM API for XMLType
Beta Draft Java DOM API for XMLType 15-11
catch (SQLException e) {
System.out.println("Thin driver Expected exception: " + e);
}
System.out.println("Testing extract() ...");
try {
XMLType xt1 = xt.extract("/RESUME", null);
System.out.println("extract RESUME: " + xt1.getStringVal());
System.out.println("should be Fragment: " + xt1.isFragment());
}
catch (SQLException e) {
System.out.println("Thin driver Expected exception: " + e);
}
System.out.println("Testing isFragment() ...");
try {
System.out.println("isFragment = " + xt.isFragment()); }
catch (SQLException e)
{
System.out.println("Thin driver Expected exception: " + e);
}
System.out.println("Testing isSchemaValid() ...");
try {
System.out.println("isSchemaValid(): " + xt.isSchemaValid(null,"RES UME"));
}
catch (SQLException e) {
System.out.println("Thin driver Expected exception: " + e);
}
System.out.println("Testing transform() ...");
System.out.println("XSLDOC: \n" + xslStr + "\n");
try {
/* XMLType xslDoc = XMLType.createXML(conn, xslStr);
System.out.println("XSLDOC Generated");
System.out.println("After transformation:\n" + (xt.transform(xslDoc,
null)).getStringVal()); */
System.out.println("After transformation:\n" + (xt.transform(null,
null)).getStringVal());
}
catch (SQLException e) {
System.out.println("Thin driver Expected exception: " + e);
}
System.out.println("Testing createXML(conn, doc) ...");
try {
XMLType xt1 = XMLType.createXML(conn, doc);
System.out.println(xt1.getStringVal());
}
```

```
catch (SQLException e) {
System.out.println("Got exception: " + e);
}
}
}
catch(Exception e)
{
e.printStackTrace(System.out);
}
}
}
```

Loading a Large XML Document into the Database using JDBC

If a large XML document (greater than 4000 characters, typically) is inserted into an `XMLType` table or column using a `String` object in JDBC, this run-time error occurs:

```
"java.sql.SQLException: Data size bigger than max size for this type"
```

This error can be avoided by using a Java `CLOB` object to hold the large XML document. Example 15–11 shows code that uses this technique. It defines `XMLType` method `insertXML()`, which can be used to insert a large XML document into `XMLType` column `purchaseOrder` of table `poTable`. The same approach can be used for an `XMLType` table.

Method `insertXML()` uses a `CLOB` object that contains the XML document. It creates the `CLOB` object using class `oracle.sql.CLOB` on the client side. This class is the Oracle JDBC driver implementation of the standard JDBC interface `java.sql.Clob`. Method `insertXML()` binds the `CLOB` object to a JDBC prepared statement, which inserts the data into the `XMLType` column.

The prerequisites for using `insertXML()` are as follows:

- Oracle Database, release 9.2.0.1 or later.
- The target database table. Execute the following SQL before running the example:

  ```
  CREATE TABLE poTable (purchaseOrder XMLType);
  ```

The formal parameters of `XMLType` method `insertXML()` are as follows:

- *`xmlData`* – XML data to be inserted into the `XMLType` column
- *`conn`* – database connection object (Oracle Connection Object)

Example 15–11 Java Method insertXML()

```
...
import oracle.sql.CLOB;
import java.sql.Connection;
import java.sql.SQLException;
import java.sql.PreparedStatement;
...

private void insertXML(String xmlData, Connection conn) {
  CLOB clob = null;
  String query;
    // Initialize statement Object
  PreparedStatement pstmt = null;
  try{
    query = "INSERT INTO potable (purchaseOrder) VALUES (XMLType(?)) ";
    // Get the statement Object
```

```
    pstmt = conn.prepareStatement(query);
    // xmlData is the string that contains the XML Data.
    // Get the CLOB object.
    clob = getCLOB(xmlData, conn);
    // Bind this CLOB with the prepared Statement
    pstmt.setObject(1, clob);
    // Execute the Prepared Statement
    if (pstmt.executeUpdate () == 1) {
    System.out.println ("Successfully inserted a Purchase Order");
    }
  } catch(SQLException sqlexp){
    sqlexp.printStackTrace();
  } catch(Exception exp){
    exp.printStackTrace();
  }
}
```

Java method `insertXML()` calls method `getCLOB()` to create and return the `CLOB` object that holds the XML data. The formal parameters of method `getCLOB()`, which is defined in Example 15–12, are as follows:

- *`xmlData`* – XML data to be inserted into the `XMLType` column
- *`conn`* – database connection object (Oracle Connection Object)

Example 15–12 Java Method getCLOB()

```
...
import oracle.sql.CLOB;
import java.sql.Connection;
import java.sql.SQLException;
import java.io.Writer;
...

private CLOB getCLOB(String xmlData, Connection conn) throws SQLException{
  CLOB tempClob = null;
  try{
    // If the temporary CLOB has not yet been created, create one
    tempClob = CLOB.createTemporary(conn, true, CLOB.DURATION_SESSION);

    // Open the temporary CLOB in readwrite mode, to enable writing
    tempClob.open(CLOB.MODE_READWRITE);
    // Get the output stream to write
    Writer tempClobWriter = tempClob.getCharacterOutputStream();
    // Write the data into the temporary CLOB
    tempClobWriter.write(xmlData);

    // Flush and close the stream
    tempClobWriter.flush();
    tempClobWriter.close();

    // Close the temporary CLOB
    tempClob.close();
  } catch(SQLException sqlexp){
    tempClob.freeTemporary();
    sqlexp.printStackTrace();
  } catch(Exception exp){
    tempClob.freeTemporary();
    exp.printStackTrace();
  }
  return tempClob;
```

```
}
```

See Also: *Oracle Database SecureFiles and Large Objects Developer's Guide*

Java DOM API for XMLType Features

When you use the Java DOM API to retrieve XML data from Oracle XML DB:

- If the connection is *thin*, you get an **XML**`Document` instance
- If the connection is *thick* or *kprb*, you get an **XDB**`Document` instance with method `getDOM()` and an **XML**`Document` instance with method `getDocument()`. Method `getDOM()` and class `XDBDocument` are deprecated.

Both `XMLDocument` and `XDBDocument` (which is deprecated) are instances of the W3C Document Object Model (DOM) interface. From this document interface you can access the document elements and perform all the operations specified in the W3C DOM Recommendation. The DOM works on:

- Any type of XML document, schema-based or non-schema-based
- Any type of underlying storage used by the document:
  - Character Large Object (CLOB)
  - Binary Large Object (BLOB)
  - object-relational

The Java DOM API for `XMLType` supports deep and shallow searching in the document to retrieve children and properties of XML objects such as name, namespace, and so on. Conforming to the DOM 2.0 recommendation, Java DOM API for `XMLType` is namespace aware.

The Java API for `XMLType` also lets applications create XML documents programmatically, even on the fly (dynamically). Such documents can conform to a registered XML schema or not.

Creating XML Schema-Based Documents

To create XML schema-based documents, Java DOM API for `XMLType` uses an extension to specify which XML schema URL to use. For XML schema-based documents, it also verifies that the DOM being created conforms to the specified XML schema, that is, that the appropriate children are being inserted under the appropriate documents.

Note: The Java DOM API for `XMLType` does *not* perform type and constraint checks.

Once the DOM object has been created, it can be saved to Oracle XML DB Repository using the Oracle XML DB resource API for Java. The XML document is stored in the appropriate format:

- As a BLOB instance for non-schema-based documents.
- In the format specified by the XML schema for XML schema-based documents.

Example 15–13 shows how you can use the Java DOM API for `XMLType` to create a DOM object and store it in the format specified by the associated XML schema. Validation against the XML schema is not shown here.

Example 15–13 Creating a DOM Object with the Java DOM API

```
import oracle.xdb.XMLType;
...
OraclePreparedStatement stmt =
    (OraclePreparedStatement) conn.prepareStatement(
        "update po_xml_XMLTypetab 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);
Document poDOM = (Document)poXML.getDOM();

Element rootElem = poDOM.createElement("PO");
poDOM.insertBefore(poDOM, rootElem, null);

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

JDBC or SQLJ

An `XMLType` instance is represented in Java by `oracle.xdb.XMLType`. When an instance of `XMLType` is fetched using JDBC, it is automatically manifested as an object of the provided `XMLType` class. Similarly, objects of this class can be bound as values to Data Manipulation Language (DML) statements where an `XMLType` is expected. The same action is supported in SQLJ clients.

Java DOM API for XMLType Classes

Oracle XML DB supports the W3C DOM Level 2 Recommendation. In addition to the W3C Recommendation, Oracle XML DB DOM API also provides Oracle-specific extensions, to facilitate your application interfacing with Oracle XDK for Java. A list of the Oracle extensions is found at:

http://www.oracle.com/technology/tech/xml/

`XMLDocument` is a class that represents the DOM for the instantiated XML document. You can retrieve the `XMLType` value from the XML document using the constructor `XMLType` constructor that takes a `Document` argument:

```
XMLType createXML(Connection conn, Document domdoc)
```

Table 15–1 lists the Java DOM API for `XMLType` classes and the W3C DOM interfaces they implement. The Java DOM API classes are all in package `oracle.xml.parser.v2`.

Table 15–1 Java DOM API for XMLType: Classes

Java DOM API for XMLType Class	W3C DOM Interface Recommendation Class
`XMLDocument`	`org.w3c.dom.Document`
`XMLCDATA`	`org.w3c.dom.CDataSection`
`XMLComment`	`org.w3c.dom.Comment`
`XMLPI`	`org.w3c.dom.ProcessingInstruction`
`XMLText`	`org.w3c.dom.Text`
`XMLEntity`	`org.w3c.dom.Entity`
`DTD`	`org.w3c.dom.DocumentType`

Table 15–1 (Cont.) Java DOM API for XMLType: Classes

Java DOM API for XMLType Class	W3C DOM Interface Recommendation Class
XMLNotation	org.w3c.dom.Notation
XMLAttr	org.w3c.dom.Attribute
XMLDomImplementation	org.w3c.dom.DOMImplementation
XMLElement	org.w3c.dom.Element
XMLAttrList	org.w3c.dom.NamedNodeMap
XMLNode	org.w3c.dom.Node

Java Methods That Are Deprecated or Not Supported

The following methods documented in release 2 (9.2.0.1) are no longer supported:

- XDBDocument.getElementByID
- XDBDocument.importNode
- XDBNode.normalize
- XDBNode.isSupported
- XDBDomImplementation.hasFeature

In addition, in releases prior to Oracle Database 11*g* Release 1, a different API, in package oracle.xdb.dom, was used for the Java DOM. Please refer to the documentation for such releases for more information on that deprecated API. The following classes in oracle.xdb.dom are *deprecated*. Use oracle.xml.parser.v2 classes instead.

- XDBAttribute – use XMLAttr
- XDBBinaryDocument
- XDBCData – use XMLCDATA
- XDBComment – use XMLComment
- XDBDocFragment – use XMLDocumentFragment
- XDBDocument – use XMLDocument
- XDBDocumentType – use DTD
- XDBDOMException – use XMLDomException
- XDBDomImplementation – use XMLDomImplementation
- XDBElement – use XMLElement
- XDBEntity – use XMLEntity
- XDBEntityReference – use XMLEntityReference
- XDBNamedNodeMap – use XMLAttrList
- XDBNode – use XMLNode
- XDBNodeList – use NodeList
- XDBNotation – use XMLNotation
- XDBProcInst – use XMLPI
- XDBText – use XMLText

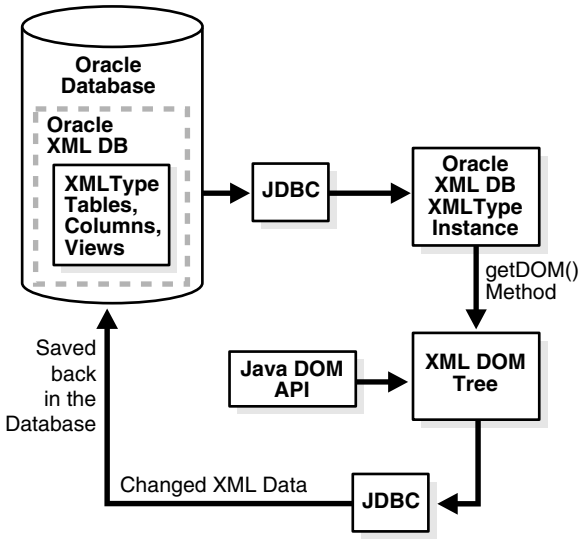
Using the Java DOM API for XMLType

Figure 15–1 illustrates how to use the Java DOM API for `XMLType`.[1] These are the steps:

1. Retrieve the XML data from the `XMLType` table or `XMLType` column in the table. When you fetch XML data, Oracle creates an instance of an `XMLType`. You can then use method `getDocument()` to retrieve a Document instance. You can then manipulate elements in the DOM tree using Java DOM API for `XMLType`.
2. Use the Java DOM API for `XMLType` to manipulate elements of the DOM tree. The XMLType instance holds the modified data, but the data is sent back using a JDBC update.

The `XMLType` and `XMLDocument` instances should be closed using method `close()` in the respective classes. This frees any underlying memory that is held.

Figure 15–1 Using the Java DOM API for XMLType



Handling Large Nodes using Java

Prior to Oracle Database 11g Release 1 (11.1), there were restrictions on the size of nodes to less than 64 KB, because the Java methods to set and get a node value supported only arguments of type `java.lang.String`. The maximum size of a string is dependent on the implementation of the Java VM, but it is bounded. Prior to Release 11.1, the Java DOM APIs to manage a node value, contained within class `oracle.xdb.dom.XDBNode.java`, were these:

```
public String getNodeValue ();
public void setNodeValue (String value);
```

The Java DOM APIs to manage an attribute, contained within class `oracle.xdb.dom.XDBAttribute.java`, were these:

```
public String getValue ();
public void setValue (String value);
```

[1] This assumes that your XML data is pre-registered with an XML schema, and that it is stored in an `XMLType` column.

Package `oracle.xdb.dom` is deprecated, starting with Oracle Database 11g Release 1 (11.1). Java classes `XDBNode` and `XDBAttribute` in that package are replaced by classes `XMLNode` and `XMLAttr`, respectively, in package `oracle.xml.parser.v2`. In addition, these DOM APIs were extended in Release 11.1 to support text and binary node values of arbitrary size.

Note: The large-node feature works only with a thick or kprb connection. It does not work with a thin connection.

See Also:

- "Large Node Handling using DBMS_XMLDOM" on page 13-12 for information on using PL/SQL with large nodes
- "Java Methods That Are Deprecated or Not Supported" on page 15-15 for more about deprecated classes `XDBNode` and `XDBAttribute`

Stream Extensions to Java DOM

All `Java String`, `Reader`, and `Writer` data is represented in `UCS2`, which might be different from the database character set. Additionally, node character data is tagged with a character set id, which is set at the time the node value is populated.

The following methods of `oracle.xml.parser.v2.XMLNode.java` can be used to access nodes of size greater than 64 KB. These APIs throw exceptions if you try to get or set a node that is not a leaf node (attribute, `PI`, `CDATA`, and so on). Also, be sure to use `close()` which actually writes the value and frees resources used to maintain the state for streaming access to nodes.

Get-Pull Model

For a binary input stream:

```
public java.io.InputStream getNodeValueAsBinaryStream ()
       throws java.io.IOException, DOMException;
```

Method `getNodeValueAsBinaryStream()` returns an instance of `java.io.InputStream` that can be read using the defined methods for this class. The data type of the node must be `RAW` or `BLOB`. If not, an `IOException` is thrown. The following example fragment illustrates reading the value of a node in binary 50-byte segments:

```
...
oracle.xml.parser.v2.XMLNode node = null;
...

java.io.InputStream value = node.getNodeValueAsBinaryStream ();
// now read InputStream...
byte buffer [] = new byte [50];
int returnValue = 0;
while ((returnValue = value.read (buffer)) != -1)
{
   // process next 50 bytes of node
}
...
```

For a character input stream:

```
public java.io.Reader getNodeValueAsCharacterStream()
       throws java.io.IOException, DOMException;
```

Method `getNodeValueAsCharacterStream()` returns an instance of `java.io.Reader` that can be read using the defined methods for this class. If the data type of the node is neither character nor `CLOB`, the node data is first converted to character. All node data is ultimately in character format and is converted to `UCS2`, if necessary. The following example fragment illustrates reading the node value in segments of 50 characters:

```
...
oracle.xml.parser.v2.XMLNode node = null;
...

java.io.Reader value = node.getNodeValueAsCharacterStream ();
// now read InputStream
char buffer [] = new char [50];
int returnValue = 0;
while ((returnValue = value.read (buffer)) != -1)
{
   // process next 50 characters of node
}
...
```

Get-Push Model

For a binary output stream:

```
public void getNodeValueAsBinaryStream (java.io.OutputStream pushValue)
       throws java.io.IOException,  DOMException;
```

The state of the `java.io.OutputStream` specified by `pushValue` must be open. The data type of the node must be `RAW` or `BLOB`. If not, an `IOException` is thrown. The node binary data is written to `pushValue` using method `write()` of `OutputStream`, and method `close()` is called when the node value has been completely written to the stream.

For a character output stream:

```
public void getNodeValueAsCharacterStream (java.io.Writer pushValue)
       throws java.io.IOException, DOMException;
```

The state of the `java.io.Writer` specified by `pushValue` must be open. If the data type of the node is neither character nor `CLOB`, then the data is first converted to character. The node data, always in character format, is converted, as necessary, to `UCS2` and then pushed into the `java.io.Writer`.

Set-Pull Model

For a binary input stream:

```
public void setNodeValueAsBinaryStream (java.io.InputStream pullValue)
       throws java.io.IOException, DOMException;
```

The state of the `java.io.InputStream` specified by `pullValue` must be open. The data type of the node must be `RAW` or `BLOB`. If not, an `IOException` is thrown. The binary data from `pullValue` is read in its entirety using method `read()` of `InputStream` and replaces the node value.

```
import java.io.InputStream;
import oracle.xml.parser.*;
```

```
...
oracle.xml.parser.v2.XMLNode node = null;
...
byte [] buffer = new byte [500];
java.io.InputStream  istream; //user-defined input stream
node.setNodeValueAsBinaryStream (istream);
```

For a character input stream:

```
public void setNodeValueAsCharacterStream (java.io.Reader pullValue)
       throws java.io.IOException, DOMException;
```

The state of the `java.io.Reader` specified by `pullValue` must be open. If the data type of the node is neither character nor `CLOB`, the character data is converted from `UCS2` to the node data type. If the data type of the node is character or `CLOB`, then the character data read from `pullValue` is converted from `UCS2` to the character set of the node.

Set-Push Model

For a binary output stream:

```
public java.io.OutputStream setNodeValueAsBinaryStream ()
     throws java.io.IOException, DOMException;
```

Method `setNodeValueAsBinaryStream()` returns an instance of `java.io.OutputStream`, into which the caller can write the node value. The data type of the node must be `RAW` or `BLOB`. Otherwise, an `IOException` is raised. The following example fragment illustrates setting the value of a node to binary data by writing to the implementation of `java.io.OutputStream` provided by Oracle XML DB or Oracle XDK.

For a character output stream:

```
public java.io.Writer setNodeValueAsCharacterStream ()
       throws java.io.IOException, DOMException;
```

Method `setNodeValueAsCharacterStream()` returns an instance of `java.io.Writer` into which the caller can write the node value. The character data written is first converted from `UCS2` to the node character set, if necessary. If the data type of the node is neither character nor `CLOB`, then the character data is converted to the node data type. Similarly, the following example fragment illustrates setting the value of a node to character data by writing to the implementation of `java.io.Writer` provided by Oracle XML DB or Oracle XDK.

```
import java.io.Writer;
import oracle.xml.parser.*;
...
oracle.xml.parser.v2.XMLNode node = null;
...
char [] buffer = new char [500];
java.io.Writer  writer = node.setNodeValueAsCharacterStream ();
for (int k = 0; k < 10; k++)
  {
    byte segment [] = new byte [50];
    // copy next subset of buffer into segment
    writer.write (segment);
  }
writer.flush ();
writer.close();
```

See Also:

- *Oracle Database XML Java API Reference*
- *Oracle Database XML C API Reference* for information about C functions for large nodes

Oracle XML DB creates a `writer` or `OutputStream` and passes it to the user who calls method `write()` repeatedly until the complete node value has been written. The new node value is reflected only when the user calls method `close()`.

Using the Java DOM API and JDBC with Binary XML

XML data can be stored in Oracle XML DB using `XMLType`, and one of the storage models for this abstract data type is binary XML. Binary XML is a compact, XML Schema-aware encoding of XML data. You can use it as a storage model for `XMLType` in the database, but you can also use it for XML data located outside the database. Client-side processing of XML data can involve data stored in Oracle XML DB or transient data that resides outside the database.

You can use the Java DOM API for XML to read or write XML data that is encoded as binary XML from or to Oracle XML DB. Doing so involves the usual read and write procedures.

Binary XML is XML Schema-aware and can use various encoding schemes, depending on your needs and your data. Because of this, in order to manipulate binary XML data, you must have both the data and this metadata about the relevant XML schemas and encodings.

For `XMLType` data stored in the database, this metadata is also stored in the database. However, depending on how your database and data are set up, the metadata might not be on the same server as the data it applies to. If this is the case, then, before you can read or write binary XML data from or to the database, you must carry out these steps:

1. Create a context instance for the metadata.
2. Associate this context with a data connection that you use to access binary XML data in the database. A data connection can be a dedicated connection or a connection pool. You use methods `getDedicatedConn()` and `getConnPool()` in class `java.sql.Connection` to obtain handles to these two types of connection, respectively.

Then, when your application needs to encode or decode binary XML data on the data connection, it automatically fetches the metadata needed for that. The overall sequence of actions is thus as follows:

1. Create an XML data connection object, in class `java.sql.Connection`.
2. Create one or more metadata contexts, as needed, using method `BinXMLMetadataProviderFactory.createDBMetadataProvider()` in package `oracle.xml.binxml`. A metadata context is sometimes referred to as a metadata repository. You can create a metadata context from a dedicated connection or from a connection pool.
3. Associate the metadata context(s) with the binary XML data connection(s). Use method `DBBinXMLMetadataProvider.associateDataConnection()` in package `oracle.xml.binxml` to do this.
4. (Optional) If the XML data originated outside of the database, use method `oracle.xdb.XMLType.setFormatPref()` to specify that XML data to be sent

to the database from now on is in binary XML format. This applies to a DOM document (class `oracle.xdb.XMLType`). If you do not specify binary XML, the data is stored as text (`CLOB`).

5. Use the usual Java methods to read and write XML data from and to the database. Whenever it is needed for encoding or decoding binary XML documents, the necessary metadata is fetched automatically using the metadata context.

   Use the Java DOM API for XML to operate on the XML data at the client level.

Example 15–14 illustrates this.

Example 15–14 Using the Java DOM API with Binary XML

```
import java.sql.*;
import oracle.jdbc.*;
import oracle.jdbc.pool.OracleDataSource;
import oracle.xdb.XMLType;
import oracle.xml.binxml.*;

class tdadxdbxdb11jav001 {
  public static void printBinXML() throws SQLException, BinXMLException
  {
    // Create datasource to connect to local database
    OracleDataSource ods = new OracleDataSource();
    ods.setURL("jdbc:oracle:kprb");
    System.out.println("Starting Binary XML Java Example");
    // Create data connection
    Connection conn = ods.getConnection();
    // Create binary XML metadata context, using connection pool
    DBBinXMLMetadataProvider repos =
      BinXMLMetadataProviderFactory.createDBMetadataProvider();
    repos.setConnectionPool(ods);
    // Associate metadata context with data connection
    repos.associateDataConnection(conn);
    // Query XML data stored in XMLType column as binary XML
    Statement stmt = conn.createStatement();
    ResultSet rset = stmt.executeQuery("SELECT doc FROM po_binxmltab");
    // Get the XMLType object
    while (rset.next())
    {
      XMLType xmlobj = (XMLType) rset.getObject(1);
      // Perform XMLType operation
      String xmlvalue = xmlobj.getStringVal();
      System.out.println(xmlvalue);
    }
    // Close result set, statement, and connection
    rset.close();
    stmt.close();
    conn.close();
    System.out.println("Completed Binary XML Java Example");
  }
}
```

See Also:

- "XMLType Storage Models" on page 1-14
- *Oracle XML Developer's Kit Programmer's Guide*

16

Using the C API for XML

This chapter provides help for using the C API for XML with Oracle XML DB. It contains these topics:

- Overview of the C API for XML (Oracle XDK and Oracle XML DB)
- Using OCI and the C API for XML with Oracle XML DB
- XML Context Parameter for C DOM API Functions
- Initializing and Terminating an XML Context
- Using the C API for XML with Binary XML
- Using the Oracle XDK Pull Parser with Oracle XML DB
- Common XMLType Operations in C

Overview of the C API for XML (Oracle XDK and Oracle XML DB)

The C API for XML is used for both Oracle XDK (Oracle XML Developer's Kit) and Oracle XML DB. It is a C-based DOM[1] API for XML. It can be used for XML data that is inside or outside the database. This API also includes performance-improving extensions that you can use in Oracle XDK, for traditional XML storage, or in Oracle XML DB, for XML stored as an `XMLType` column in a table.

> **Note:** C DOM functions from releases prior to Oracle Database 10*g* Release 1 are supported only for backward compatibility.

The C API for XML is implemented on `XMLType` in Oracle XML DB. In the W3C DOM Recommendation, the term "document" is used in a broad sense (URI, file system, memory buffer, standard input and output).

The C API for XML is a combined programming interface that includes all of the functionality needed by Oracle XDK and Oracle XML DB applications. It provides XSLT and XML Schema implementations. Although the DOM 2.0 Recommendation was followed closely, some naming changes were required for mapping from the objected-oriented DOM 2.0 Recommendation to the flat C namespace. For example, method `getName()` was renamed to `getAttrName()`.

[1] DOM refers to compliance with the World Wide Web Consortium (W3C) DOM 2.0 Recommendation.

The C API for XML supersedes older Oracle APIs. In particular, the `oraxml` interface (top-level, DOM, SAX, and XSLT) and `oraxsd.h` (Schema) interfaces will be deprecated in a future release.

Using OCI and the C API for XML with Oracle XML DB

OCI applications typically operate on XML data stored in the server or created on the client. This section explains these two access methods in more detail.

Accessing XMLType Data Stored in the Database

Oracle XML DB provides support for storing and manipulating XML instances using abstract data type `XMLType`. These XML instances can be accessed and manipulated on the client side using the Oracle Call Interface (OCI) in conjunction with the C DOM API for XML. You can bind and define `XMLType` values using the C DOM structure `xmldocnode`. This structure can be used for binding, defining and operating on XML values in OCI statements. You can use OCI statements to select XML data from the server, which you can then use with C DOM API functions. Similarly, values can be bound back to SQL statements directly.

The main flow for an application program involves initializing the usual OCI handles, such as server handle and statement handle, and then initializing an XML context parameter. You can then either operate on XML instances in the database or create new instances on the client side. The initialized XML context can be used with all of the C DOM functions.

> **See Also:** "XML Context Parameter for C DOM API Functions" on page 16-2

Creating XMLType Instances on the Client

You can construct new `XMLType` instances on the client side using `XmlLoadDom()`, as follows:

1. Initialize the `xmlctx` as in Example 16–1.
2. Construct the XML data from a user buffer, local file, or URI. The return value, a (`xmldocnode*`), can be used in the rest of the common C API.
3. If required, you can cast (`xmldocnode *`) to (`void*`) and provide it directly as the bind value.

You can construct empty `XMLType` instances with `XMLCreateDocument()`. This is similar to using `OCIObjectNew()` for other types.

XML Context Parameter for C DOM API Functions

An *XML context* is a required parameter to all the C DOM API functions. This opaque context encapsulates information about the data encoding, the error message language, and so on. The contents of the context are different for Oracle XDK applications and Oracle XML DB. For Oracle XML DB, there are two OCI functions that initialize (`OCIXmlDbInitXmlCtx()`) and terminate (`OCIXmlDbFreeXmlCtx()`) an XML context.

OCIXmlDbInitXmlCtx() Syntax

The syntax of function `OCIXmlDbInitXmlCtx()` is as follows:

```
xmlctx *OCIXmlDbInitXMlCtx (OCIEnv        *envhp,
                            OCISvcHp      *svchp,
                            OCIError      *errhp,
                            ociXmldbparam *params,
                            ub4           num_params);
```

Table 16–1 describes the parameters.

Table 16–1 OCIXmlDbInitXMlCtx() Parameters

Parameter	Description
`envhp (IN)`	The OCI environment handle.
`svchp (IN)`	The OCI service handle.
`errhp (IN)`	The OCI error handle.
`params (IN)`	An array of optional values: ▪ OCI duration. Default value is `OCI_DURATION_SESSION`. ▪ Error handler, which is a user-registered callback: `void (*err_handler) (sword errcode, (CONST OraText *) errmsg);`
`num_params (IN)`	Number of parameters to be read from `params`.

OCIXmlDbFreeXmlCtx() Syntax

The syntax of function `OCIXmlDbFreeXmlCtx()` is as follows, where parameter `xctx (IN)` is the XML context to terminate.:

```
void OCIXmlDbFreeXmlCtx (xmlctx *xctx);
```

Initializing and Terminating an XML Context

Example 16–1 shows a C program that uses the C DOM API to construct an XML document and save it to Oracle Database in table `my_table`. It calls OCI functions `OCIXmlDbInitXmlCtx()` and `OCIXmlDbFreeXmlCtx()` to initialize and terminate the XML context. These OCI functions are defined in header file `ociXmldb.h`.

The C code in Example 16–1 assumes that the following SQL code has first been executed to create table `my_table` in database schema `capiuser`:

```
CONNECT CAPIUSER
Enter password: password

Connected.

CREATE TABLE my_table OF XMLType;
```

Example 16–1 shows how to use OCI functions `OCIXmlDbInitXmlCtx()` and `OCIXmlDbFreeXmlCtx()` to initialize and terminate the XML context.

Example 16–1 Using OCIXMLDBINITXMLCTX() and OCIXMLDBFREEXMLCTX()

```
#ifndef S_ORACLE
#include <s.h>
#endif
#ifndef ORATYPES_ORACLE
#include <oratypes.h>
#endif
```

```
#ifndef XML_ORACLE
#include <xml.h>
#endif
#ifndef OCIXML_ORACLE
#include <ocixml.h>
#endif
#ifndef OCI_ORACLE
#include <oci.h>
#endif
#include <string.h>

typedef struct test_ctx {
        OCIEnv *envhp;
        OCIError *errhp;
        OCISvcCtx *svchp;
        OCIStmt *stmthp;
        OCIServer *srvhp;
        OCIDuration dur;
        OCISession *sesshp;
        oratext *username;
        oratext *password;
} test_ctx;

/* Helper function 1: execute a sql statement which binds xml data */
STATICF sword exec_bind_xml(OCISvcCtx *svchp,
                    OCIError *errhp,
                    OCIStmt *stmthp,
                    void *xml,
                    OCIType *xmltdo,
                    OraText *sqlstmt);

/* Helper function 2: Initialize OCI handles and connect */
STATICF sword init_oci_handles(test_ctx *ctx);

/* Helper function 3: Free OCI handles and disconnect */
STATICF sword free_oci_handles(test_ctx *ctx);

void main()
{
  test_ctx temp_ctx;
  test_ctx *ctx = &temp_ctx;
  OCIType *xmltdo = (OCIType *) 0;
  xmldocnode *doc = (xmldocnode *)0;
  ocixmldbparam params[1];
  xmlnode *quux, *foo, *foo_data, *top;
  xmlerr err;
  sword status = 0;
  xmlctx *xctx;

  oratext ins_stmt[] = "insert into my_table values (:1)";
  oratext tlpxml_test_sch[] = "<TOP/>";
  ctx->username = (oratext *)"capiuser";
  ctx->password = (oratext *)"************"; /* Replace with real password */

  /* Initialize envhp, svchp, errhp, dur, stmthp */
  init_oci_handles(ctx);

  /* Get an xml context */
  params[0].name_ocixmldbparam = XCTXINIT_OCIDUR;
  params[0].value_ocixmldbparam = &ctx->dur;
```

```
xctx = OCIXmlDbInitXmlCtx(ctx->envhp, ctx->svchp, ctx->errhp, params, 1);

/* Start processing - first, check that this DOM supports XML 1.0 */
printf("\n\nSupports XML 1.0? : %s\n",
       XmlHasFeature(xctx, (oratext *) "xml", (oratext *) "1.0") ?
       "YES" : "NO");

/* Parse a document */
if (!(doc = XmlLoadDom(xctx, &err, "buffer", tlpxml_test_sch,
                       "buffer_length", sizeof(tlpxml_test_sch)-1,
                       "validate", TRUE, NULL)))
{
  printf("Parse failed, code %d\n", err);
}
else
{
  /* Get the document element */
  top = (xmlnode *)XmlDomGetDocElem(xctx, doc);

  /* Print out the top element */
  printf("\n\nOriginal top element is :\n");
  XmlSaveDom(xctx, &err, top, "stdio", stdout, NULL);

  /* Print out the document-note that the changes are reflected here */
  printf("\n\nOriginal document is :\n");
  XmlSaveDom(xctx, &err, (xmlnode *)doc, "stdio", stdout, NULL);

  /* Create some elements and add them to the document */
  quux = (xmlnode *) XmlDomCreateElem(xctx ,doc, (oratext *) "QUUX");
  foo = (xmlnode *) XmlDomCreateElem(xctx, doc, (oratext *) "FOO");
  foo_data = (xmlnode *) XmlDomCreateText(xctx, doc, (oratext *) "data");
  foo_data = XmlDomAppendChild(xctx, (xmlnode *) foo, (xmlnode *) foo_data);
  foo = XmlDomAppendChild(xctx, quux, foo);
  quux = XmlDomAppendChild(xctx, top, quux);

  /* Print out the top element */
  printf("\n\nNow the top element is :\n");
  XmlSaveDom(xctx, &err, top, "stdio", stdout, NULL);

  /* Print out the document. Note that the changes are reflected here */
  printf("\n\nNow the document is :\n");
  XmlSaveDom(xctx, &err, (xmlnode *)doc, "stdio", stdout, NULL);

  /* Insert the document into my_table */
  status = OCITypeByName(ctx->envhp, ctx->errhp, ctx->svchp,
                         (const text *) "SYS", (ub4) strlen((char *)"SYS"),
                         (const text *) "XMLTYPE",
                         (ub4) strlen((char *)"XMLTYPE"), (CONST text *) 0,
                         (ub4) 0, OCI_DURATION_SESSION, OCI_TYPEGET_HEADER,
                         (OCIType **) &xmltdo);
  if (status == OCI_SUCCESS)
  {
    exec_bind_xml(ctx->svchp, ctx->errhp, ctx->stmthp, (void *)doc, xmltdo,
                  ins_stmt);
  }
}
/* Free xml ctx */
OCIXmlDbFreeXmlCtx(xctx);

/* Free envhp, svchp, errhp, stmthp */
```

```
  free_oci_handles(ctx);
}
```

The output from compiling and running this C program is as follows:

```
Supports XML 1.0? : YES

Original top element is :
<TOP/>

Original document is :
<TOP/>

Now the top element is :
<TOP>
  <QUUX>
    <FOO>data</FOO>
  </QUUX>
</TOP>

Now the document is :
<TOP>
  <QUUX>
    <FOO>data</FOO>
  </QUUX>
</TOP>
```

This is the result of querying the constructed document in `my_table`:

```
SELECT * FROM my_table;

SYS_NC_ROWINFO$
---------------
<TOP>
  <QUUX>
    <FOO>data</FOO>
  </QUUX>
</TOP>

1 row selected.
```

Example 16–1 constructs an XML document using the C DOM API and saves it to the database. The code uses helper functions `exec_bind_xml`, `init_oci_handles`, and `free_oci_handles`, which are not listed here. The complete listing of this example, including the helper functions, can be found in Appendix A, "Oracle-Supplied XML Schemas and Examples", "Initializing and Terminating an XML Context (OCI)" on page A-47.

Example 16–4 queries table `my_table` to show the data that was inserted by Example 16–1.

Using the C API for XML with Binary XML

XML data can be stored in Oracle XML DB using `XMLType`, and one of the storage models for this abstract data type is binary XML. Binary XML is a compact, XML Schema-aware encoding of XML data. You can use it as a storage model for `XMLType` in the database, but you can also use it for XML data located outside the database. As explained in "Using OCI and the C API for XML with Oracle XML DB" on page 16-2,

client-side processing of XML data can involve data stored in Oracle XML DB or transient data that resides outside the database.

You can use the C API for XML to read or write XML data that is encoded as binary XML from or to Oracle XML DB. Doing so involves the usual read and write procedures.

Binary XML is XML Schema-aware and can use various encoding schemes, depending on your needs and your data. Because of this, in order to manipulate binary XML data, you must have both the data and this metadata about the relevant XML schemas and encodings.

For `XMLType` data stored in the database, this metadata is also stored in the database. However, depending on how your database and data are set up, the metadata might not be on the same server as the data it applies to. If this is the case, then, before you can read or write binary XML data from or to the database, you must carry out these steps:

1. Create a context instance for the metadata.
2. Associate this context with a data connection that you use to access binary XML data in the database. A data connection can be a dedicated connection (`OCISvcCtx`) or a connection pool (`OCICPool`).

Then, when your application needs to encode or decode binary XML data on the data connection, it automatically fetches the metadata needed for that. The overall sequence of actions is thus as follows:

1. Create the usual OCI handles for environment (`OCIEnv`), connection (`OCISvcCtx`), and error context (`OCIError`).
2. Create one or more metadata contexts, as needed. A metadata context is sometimes referred to as a metadata repository, and `OCIBinXMLReposCtx` is the OCI context data structure.

   You use `OCIBinXMLCreateReposCtxFromConn` to create a metadata context from a dedicated connection and `OCIBinXMLCreateReposCtxFromCPool` to create a context from a connection pool.
3. Associate the metadata context(s) with the binary XML data connection(s). You use `OCIBinXmlSetReposCtxForConn` to do this.
4. (Optional) If the XML data originated outside of the database, use `setPicklePreference` to specify that XML data to be sent to the database from now on is in binary XML format. This applies to a DOM document (`xmldomdoc`). If you do not specify binary XML, the data is stored as text (`CLOB`).
5. Use OCI libraries to read and write XML data from and to the database. Whenever it is needed for encoding or decoding binary XML documents, the necessary metadata is fetched automatically using the metadata context.

   Use the C DOM API for XML to operate on the XML data at the client level.

Example 16–2 illustrates this.

Example 16–2 Using the C API for XML with Binary XML

```
. . .
/* Private types and constants */
#define SCHEMA        (OraText *)"SYS"
#define TYPE          (OraText *)"XMLTYPE"
#define USER          (OraText *)"oe"
#define USER_LEN      (ub2)(strlen((char *)USER))
```

```
#define PWD           (OraText *)"oe"
#define PWD_LEN       (ub2)(strlen((char *)PWD))
#define NUM_PARAMS    1
STATICF void checkerr(OCIError *errhp, sword status);
STATICF sword create_env(OraText *user, ub2 user_len, OraText *pwd, ub2 pwd_len,
                         OCIEnv **envhp, OCISvcCtx **svchp, OCIError **errhp);
STATICF sword run_example(OCIEnv *envhp, OCISvcCtx *svchp, OCIError *errhp,
                          OCIDuration dur);
STATICF void cleanup(OCIEnv *envhp, OCISvcCtx *svchp, OCIError *errhp);

int main (int argc, char *argv[])
{
  OCIEnv     *envhp;
  OCISvcCtx  *svchp;
  OCIError   *errhp;
  printf("*** Starting Binary XML Example program\n");
  if (create_env(USER, USER_LEN, PWD, PWD_LEN, &envhp, &svchp, &errhp))
    {
      printf("FAILED: create_env()\n");
      cleanup(envhp, svchp, errhp);
      return OCI_ERROR;
    }
  if (run_example(envhp, svchp, errhp, OCI_DURATION_SESSION))
    {
      printf("FAILED: run_example()\n");
      cleanup(envhp, svchp, errhp);
      return OCI_ERROR;
    }
  cleanup(envhp, svchp, errhp);
  printf ("*** Completed Binary XML example\n");
  return OCI_SUCCESS;
}

STATICF sword create_env(OraText *user, ub2 user_len,
                         OraText *pwd,  ub2 pwd_len,
                         OCIEnv **envhp, OCISvcCtx **svchp, OCIError **errhp)
{
  sword       status;
  OCIServer  *srvhp;
  OCISession *usrp;
  OCICPool   *poolhp;
  OraText    *poolname;
  ub4         poolnamelen;
  OraText    *database =(OraText *)"";
  OCIBinXmlReposCtx *rctx;
  /* Create and initialize environment. Allocate error handle. */
  . . .
  if ((status = OCIConnectionPoolCreate((dvoid *)*envhp, (dvoid*)*errhp,
                                        (dvoid *)poolhp, &poolname,
                                        (sb4 *)&poolnamelen,
                                        (OraText *)0,
                                        (sb4) 0, 1, 10, 1,
                                        (OraText *)USER,
                                        (sb4) USER_LEN,
                                        (OraText *)PWD,
                                        (sb4) PWD_LEN,
                                        OCI_DEFAULT)) != OCI_SUCCESS)
    {
      printf ("OCIConnectionPoolCreate - Fail %d\n", status);
      return OCI_ERROR;
```

```
    }
  status = OCILogon2((OCIEnv *)*envhp, *errhp, svchp, (OraText *)USER,
                     (ub4)USER_LEN, (const oratext *)PWD, (ub4)PWD_LEN,
                     (const oratext *)poolname, poolnamelen, OCI_CPOOL);
  if (status)
    {
      printf ("OCILogon2 - Fail %d\n", status);
      return OCI_ERROR;
    }
  OCIBinXmlCreateReposCtxFromCPool(*envhp, poolhp, *errhp, &rctx);
  OCIBinXmlSetReposCtxForConn(*svchp, rctx);
  return OCI_SUCCESS;
}

STATICF sword run_example(OCIEnv *envhp, OCISvcCtx *svchp, OCIError *errhp,
                          OCIDuration dur)
{
  OCIType   *xmltdo = (OCIType *)0;
  OCIStmt   *stmthp;
  OCIDefine *defnp;
  xmldocnode *xmldoc = (xmldocnode *)0;
  ub4        xmlsize = 0;
  text      *selstmt = (text *)"SELECT doc FROM po_binxmltab";
  sword      status;
  struct xmlctx *xctx = (xmlctx *) 0;
  ocixmldbparam params[NUM_PARAMS];
  xmlerr xerr = (xmlerr) 0;
  /* Obtain type definition for XMLType. Allocate statement handle.
     Prepare SELECT statement. Define variable for XMLType. Execute statement. */
  . . .
  /* Construct xmlctx for using XML C API */
  params[0].name_ocixmldbparam = XCTXINIT_OCIDUR;
  params[0].value_ocixmldbparam = &dur;
  xctx = OCIXmlDbInitXmlCtx(envhp, svchp, errhp, params, NUM_PARAMS);
  /* Print result to local string */
  XmlSaveDom(xctx, &xerr, (xmlnode *)xmldoc, "stdio", stdout, NULL);
  /* Free instances */
  . . .
}
```

See Also:

- "XMLType Storage Models" on page 1-14
- *Oracle XML Developer's Kit Programmer's Guide*

Using the Oracle XDK Pull Parser with Oracle XML DB

You can use the Oracle XDK pull parser with `XMLType` instances in Oracle XML DB. When you use this parser, parsing is done on demand, so your application drives the parsing process. Your application accesses an XML document through a sequence of events, with start tags, end tags, and comments, just as in Simple API for XML (SAX) parsing. However, unlike the case of SAX parsing, where parsing events are handled by callbacks, in pull parsing your application calls methods to ask for (pull) events only when it needs them. This gives the application more control over XML processing. In particular, filtering is more flexible with the pull parser than with the SAX parser.

You can also use the Oracle XDK pull parser to perform stream-based XML Schema validation.

Example 16–3 shows how to use the Oracle XML DB pull parser with an `XMLType` instance. To use the pull parser, you also need static library `libxml10.a` on UNIX and Linux systems or `oraxml10.dll` on Microsoft Windows systems. You also need header file `xmlev.h`.

See Also:

- *Oracle XML Developer's Kit Programmer's Guide* for information about the Oracle XDK pull parser
- *Oracle XML Developer's Kit Programmer's Guide* for information on using the pull parser for stream-based validation

Example 16–3 Using the Oracle XML DB Pull Parser

```
#define MAXBUFLEN 64*1024
void main()
{
  test_ctx temp_ctx;
  test_ctx *ctx = &temp_ctx;
  OCIType *xmltdo = (OCIType *) 0;
  ocixmldbparam params[1];
  sword status = 0;
  xmlctx *xctx;
  OCIDefine *defnp = (OCIDefine *) 0;
  oratext sel_stmt[] =
    "SELECT XMLSerialize(DOCUMENT x.OBJECT_VALUE AS CLOB) FROM PURCHASEORDER x where rownum = 1";
  OCILobLocator *cob;
  ub4 amtp, nbytes;
  ub1 bufp[MAXBUFLEN];
  ctx->username = (oratext *)"oe";
  ctx->password = (oratext *)"**************";    /* Replace with real password */

  /* Initialize envhp, svchp, errhp, dur, stmthp */
  init_oci_handles(ctx);

  /* Get an xml context */
  params[0].name_ocixmldbparam = XCTXINIT_OCIDUR;
  params[0].value_ocixmldbparam = &ctx->dur;
  xctx = OCIXmlDbInitXmlCtx(ctx->envhp, ctx->svchp, ctx->errhp, params, 1);

  /* Start processing */
  printf("\n\nSupports XML 1.0? : %s\n",
         XmlHasFeature(xctx, (oratext *) "xml", (oratext *) "1.0") ?
         "YES" : "NO");

  /* Allocate the lob descriptor */
  status = OCIDescriptorAlloc((dvoid *) ctx->envhp, (dvoid **) &clob,
                      (ub4)OCI_DTYPE_LOB, (size_t) 0, (dvoid **) 0);
  if (status)
  {
    printf("OCIDescriptorAlloc Failed\n");
    goto error;
  }
  status = OCIStmtPrepare(ctx->stmthp, ctx->errhp,
                (CONST OraText *)sel_stmt, (ub4) strlen((char *)sel_stmt),
                (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT);
  if (status)
  {
    printf("OCIStmtPrepare Failed\n");
    goto error;
```

```
  }
  status = OCIDefineByPos(ctx->stmthp, &defnp, ctx->errhp, (ub4) 1,
                (dvoid *) &clob, (sb4) -1, (ub2 ) SQLT_CLOB,
                (dvoid *) 0, (ub2 *)0,
                (ub2 *)0, (ub4) OCI_DEFAULT);
  if (status)
  {
    printf("OCIDefineByPos Failed\n");
    goto error;
  }
  status = OCIStmtExecute(ctx->svchp, ctx->stmthp, ctx->errhp, (ub4) 1,
                         (ub4) 0, (CONST OCISnapshot*) 0, (OCISnapshot*) 0,
                         (ub4) OCI_DEFAULT);
  if (status)
  {
    printf("OCIStmtExecute Failed\n");
    goto error;
  }
  /* read the fetched value into a buffer */
  amtp = nbytes = MAXBUFLEN-1;
  status = OCILobRead(ctx->svchp, ctx->errhp, clob, &amtp,
               (ub4) 1, (dvoid *) bufp, (ub4) nbytes, (dvoid *)0,
               (sb4 (*)(dvoid *, CONST dvoid *, ub4, ub1)) 0,
               (ub2) 0, (ub1) SQLCS_IMPLICIT);
  if (status)
  {
    printf("OCILobRead Failed\n");
    goto error;
  }
  bufp[amtp] = '\0';
  if (amtp > 0)
  {
     printf("\n=> Query result of %s: \n%s\n", sel_stmt, bufp);
     /********** PULL PARSING *****************/
     status = pp_parse(xctx, bufp, amtp);
     if (status)
       printf("Pull Parsing failed\n");
  }
error:
  /* Free XML Ctx */
  OCIXmlDbFreeXmlCtx(xctx);

  /* Free envhp, svchp, errhp, stmthp */
  free_oci_handles(ctx);
}
#define ERRBUFLEN 256
sb4 pp_parse(xctx, buf, amt)
xmlctx  *xctx;
oratext *buf;
ub4     amt;
{
  xmlevctx *evctx;
  xmlerr   xerr = XMLERR_OK;
  oratext  message[ERRBUFLEN];
  oratext  *emsg = message;
  xmlerr   ecode;
  boolean  done, inattr = FALSE;
  xmlevtype event;

  /* Create an XML event context - Pull Parser Context */
```

```
 evctx = XmlEvCreatePPCtx(xctx, &xerr,
                          "expand_entities", FALSE,
                          "validate", TRUE,
                          "attr_events", TRUE,
                          "raw_buffer_len", 1024,
                          NULL);
if (!evctx)
 {
   printf("FAILED: XmlEvCreatePPCtx: %d\n", xerr);
   return OCI_ERROR;
 }
 /* Load the document from input buffer */
 xerr = XmlEvLoadPPDoc(xctx, evctx, "buffer", buf, amt, "utf-8");
 if (xerr)
 {
   printf("FAILED: XmlEvLoadPPDoc: %d\n", xerr);
   return OCI_ERROR;
 }
 /* Process the events until END_DOCUMENT event or error */
 done = FALSE;
 while(!done)
 {
   event = XmlEvNext(evctx);
   switch(event)
   {
     case XML_EVENT_START_ELEMENT:
       printf("START ELEMENT: %s\n", XmlEvGetName0(evctx));
       break;
     case XML_EVENT_END_ELEMENT:
       printf("END ELEMENT: %s\n", XmlEvGetName0(evctx));
       break;
     case XML_EVENT_START_DOCUMENT:
       printf("START DOCUMENT\n");
       break;
     case XML_EVENT_END_DOCUMENT:
       printf("END DOCUMENT\n");
       done = TRUE;
       break;
     case XML_EVENT_START_ATTR:
       printf("START ATTRIBUTE: %s\n", XmlEvGetAttrName0(evctx, 0));
       inattr = TRUE;
       break;
     case XML_EVENT_END_ATTR:
       printf("END ATTRIBUTE: %s\n", XmlEvGetAttrName0(evctx, 0));
       inattr = FALSE;
       break;
     case XML_EVENT_CHARACTERS:
       if (inattr)
         printf("ATTR VALUE: %s\n", XmlEvGetText0(evctx));
       else
         printf("TEXT: %s\n", XmlEvGetText0(evctx));
       break;
     case XML_EVENT_ERROR:
     case XML_EVENT_FATAL_ERROR:
       done = TRUE;
       ecode = XmlEvGetError(evctx, &emsg);
       printf("ERROR: %d: %s\n", ecode, emsg);
       break;
   }
 }
```

```
  /* Destroy the event context */
  XmlEvDestroyPPCtx(xctx, evctx);
  return OCI_SUCCESS;
}
```

The output from compiling and running this C program is as follows:

```
=> Query result of XMLSerialize(DOCUMENT x.OBJECT_VALUE AS CLOB) FROM PURCHASEORDER x where rownum = 1:
<PurchaseOrder xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xsi:noNamespaceSchemaLocation=
                 "http://localhost:8080/source/schemas/poSource/xsd/purchaseOrder.xsd">
  <Reference>AMCEWEN-20021009123336171PDT</Reference>
  <Actions>
    <Action>
      <User>KPARTNER</User>
    </Action>
  </Actions>
  <Reject/>
  <Requestor>Allan D. McEwen</Requestor>
  <User>AMCEWEN</User>
  <CostCenter>S30</CostCenter>
  <ShippingInstructions>
    <name>Allan D. McEwen</name>
    <address>Oracle Plaza
Twin Dolphin Drive
Redwood Shores
CA
94065
USA</address>
    <telephone>650 506 7700</telephone>
  </ShippingInstructions>
  <SpecialInstructions>Ground</SpecialInstructions>
  <LineItems>
    <LineItem ItemNumber="1">
      <Description>Salesman</Description>
      <Part Id="37429158920" UnitPrice="39.95" Quantity="2"/>
    </LineItem>
    . . .
  </LineItems>
</PurchaseOrder>

START DOCUMENT
START ELEMENT: PurchaseOrder
START ATTRIBUTE: xmlns:xsi
ATTR VALUE: http://www.w3.org/2001/XMLSchema-instance
END ATTRIBUTE: xmlns:xsi
START ATTRIBUTE: xsi:noNamespaceSchemaLocation
ATTR VALUE: http://localhost:8080/source/schemas/poSource/xsd/purchaseOrder.xsd
END ATTRIBUTE: xsi:noNamespaceSchemaLocation
START ELEMENT: Reference
TEXT: AMCEWEN-20021009123336171PDT
END ELEMENT: Reference
START ELEMENT: Actions
START ELEMENT: Action
START ELEMENT: User
TEXT: KPARTNER
END ELEMENT: User
END ELEMENT: Action
END ELEMENT: Actions
START ELEMENT: Reject
END ELEMENT: Reject
START ELEMENT: Requestor
```

```
TEXT: Allan D. McEwen
END ELEMENT: Requestor
START ELEMENT: User
TEXT: AMCEWEN
END ELEMENT: User
START ELEMENT: CostCenter
TEXT: S30
END ELEMENT: CostCenter
START ELEMENT: ShippingInstructions
START ELEMENT: name
TEXT: Allan D. McEwen
END ELEMENT: name
START ELEMENT: address
TEXT: Oracle Plaza
Twin Dolphin Drive
Redwood Shores
CA
94065
USA
END ELEMENT: address
START ELEMENT: telephone
TEXT: 650 506 7700
END ELEMENT: telephone
END ELEMENT: ShippingInstructions
START ELEMENT: SpecialInstructions
TEXT: Ground
END ELEMENT: SpecialInstructions
START ELEMENT: LineItems
START ELEMENT: LineItem
START ATTRIBUTE: ItemNumber
ATTR VALUE: 1
END ATTRIBUTE: ItemNumber
START ELEMENT: Description
TEXT: Salesman
END ELEMENT: Description
START ELEMENT: Part
START ATTRIBUTE: Id
ATTR VALUE: 37429158920
END ATTRIBUTE: Id
START ATTRIBUTE: UnitPrice
ATTR VALUE: 39.95
END ATTRIBUTE: UnitPrice
START ATTRIBUTE: Quantity
ATTR VALUE: 2
END ATTRIBUTE: Quantity
END ELEMENT: Part
END ELEMENT: LineItem
. . .
END ELEMENT: LineItems
END ELEMENT: PurchaseOrder
END DOCUMENT
```

Common XMLType Operations in C

Table 16–2 provides the `XMLType` functional equivalent of common XML operations.

Table 16–2 Common XMLType Operations in C

Description	C API XMLType Function
Create empty `XMLType` instance	`XmlCreateDocument()`
Create from a source buffer	`XmlLoadDom()`
Extract an XPath expression	`XmlXPathEvalexpr()` and family
Transform using an XSLT style sheet	`XmlXslProcess()` and family
Check if an XPath exists	`XmlXPathEvalexpr()` and family
Is document schema-based?	`XmlDomIsSchemaBased()`
Get schema information	`XmlDomGetSchema()`
Get document namespace	`XmlDomGetNodeURI()`
Validate using schema	`XmlSchemaValidate()`
Obtain DOM from XMLType	Cast `(void *)` to `(xmldocnode *)`
Obtain XMLType from DOM	Cast `(xmldocnode *)` to `(void *)`

See Also: *Oracle XML Developer's Kit Programmer's Guide* "XML Parser for C"

Example 16–4 shows how to use the DOM to determine how many instances of a particular part have been ordered. The part in question has `Id` `37429158722`. See Appendix A, "Oracle-Supplied XML Schemas and Examples", Example A–5 on page A-47 for the definitions of helper functions `exec_bind_xml`, `free_oci_handles`, and `init_oci_handles`.

Example 16–4 Using the DOM to Count Ordered Parts

```
#ifndef S_ORACLE
#include <s.h>
#endif
#ifndef ORATYPES_ORACLE
#include <oratypes.h>
#endif
#ifndef XML_ORACLE
#include <xml.h>
#endif
#ifndef OCIXML_ORACLE
#include <ocixml.h>
#endif
#ifndef OCI_ORACLE
#include <oci.h>
#endif
#include <string.h>

typedef struct test_ctx {
        OCIEnv *envhp;
        OCIError *errhp;
        OCISvcCtx *svchp;
        OCIStmt *stmthp;
        OCIServer *srvhp;
        OCIDuration dur;
        OCISession *sesshp;
        oratext *username;
        oratext *password;
```

```
} test_ctx;

/* Helper function 1: execute a sql statement which binds xml data */
STATICF sword exec_bind_xml(OCISvcCtx *svchp,
                    OCIError *errhp,
                    OCIStmt *stmthp,
                    void *xml,
                    OCIType *xmltdo,
                    OraText *sqlstmt);

/* Helper function 2: Initialize OCI handles and connect */
STATICF sword init_oci_handles(test_ctx *ctx);

/* Helper function 3: Free OCI handles and disconnect */
STATICF sword free_oci_handles(test_ctx *ctx);

void main()
{
  test_ctx temp_ctx;
  test_ctx *ctx = &temp_ctx;
  OCIType *xmltdo = (OCIType *) 0;
  xmldocnode *doc = (xmldocnode *)0;
  ocixmldbparam params[1];
  xmlnode *quux, *foo, *foo_data, *top;
  xmlerr err;
  sword status = 0;
  xmlctx *xctx;
  ub4 xmlsize = 0;
  OCIDefine *defnp = (OCIDefine *) 0;
  oratext sel_stmt[] = "SELECT SYS_NC_ROWINFO$ FROM PURCHASEORDER";
  xmlnodelist *litems = (xmlnodelist *)0;
  xmlnode *item = (xmlnode *)item;
  xmlnode *part;
  xmlnamedmap *attrs;
  xmlnode *id;
  xmlnode *qty;
  oratext *idval;
  oratext *qtyval;
  ub4 total_qty;
  int i;
  int numdocs;

  ctx->username = (oratext *)"oe";
  ctx->password = (oratext *)"************";   /* Replace with real password */

  /* Initialize envhp, svchp, errhp, dur, stmthp */
  init_oci_handles(ctx);

  /* Get an xml context */
  params[0].name_ocixmldbparam = XCTXINIT_OCIDUR;
  params[0].value_ocixmldbparam = &ctx->dur;
  xctx = OCIXmlDbInitXmlCtx(ctx->envhp, ctx->svchp, ctx->errhp, params, 1);

  /* Start processing */
  printf("\n\nSupports XML 1.0? : %s\n",
         XmlHasFeature(xctx, (oratext *) "xml", (oratext *) "1.0") ?
         "YES" : "NO");

  /* Get the documents from the database using a select statement */
  status = OCITypeByName(ctx->envhp, ctx->errhp, ctx->svchp, (const text *) "SYS",
```

```
                        (ub4) strlen((char *)"SYS"), (const text *) "XMLTYPE",
                        (ub4) strlen((char *)"XMLTYPE"), (CONST text *) 0,
                        (ub4) 0, OCI_DURATION_SESSION, OCI_TYPEGET_HEADER,
                        (OCIType **) &xmltdo);
  status = OCIStmtPrepare(ctx->stmthp, ctx->errhp,
                 (CONST OraText *)sel_stmt, (ub4) strlen((char *)sel_stmt),
                 (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT);
  status = OCIDefineByPos(ctx->stmthp, &defnp, ctx->errhp, (ub4) 1, (dvoid *) 0,
                 (sb4) 0, SQLT_NTY, (dvoid *) 0, (ub2 *)0,
                 (ub2 *)0, (ub4) OCI_DEFAULT);
  status = OCIDefineObject(defnp, ctx->errhp, (OCIType *) xmltdo,
                  (dvoid **) &doc,
                  &xmlsize, (dvoid **) 0, (ub4 *) 0);
  status = OCIStmtExecute(ctx->svchp, ctx->stmthp, ctx->errhp, (ub4) 0, (ub4) 0,
                 (CONST OCISnapshot*) 0, (OCISnapshot*) 0, (ub4) OCI_DEFAULT);

  /* Initialize variables */
  total_qty = 0;
  numdocs = 0;

  /* Loop through all the documents */
  while ((status = OCIStmtFetch2(ctx->stmthp, ctx->errhp, (ub4) 1, (ub4) OCI_
FETCH_NEXT,
                                (ub4)1, (ub4) OCI_DEFAULT)) == 0)
  {
    numdocs++;

    /* Get all the LineItem elements */
    litems = XmlDomGetDocElemsByTag(xctx, doc, (oratext *)"LineItem");
    i = 0;

    /* Loop through all LineItems */
    while (item = XmlDomGetNodeListItem(xctx, litems, i))
    {
      /* Get the part */
      part = XmlDomGetLastChild(xctx, item);

      /* Get the attributes */
      attrs = XmlDomGetAttrs(xctx, (xmlelemnode *)part);

      /* Get the id attribute and its value */
      id = XmlDomGetNamedItem(xctx, attrs, (oratext *)"Id");
      idval = XmlDomGetNodeValue(xctx, id);

      /* Keep only parts with id 37429158722 */
      if (idval && (strlen((char *)idval) == 11 )
          && !strncmp((char *)idval, (char *)"37429158722", 11))
      {
        /* Get the quantity attribute and its value.*/
        qty = XmlDomGetNamedItem(xctx, attrs, (oratext *)"Quantity");
        qtyval = XmlDomGetNodeValue(xctx, qty);

        /* Add the quantity to total_qty */
        total_qty += atoi((char *)qtyval);
      }
      i++;
    }
    XmlFreeDocument(xctx, doc);
    doc = (xmldocnode *)0;
  }
```

```
  printf("Total quantity needed for part 37429158722 = %d\n", total_qty);
  printf("Number of documents in table PURCHASEORDER = %d\n", numdocs);

  /* Free Xml Ctx */
  OCIXmlDbFreeXmlCtx(xctx);

  /* Free envhp, svchp, errhp, stmthp */
  free_oci_handles(ctx);
}
```

The output from compiling and running this C program is as follows:

```
Supports XML 1.0? : YES
Total quantity needed for part 37429158722 = 42
Number of documents in table PURCHASEORDER = 132
```

17

Using Oracle Data Provider for .NET with Oracle XML DB

Oracle Data Provider for Microsoft .NET (ODP.NET) is an implementation of a data provider for Oracle Database. It uses Oracle native APIs to offer fast and reliable access to Oracle data and features from any .NET application. It also uses and inherits classes and interfaces available in the Microsoft .NET Framework Class Library. ODP.NET supports the following LOB data types natively with .NET: `BLOB`, `CLOB`, `NCLOB`, and `BFILE`.

This chapter describes how to use ODP.NET with Oracle XML DB. It contains these topics:

- ODP.NET XML Support and Oracle XML DB
- ODP.NET Sample Code

ODP.NET XML Support and Oracle XML DB

ODP.NET supports XML natively in the database, through Oracle XML DB. ODP.NET XML support includes the following features:

- Stores XML data natively in Oracle Database as `XMLType`.
- Accesses relational and object-relational data as XML data from Oracle Database to a Microsoft .NET environment, and processes the XML using Microsoft .NET framework.
- Saves changes to the database server using XML data.

For the .NET application developer, these features include the following:

- Enhancements to the `OracleCommand`, `OracleConnection`, and `OracleDataReader` classes. Provides the following XML-specific classes:
  - `OracleXmlType`
  - `OracleXmlStream`
  - `OracleXmlQueryProperties`
  - `OracleXmlSaveProperties`

ODP.NET Sample Code

Example 17–1 retrieves `XMLType` data from the database to .NET and outputs the results:

Example 17–1 Retrieve XMLType Data to .NET

```
//Create OracleCommand and query XMLType
OracleCommand xmlCmd = new OracleCommand();
poCmd.CommandText = "SELECT po FROM po_tab";
poCmd.Connection = conn;
// Execute OracleCommand and output XML results to an OracleDataReader
OracleDataReader poReader = poCmd.ExecuteReader();
// ODP.NET native XML data type object from Oracle XML DB
OracleXmlType poXml;
string str = ""; //read XML results
while (poReader.Read())
{
  // Return OracleXmlType object of the specified XmlType column
  poXml = poReader.GetOracleXmlType(0);
  // Concatenate output for all the records
  str = str + poXml.Value;
} //Output XML results to the screen
Console.WriteLine(str);
```

See Also: *Oracle Data Provider for .NET Developer's Guide* for complete information about Oracle .NET support for Oracle XML DB.

Part IV

Viewing Existing Data as XML

Part IV of this manual introduces you to ways you can view your existing data as XML. It contains the following chapters:

- Chapter 18, "Generating XML Data from the Database"
- Chapter 19, "XMLType Views"
- Chapter 20, "Accessing Data Through URIs"

18

Generating XML Data from the Database

This chapter describes Oracle XML DB features for generating (constructing) XML data from relational data in the database. It describes the SQL/XML standard functions and Oracle Database-provided functions and packages for generating XML data from relational content.

This chapter contains these topics:

- Overview of Generating XML Data From Oracle Database
- Generating XML using SQL Functions
- Generating XML using DBMS_XMLGEN
- Generating XML using SQL Function SYS_XMLGEN
- Generating XML using SQL Function SYS_XMLAGG
- Generating XML using XSQL Pages Publishing Framework
- Generating XML using XML SQL Utility (XSU)
- Guidelines for Generating XML with Oracle XML DB

> **See Also:** Chapter 5, "Using XQuery with Oracle XML DB" for information about constructing XML data using SQL/XML functions `XMLQuery` and `XMLTable`

Overview of Generating XML Data From Oracle Database

This section provides an overview of the various ways you can generate XML data with Oracle Database.

Overview of Generating XML using SQL/XML Functions

You can generate XML data using any of the following standard SQL/XML functions supported by Oracle XML DB. This is described in "Generating XML using SQL Functions" on page 18-2.

Overview of Generating XML using Oracle SQL Functions

You can generate XML data using any of the following Oracle SQL functions:

- XMLROOT SQL Function on page 18-19
- XMLCOLATTVAL SQL Function on page 18-20
- XMLCDATA SQL Function on page 18-21

- Generating XML using SQL Function SYS_XMLGEN on page 18-46. This operates on rows, generating XML documents.
- Generating XML using SQL Function SYS_XMLAGG on page 18-54. This operates on groups of rows, aggregating several XML documents into one.

Overview of Generating XML using DBMS_XMLGEN

You can generate XML from SQL queries using PL/SQL package `DBMS_XMLGEN`. This is described in "Generating XML using DBMS_XMLGEN" on page 18-22.

Overview of Generating XML using XSQL Pages Publishing Framework

You can generate XML using XSQL Pages Publishing Framework, also known as XSQL Servlet. This is described in Generating XML using XSQL Pages Publishing Framework on page 18-54. XSQL Pages Publishing Framework is part of Oracle XML Developer's Kit for Java.

Overview of Generating XML using XML SQL Utility (XSU)

You can use XML SQL Utility (XSU) to perform the following tasks on data in `XMLType` tables and columns:

- Transform data retrieved from object-relational database tables or views into XML.
- Extract data from an XML document, and using a canonical mapping, insert the data into appropriate columns or attributes of a table or a view.
- Extract data from an XML document and apply this data to updating or deleting values of the appropriate columns or attributes.

See Also:

- "Generating XML using XML SQL Utility (XSU)" on page 18-57
- Chapter 3, "Using Oracle XML DB"
- Chapter 11, "Transforming and Validating XMLType Data"
- Chapter 13, "PL/SQL APIs for XMLType"
- Chapter 15, "Java DOM API for XMLType"

Overview of Generating XML using DBURIType

You can use a `DBURIType` instance to construct XML documents that contain database data and whose structure reflects the database structure. This is described in Chapter 20, "Accessing Data Through URIs".

Generating XML using SQL Functions

This section describes SQL functions that you can use to construct XML data. Most of these functions belong to the SQL/XML standard.

- XMLELEMENT and XMLATTRIBUTES SQL/XML Functions on page 18-3
- XMLFOREST SQL/XML Function on page 18-10
- XMLCONCAT SQL/XML Function on page 18-11
- XMLAGG SQL/XML Function on page 18-12

- XMLPI SQL/XML Function on page 18-15
- XMLCOMMENT SQL/XML Function on page 18-16
- XMLSERIALIZE SQL/XML Function on page 18-17
- XMLPARSE SQL/XML Function on page 18-18

The standard XML-generation functions are also known as **SQL/XML publishing** functions.

You can also construct XML data using the SQL/XML function `XMLQuery`. The use of `XMLQuery` is not limited to publishing XML data. It is very general and is referred to in this book as a SQL/XML *query and access* function.

This standardization process is ongoing. Please refer to `http://www.sqlx.org` for the latest information about SQL/XML functions, including `XMLQuery` and `XMLTable`.

Other XML-generating SQL functions presented in this section are Oracle-specific (not part of the SQL/XML standard):

- XMLROOT SQL Function on page 18-19
- XMLCOLATTVAL SQL Function on page 18-20
- XMLCDATA SQL Function on page 18-21
- Generating XML using SQL Function SYS_XMLGEN on page 18-46. This operates on relational rows, generating XML documents.
- Generating XML using SQL Function SYS_XMLAGG on page 18-54. This operates on groups of relational rows, aggregating several XML documents into one.

All of the XML-generation SQL functions convert scalars and user-defined data-type instances to their canonical XML format. In this canonical mapping, user-defined data-type attributes are mapped to XML elements.

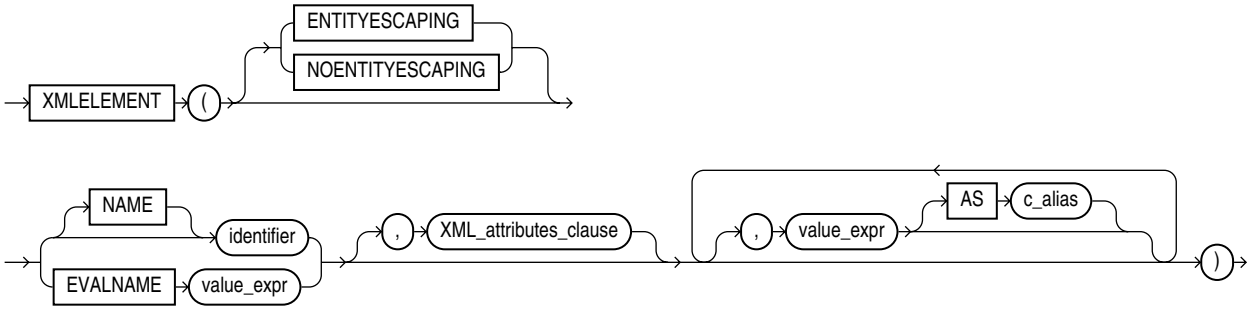
See Also:

- Chapter 5, "Using XQuery with Oracle XML DB" for information about constructing XML data using SQL/XML function `XMLQuery`
- *Oracle Database SQL Language Reference* for information about Oracle support for the SQL/XML standard

XMLELEMENT and XMLATTRIBUTES SQL/XML Functions

You use SQL/XML standard function `XMLElement` to construct XML elements from relational data. It takes as arguments an element name, an optional collection of attributes for the element, and zero or more additional arguments that make up the element content. It returns an `XMLType` instance.

Figure 18–1 XMLELEMENT Syntax



For an explanation of keywords `ENTITYESCAPING` and `NOENTITYESCAPING`, see "Escaping Characters in Generated XML Data" on page 18-5.

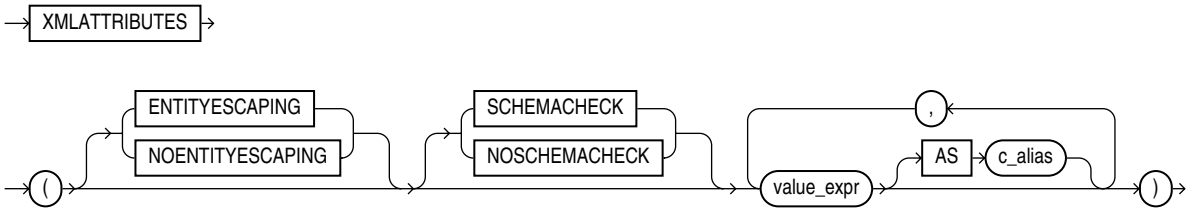
The first argument to function `XMLElement` defines an identifier that names the *root* XML element to be created. The root-element identifier argument can be defined using a literal identifier (*`identifier`*, in Figure 18–1) or by `EVALNAME` followed by an expression (*`value_expr`*) that evaluates to an identifier. However it is defined, the identifier must not be `NULL` or else an error is raised.

The optional *`XML-attributes-clause`* argument of function `XMLElement` specifies the attributes of the root element to be generated. Figure 18–2 shows the syntax of this argument.

In addition to the optional *`XML-attributes-clause`* argument, function `XMLElement` accepts zero or more *`value_expr`* arguments that make up the *content* of the root element (child elements and text content). If an *`XML-attributes-clause`* argument is also present, these content arguments must follow the *`XML-attributes-clause`* argument. Each of the content-argument expressions is evaluated, and the result is converted to XML format. If a value argument evaluates to `NULL`, then no content is created for that argument.

The optional *`XML-attributes-clause`* argument uses SQL/XML standard function `XMLAttributes` to specify the *attributes* of the root element. Function `XMLAttributes` can be used *only* in a call to function `XMLElement`. It cannot be used on its own.

Figure 18–2 XMLAttributes Clause Syntax (XMLATTRIBUTES)



For an explanation of keywords `ENTITYESCAPING` and `NOENTITYESCAPING`, see "Escaping Characters in Generated XML Data" on page 18-5.

Keywords `SCHEMACHECK` and `NOSCHEMACHECK` determine whether or not a run-time check is made of the generated attributes, to see if any of them specify a schema location that corresponds to an XML schema that is registered with Oracle XML DB, and, if so, to try to generate XML schema-based XML data accordingly. The default behavior is that provided by `NOSCHEMACHECK`: no check is made. In releases prior to

11*g* Release 2 (11.2), the default behavior is to perform the check. Keyword `SCHEMACHECK` can be used to obtain backward compatibility.

Note that a similar check is *always* made at *compile* time, regardless of the presence or absence of `NOSCHEMACHECK`. This means, in particular, that if you use a string literal to specify an XML schema location attribute value, then a (compile-time) check is made, and, if appropriate, XML schema-based data is generated accordingly.

Note: If a view is created to generate XML data, function `XMLAttributes` is used to add XML-schema location references, and the target XML schema has not yet been registered with Oracle XML DB, then the XML data generated is not XML schema-based. If the XML schema is subsequently registered, then XML data generated thereafter is also *not* XML-schema-based. To create XML schema-based data, you must recompile the view.

Argument *`XML-attributes-clause`* itself contains one or more *`value_expr`* expressions as arguments to function `XMLAttributes`. These are evaluated to obtain the values for the attributes of the root element. (Do not confuse these *`value_expr`* arguments to function `XMLAttributes` with the *`value_expr`* arguments to function `XMLElement`, which specify the content of the root element.) The optional `AS` *`c_alias`* clause for each *`value_expr`* specifies that the attribute name is *`c_alias`*, which can be either a string literal or `EVALNAME` followed by an expression that evaluates to a string literal.

If an attribute value expression evaluates to `NULL`, then no corresponding attribute is created. The data type of an attribute value expression cannot be an object type or a collection.

Escaping Characters in Generated XML Data

As specified by the SQL/XML standard, characters in explicit *identifiers* are *not* escaped in any way – it is up to you to ensure that valid XML names are used. This applies to all SQL/XML functions. In particular, it applies to the root-element identifier of `XMLElement` (*`identifier`*, in Figure 18–1) and to attribute identifier aliases named with `AS` clauses of `XMLAttributes` (see Figure 18–2).

However, other XML data that is generated is *escaped*, by default, to ensure that only valid XML `NameChar` characters are generated. As part of generating a valid XML element or attribute name from a SQL identifier, each character that is disallowed in an XML name is replaced with an underscore character (_), followed by the hexadecimal Unicode representation of the original character, followed by a second underscore character. For example, the colon character (`:`) is escaped by replacing it with `_003A_`, where 003A is the hexadecimal Unicode representation.

Escaping applies to characters in the evaluated *`value_expr`* arguments to *all* SQL/XML functions, including `XMLElement` and `XMLAttributes`. It applies also to the characters of an attribute identifier that is defined implicitly from an `XMLAttributes` attribute value expression that is *not* followed by an `AS` clause: the escaped form of the SQL column name is used as the name of the attribute.

In some cases, you might not need or want character escaping. If you know, for example, that the XML data being generated is well-formed, then you can save some processing time by inhibiting escaping. You can do that by specifying the keyword `NOENTITYESCAPING` for SQL/XML functions `XMLElement` and `XMLAttributes`. Keyword `ENTITYESCAPING` imposes escaping, which is the default behavior.

Formatting of XML Dates and Timestamps

The XML Schema standard specifies that dates and timestamps in XML data be in standard formats. XML generation functions in Oracle XML DB produce XML dates and timestamps according to this standard.

In releases prior to Oracle Database 10g Release 2, the database settings for date and timestamp formats, not the XML Schema standard formats, were used for XML. You can reproduce this *previous* behavior by setting the database event 19119, level 0x8, as follows:

```
ALTER SESSION SET EVENTS '19119 TRACE NAME CONTEXT FOREVER, LEVEL 0x8';
```

If you must otherwise produce a nonstandard XML date or timestamp, use SQL function `to_char` – see Example 18–1.

> **See Also:**
>
> `http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/datatypes.html#isoformats` for the XML Schema specification of XML date and timestamp formats

XMLElement Examples

This section provides examples that use SQL/XML function `XMLElement`.

Example 18–1 uses `XMLElement` to generate an XML date with a format that is different from the XML Schema standard date format.

Example 18–1 XMLELEMENT: Formatting a Date

```
-- With standard XML date format:
SELECT XMLElement("Date", hire_date)
  FROM hr.employees
  WHERE employee_id = 203;

XMLELEMENT("DATE",HIRE_DATE)
----------------------------
<Date>2002-06-07</Date>

1 row selected.

-- With an alternative date format:
SELECT XMLElement("Date", to_char(hire_date))
  FROM hr.employees
  WHERE employee_id = 203;

XMLELEMENT("DATE",TO_CHAR(HIRE_DATE))
-------------------------------------
<Date>07-JUN-02</Date>

1 row selected.
```

Example 18–2 uses `XMLElement` to generate an `Emp` element for each employee, with the employee name as the content.

Example 18–2 XMLELEMENT: Generating an Element for Each Employee

```
SELECT e.employee_id,
       XMLELEMENT ("Emp", e.first_name ||' '|| e.last_name) AS "RESULT"
   FROM hr.employees e
   WHERE employee_id > 200;
```

This query produces the following typical result:

```
EMPLOYEE_ID RESULT
----------- ----------------------------------
        201 <Emp>Michael Hartstein</Emp>
        202 <Emp>Pat Fay</Emp>
        203 <Emp>Susan Mavris</Emp>
        204 <Emp>Hermann Baer</Emp>
        205 <Emp>Shelley Higgins</Emp>
        206 <Emp>William Gietz</Emp>

6 rows selected.
```

SQL/XML function `XMLElement` can also be nested, to produce XML data with a nested structure.

Example 18–3 uses `XMLElement` to generate an `Emp` element for each employee, with child elements that provide the employee name and hire date.

Example 18–3 XMLELEMENT: Generating Nested XML

```
SELECT XMLElement("Emp",
                  XMLElement("name", e.first_name ||' '|| e.last_name),
                  XMLElement("hiredate", e.hire_date)) AS "RESULT"
FROM hr.employees e
WHERE employee_id > 200;
```

This query produces the following typical XML result:

```
RESULT
----------------------------------------------------------------------
<Emp><name>Michael Hartstein</name><hiredate>2004-02-17</hiredate></Emp>
<Emp><name>Pat Fay</name><hiredate>2005-08-17</hiredate></Emp>
<Emp><name>Susan Mavris</name><hiredate>2002-06-07</hiredate></Emp>
<Emp><name>Hermann Baer</name><hiredate>2002-06-07</hiredate></Emp>
<Emp><name>Shelley Higgins</name><hiredate>2002-06-07</hiredate></Emp>
<Emp><name>William Gietz</name><hiredate>2002-06-07</hiredate></Emp>

6 rows selected.
```

Example 18–4 uses `XMLElement` to generate an `Emp` element for each employee, with attributes `id` and `name`.

Example 18–4 XMLELEMENT: Generating Employee Elements with Attributes ID and Name

```
SELECT XMLElement("Emp", XMLAttributes(
                          e.employee_id as "ID",
                          e.first_name ||' ' || e.last_name AS "name"))
  AS "RESULT"
  FROM hr.employees e
  WHERE employee_id > 200;
```

This query produces the following typical XML result fragment:

```
RESULT
-----------------------------------------------
<Emp ID="201" name="Michael Hartstein"></Emp>
<Emp ID="202" name="Pat Fay"></Emp>
<Emp ID="203" name="Susan Mavris"></Emp>
<Emp ID="204" name="Hermann Baer"></Emp>
```

```
<Emp ID="205" name="Shelley Higgins"></Emp>
<Emp ID="206" name="William Gietz"></Emp>

6 rows selected.
```

As mentioned in "Escaping Characters in Generated XML Data" on page 18-5, characters in the root-element name and the names of any attributes defined by `AS` clauses are *not* escaped. Characters in an identifier name are escaped only if the name is created from an evaluated expression (such as a column reference).

Example 18–5 shows that, with XML data constructed using `XMLElement`, the root-element name and the attribute name are *not* escaped. Invalid XML is produced because greater-than sign (>) and a comma (`,`) are not allowed in XML element and attribute names.

Example 18–5 XMLELEMENT: Characters in Generated XML Are Not Escaped

```
SELECT XMLElement("Emp->Special",
                  XMLAttributes(e.last_name || ', ' || e.first_name
                                AS "Last,First"))
   AS "RESULT"
   FROM hr.employees e
   WHERE employee_id = 201;
```

This query produces the following result, which is *not* well-formed XML:

```
RESULT
--------------------------------------------------------------------
<Emp->Special Last,First="Hartstein, Michael"></Emp->Special>

1 row selected.
```

A full description of character escaping is included in the SQL/XML standard.

Example 18–6 illustrates the use of namespaces to create an XML schema-based document. Assuming that an XML schema "`http://www.oracle.com/Employee.xsd`" exists and has no target namespace, the query in Example 18–6 creates an `XMLType` instance conforming to that schema:

Example 18–6 Creating a Schema-Based XML Document using XMLELEMENT with Namespaces

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

This creates the following XML document that conforms to XML schema `Employee.xsd`. (The result is shown here pretty-printed, for clarity.)

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

```
</Employee>

1 row selected.
```

Example 18–7 uses `XMLElement` to generate an XML document with employee and department information, using data from sample database schema table `hr.departments`.

Example 18–7 XMLELEMENT: Generating an Element from a User-Defined Data-Type Instance

```
CREATE OR REPLACE TYPE emp_t AS OBJECT ("@EMPNO" NUMBER(4),
                                        ENAME VARCHAR2(10));

CREATE OR REPLACE TYPE emplist_t AS TABLE OF emp_t;

CREATE OR REPLACE TYPE dept_t AS OBJECT ("@DEPTNO" NUMBER(2),
                                         DNAME VARCHAR2(14),
                                         EMP_LIST emplist_t);

SELECT XMLElement("Department",
                  dept_t(department_id,
                         department_name,
                         cast(MULTISET
                              (SELECT employee_id, last_name
                                 FROM hr.employees e
                                 WHERE e.department_id = d.department_id)
                              AS emplist_t)))
  AS deptxml
  FROM hr.departments d
  WHERE d.department_id = 10;
```

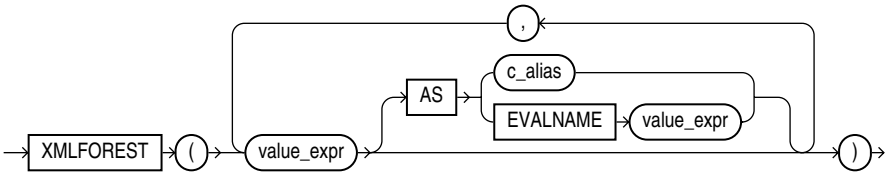
This produces an XML document which contains the `Department` element and the canonical mapping of type `dept_t`.

```
DEPTXML
-------------
<Department>
  <DEPT_T DEPTNO="10">
    <DNAME>ACCOUNTING</DNAME>
    <EMPLIST>
      <EMP_T EMPNO="7782">
        <ENAME>CLARK</ENAME>
      </EMP_T>
      <EMP_T EMPNO="7839">
        <ENAME>KING</ENAME>
      </EMP_T>
      <EMP_T EMPNO="7934">
        <ENAME>MILLER</ENAME>
      </EMP_T>
    </EMPLIST>
  </DEPT_T>
</Department>

1 row selected.
```

XMLFOREST SQL/XML Function

You use SQL/XML standard function `XMLForest` to construct a forest of XML elements. Its arguments are expressions to be evaluated, with optional aliases. Figure 18–3 describes the `XMLForest` syntax.

Figure 18–3 XMLFOREST Syntax



Each of the value expressions (*value_expr* in Figure 18–3) is converted to XML format, and, optionally, identifier *c_alias* is used as the attribute identifier (*c_alias* can be a string literal or `EVALNAME` followed by an expression that evaluates to a string literal).

For an object type or collection, the `AS` clause is required. For other types, the `AS` clause is optional. For a given expression, if the `AS` clause is omitted, then characters in the evaluated value expression are *escaped* to form the name of the enclosing tag of the element. The escaping is as defined in "Escaping Characters in Generated XML Data" on page 18-5. If the value expression evaluates to `NULL`, then no element is created for that expression.

Example 18–8 uses `XMLElement` and `XMLForest` to generate an `Emp` element for each employee, with a `name` attribute and with child elements containing the employee hire date and department as the content.

Example 18–8 XMLFOREST: Generating Elements with Attribute and Child Elements

```
SELECT XMLElement("Emp",
                  XMLAttributes(e.first_name ||' '|| e.last_name AS "name"),
                  XMLForest(e.hire_date, e.department AS "department"))
AS "RESULT"
FROM employees e WHERE e.department_id = 20;
```

(The `WHERE` clause is used here to keep the example brief.) This query produces the following XML result:

```
RESULT
-------------------------------------
<Emp name="Michael Hartstein">
  <HIRE_DATE>2004-02-17</HIRE_DATE>
  <department>20</department>
</Emp>
<Emp name="Pat Fay">
  <HIRE_DATE>2005-08-17</HIRE_DATE>
  <department>20</department>
</Emp>

2 rows selected.
```

See Also: Example 18–20, "XMLCOLATTVAL: Generating Elements with Attribute and Child Elements"

Example 18–9 uses `XMLForest` to generate hierarchical XML data from user-defined data-type instances.

Example 18–9 XMLFOREST: Generating an Element from a User-Defined Data-Type Instance

```
SELECT XMLForest(
  dept_t(department_id,
         department_name,
         cast(MULTISET
              (SELECT employee_id, last_name
                 FROM hr.employees e WHERE e.department_id = d.department_id)
              AS emplist_t))
         AS "Department")
  AS deptxml
  FROM hr.departments d
  WHERE department_id=10;
```

This produces an XML document with element `Department` containing attribute `DEPTNO` and child element `DNAME`.

```
DEPTXML
--------------------------------
<Department DEPTNO="10">
  <DNAME>Administration</DNAME>
    <EMP_LIST>
      <EMP_T EMPNO="200">
        <ENAME>Whalen</ENAME>
      </EMP_T>
    </EMP_LIST>
</Department>

1 row selected.
```

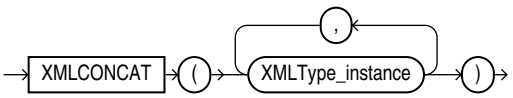
You may want to compare this example with Example 18–7 and Example 18–25.

XMLCONCAT SQL/XML Function

You use SQL/XML standard function `XMLConcat` to construct an XML fragment by concatenating multiple `XMLType` instances. Figure 18–4 shows the `XMLConcat` syntax. Function `XMLConcat` has two forms:

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

Figure 18–4 XMLCONCAT Syntax



Example 18–10 uses SQL/XML function `XMLConcat` to return a concatenation of `XMLType` instances from an `XMLSequenceType` value (a varray of `XMLType` instances).

Example 18–10 XMLCONCAT: Concatenating XMLType Instances from a Sequence

```
SELECT XMLSerialize(
         CONTENT
         XMLConcat(XMLSequenceType(
                   XMLType('<PartNo>1236</PartNo>'),
                   XMLType('<PartName>Widget</PartName>'),
                   XMLType('<PartPrice>29.99</PartPrice>')))
         AS CLOB)
  AS "RESULT"
  FROM DUAL;
```

This query returns a single XML fragment. (The result is shown here pretty-printed, for clarity.)

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

1 row selected.
```

Example 18–11 uses `XMLConcat` to create and concatenate XML elements for employee first and the last names.

Example 18–11 XMLCONCAT: Concatenating XML Elements

```
SELECT XMLConcat(XMLElement("first", e.first_name),
                 XMLElement("last", e.last_name))
  AS "RESULT"
  FROM employees e;
```

This query produces the following XML fragment:

```
RESULT
---------------------------------------------
<first>Den</first><last>Raphaely</last>
<first>Alexander</first><last>Khoo</last>
<first>Shelli</first><last>Baida</last>
<first>Sigal</first><last>Tobias</last>
<first>Guy</first><last>Himuro</last>
<first>Karen</first><last>Colmenares</last>

6 rows selected.
```

XMLAGG SQL/XML Function

You use SQL/XML standard function `XMLAgg` to construct a forest of XML elements from a collection of XML elements—it is an aggregate function.

Figure 18–5 XMLAGG Syntax

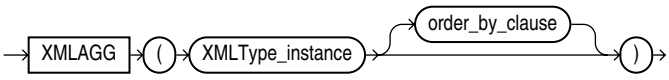


Figure 18–5 describes the XMLAgg syntax, where the order_by_clause is the following:

```
ORDER BY [list of: expr [ASC|DESC] [NULLS {FIRST|LAST}]]
```

Numeric literals are *not* interpreted as column positions. For example, ORDER BY 1 does not mean order by the first column. Instead, numeric literals are interpreted as any other literals.

As with SQL/XML function XMLConcat, any arguments whose value is NULL are dropped from the result. Function XMLAgg is similar to function sys_XMLAgg, except that it returns a forest of nodes and does not accept the XMLFormat parameter.

Function XMLAgg can be used to concatenate XMLType instances across *multiple rows*. It also accepts an optional ORDER BY clause, to order the XML values being aggregated. Function XMLAgg produces one aggregated XML result for each group. If there is no group by specified in the query, then it returns a single aggregated XML result for all the rows of the query.

Example 18–12 uses SQL/XML functions XMLAgg and XMLElement to construct a Department element that contains Employee elements that have employee job ID and last name as their contents. It also orders the Employee elements in the department by employee last name. (The result is shown pretty-printed, for clarity.)

Example 18–12 XMLAGG: Generating a Department Element with Child Employee Elements

```
SELECT XMLElement("Department", XMLAgg(XMLElement("Employee",
                                                  e.job_id||' '||e.last_name)
                                       ORDER BY e.last_name))
  AS "Dept_list"
  FROM hr.employees e
  WHERE e.department_id = 30 OR e.department_id = 40;

Dept_list
------------------
<Department>
  <Employee>PU_CLERK Baida</Employee>
  <Employee>PU_CLERK Colmenares</Employee>
  <Employee>PU_CLERK Himuro</Employee>
  <Employee>PU_CLERK Khoo</Employee>
  <Employee>HR_REP Mavris</Employee>
  <Employee>PU_MAN Raphaely</Employee>
  <Employee>PU_CLERK Tobias</Employee>
</Department>

1 row selected.
```

The result is a *single* row, because XMLAgg aggregates the employee rows.

Example 18–13 shows how to use the GROUP BY clause to group the returned set of rows into multiple groups, forming multiple Department elements. (The result is shown here pretty-printed, for clarity.)

Example 18–13 XMLAGG: Using GROUP BY to Generate Multiple Department Elements

```
SELECT XMLElement("Department", XMLAttributes(department_id AS "deptno"),
                  XMLAgg(XMLElement("Employee", e.job_id||' '||e.last_name)))
   AS "Dept_list"
   FROM hr.employees e
   GROUP BY e.department_id;
```

```
Dept_list
------------------
<Department deptno="30">
  <Employee>PU_MAN Raphaely</Employee>
  <Employee>PU_CLERK Colmenares</Employee>
  <Employee>PU_CLERK Himuro</Employee>
  <Employee>PU_CLERK Tobias</Employee>
  <Employee>PU_CLERK Baida</Employee>
  <Employee>PU_CLERK Khoo</Employee></Department>

<Department deptno="40">
  <Employee>HR_REP Mavris</Employee>
</Department>

2 rows selected.
```

You can order the employees within each department by using the `ORDER BY` clause inside the `XMLAgg` expression.

> **Note:** Within the `ORDER BY` clause, Oracle Database does not interpret number literals as column positions, as it does in other uses of this clause.

Function `XMLAgg` can be used to reflect the hierarchical nature of some relationships that exist in tables. Example 18–14 generates a department element for department 30. Within this element is a child element `emp` for each employee of the department. Within each employee element is a `dependent` element for each dependent of that employee.

Example 18–14 XMLAGG: Generating Nested Elements

```
SELECT last_name, employee_id FROM employees WHERE department_id = 30;

LAST_NAME                 EMPLOYEE_ID
------------------------- -----------
Raphaely                          114
Khoo                              115
Baida                             116
Tobias                            117
Himuro                            118
Colmenares                        119

6 rows selected.
```

A `dependents` table holds the dependents of each employee.

```
CREATE TABLE hr.dependents (id NUMBER(4) PRIMARY KEY,
                            employee_id NUMBER(4),
                            name VARCHAR2(10));
Table created.
INSERT INTO dependents VALUES (1, 114, 'MARK');
1 row created.
INSERT INTO dependents VALUES (2, 114, 'JACK');
1 row created.
INSERT INTO dependents VALUES (3, 115, 'JANE');
1 row created.
INSERT INTO dependents VALUES (4, 116, 'HELEN');
```

```
1 row created.
INSERT INTO dependents VALUES (5, 116, 'FRANK');
1 row created.
COMMIT;
Commit complete.
```

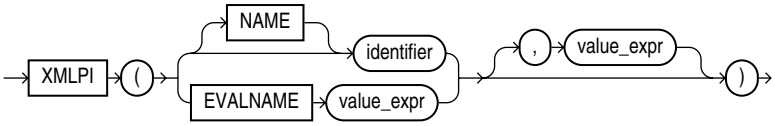
The following query generates the XML data for a department that contains the information about dependents. (The result is shown here pretty-printed, for clarity.)

```
SELECT
  XMLElement(
    "Department",
    XMLAttributes(d.department_name AS "name"),
    (SELECT
       XMLAgg(XMLElement("emp",
                         XMLAttributes(e.last_name AS name),
                         (SELECT XMLAgg(XMLElement("dependent",
                                        XMLAttributes(de.name AS "name")))
                            FROM dependents de
                            WHERE de.employee_id = e.employee_id)))
       FROM employees e
       WHERE e.department_id = d.department_id)) AS "dept_list"
  FROM departments d
  WHERE department_id = 30;

dept_list
--------------------------------------------------------------------------------
<Department name="Purchasing">
  <emp NAME="Raphaely">
    <dependent name="MARK"></dependent>
    <dependent name="JACK"></dependent>
  </emp><emp NAME="Khoo">
    <dependent name="JANE"></dependent>
  </emp>
  <emp NAME="Baida">
    <dependent name="HELEN"></dependent>
    <dependent name="FRANK"></dependent>
  </emp><emp NAME="Tobias"></emp>
  <emp NAME="Himuro"></emp>
  <emp NAME="Colmenares"></emp>
</Department>

1 row selected.
```

XMLPI SQL/XML Function

You use SQL/XML standard function `XMLPI` to construct an XML processing instruction (PI). Figure 18–6 shows the syntax:

Figure 18–6 XMLPI Syntax



Argument *value_expr* is evaluated, and the string result is appended to the optional identifier (*identifier*), separated by a space. This concatenation is then enclosed between `"<?"` and `"?>"` to create the processing instruction. That is, if

string-result is the result of evaluating *value_expr*, then the generated processing instruction is <?*identifier string-result*?>. If *string-result* is the empty string, '', then the function returns <?*identifier*?>.

As an alternative to using keyword NAME followed by a *literal* string *identifier*, you can use keyword EVALNAME followed by an expression that evaluates to a string to be used as the identifier.

An error is raised if the constructed XML is not a legal XML processing instruction. In particular:

- *identifier* must *not* be the word "xml" (uppercase, lowercase, or mixed case).
- *string-result* must *not* contain the character sequence "?>".

Function XMLPI returns an instance of XMLType. If *string-result* is NULL, then it returns NULL.

Example 18–15 uses XMLPI to generate a simple processing instruction.

Example 18–15 Using SQL/XML Function XMLPI

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

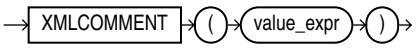
This results in the following output:

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

1 row selected.
```

XMLCOMMENT SQL/XML Function

You use SQL/XML standard function XMLComment to construct an XML comment. Figure 18–7 shows the syntax:

Figure 18–7 XMLComment Syntax



Argument *value_expr* is evaluated to a string, and the result is used as the body of the generated XML comment. The result is thus <!--*string-result*-->, where *string-result* is the string result of evaluating *value_expr*. If *string-result* is the empty string, then the comment is empty: <!---->.

An error is raised if the constructed XML is not a legal XML comment. In particular, *string-result* must *not* contain two consecutive hyphens (-): "--".

Function XMLComment returns an instance of XMLType. If *string-result* is NULL, then the function returns NULL.

Example 18–16 uses XMLComment to generate a simple XML comment.

Example 18–16 Using SQL/XML Function XMLCOMMENT

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

This query results in the following output:

```
CMNT
```

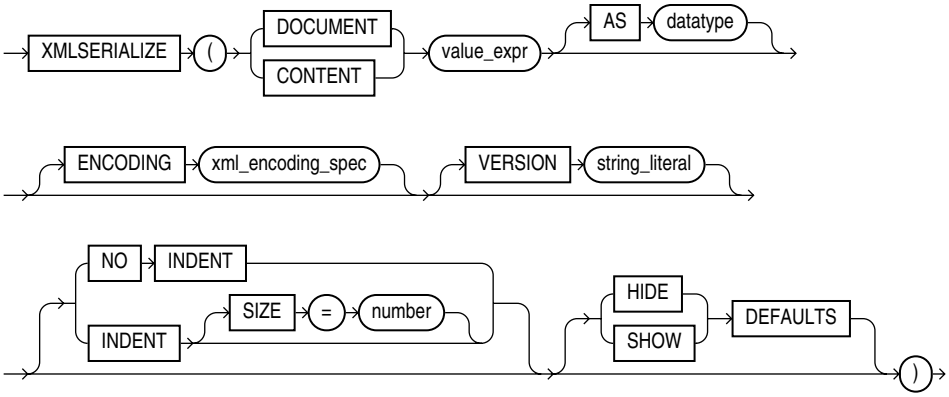
```
-------------------------
<!--This is a comment-->
```

XMLSERIALIZE SQL/XML Function

You use SQL/XML standard function `XMLSerialize` to obtain a string or LOB representation of XML data.

Figure 18–8 shows the syntax of `XMLSerialize`:

Figure 18–8 XMLSerialize Syntax



Argument *value_expr* is evaluated, and the resulting `XMLType` instance is serialized to produce the content of the created string or LOB. If present[1], the specified *datatype* must be one of the following (the default data type is `CLOB`):

- `VARCHAR2(N)`, where *N* is the size in bytes (limit 4000)
- `CLOB`
- `BLOB`

If you specify `DOCUMENT`, then the result of evaluating *value_expr* must be a well-formed document. In particular, it must have a single root. If the result is not a well-formed document, then an error is raised. If you specify `CONTENT`, however, then the result of *value_expr* is *not* checked for being well-formed.

If *value_expr* evaluates to `NULL` or to the empty string (`''`), then function `XMLSerialize` returns `NULL`.

The `ENCODING` clause specifies the character encoding for XML data that is serialized as a `BLOB` instance. *xml_encoding_spec* is an XML encoding declaration (`encoding="..."`). If *datatype* is `BLOB` and you specify an `ENCODING` clause, then the output is encoded as specified, and *xml_encoding_spec* is added to the prolog to indicate the `BLOB` encoding. If you specify an `ENCODING` clause with a *datatype* other than `BLOB`, then an error is raised.

If you specify the `VERSION`, then that version is used in the XML declaration (`<?xml version="..." ...?>`).

If you specify `NO INDENT`, then all insignificant whitespace is stripped, so that it does not appear in the output. If you specify `INDENT SIZE = N`, where *N* is a whole number, then the output is *pretty-printed* using a relative indentation of *N* spaces. If *N* is `0`, then pretty-printing inserts a newline character after each element, placing each

[1] The SQL/XML standard requires argument *data-type* to be present, but it is *optional* in the Oracle XML DB implementation of the standard, for ease of use.

element on a line by itself, but there is no other insignificant whitespace in the output. If you specify `INDENT` without a `SIZE` specification, then 2-space indenting is used. If you specify neither `NO INDENT` nor `INDENT`, then the behavior (pretty-printing or not) is indeterminate.

`HIDE DEFAULTS` and `SHOW DEFAULTS` apply only to XML schema-based data. If you specify `SHOW DEFAULTS` and the input data is missing any optional elements or attributes for which the XML schema defines default values, then those elements or attributes are included in the output with their default values. If you specify `HIDE DEFAULTS`, then no such elements or attributes are included in the output. `HIDE DEFAULTS` is the default behavior.

Example 18–17 uses `XMLSerialize` to produce a `CLOB` instance containing serialized XML data.

Example 18–17 Using SQL/XML Function XMLSERIALIZE

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

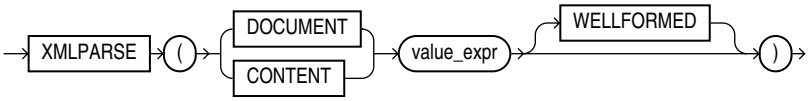
This results in the following output:

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

XMLPARSE SQL/XML Function

You use SQL/XML standard function `XMLParse` to parse a string containing XML data and construct a corresponding `XMLType` instance. Figure 18–9 shows the syntax:

Figure 18–9 XMLParse Syntax



Argument *value_expr* is evaluated to produce the string that is parsed. If you specify `DOCUMENT`, then *value_expr* must correspond to a *singly rooted*, well-formed XML document. If you specify `CONTENT`, then *value_expr* need only correspond to a well-formed XML fragment (it need not be singly rooted).

Keyword `WELLFORMED` is an Oracle XML DB extension to the SQL/XML standard. When you specify `WELLFORMED`, you are informing the parser that argument *value_expr* is well-formed, so Oracle XML DB does *not* check to ensure that it is well-formed.

Function `XMLParse` returns an instance of `XMLType`. If *value_expr* evaluates to `NULL`, then the function returns `NULL`.

Example 18–18 uses `XMLParse` to parse a string of XML code and produce an `XMLType` instance.

Example 18–18 Using SQL/XML Function XMLPARSE

```
SELECT XMLParse(CONTENT
                '124 <purchaseOrder poNo="12435">
                       <customerName> Acme Enterprises</customerName>
                       <itemNo>32987457</itemNo>
                     </purchaseOrder>'
```

```
                WELLFORMED)
  AS po FROM DUAL d;
```

This results in the following output:

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

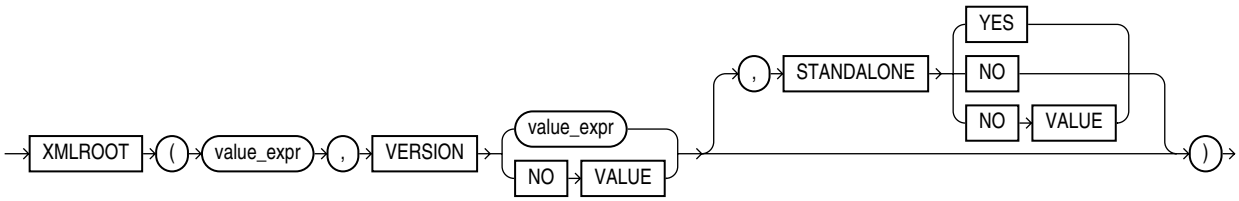
> **See Also:** http://www.w3.org/TR/REC-xml/, *Extensible Markup Language (XML) 1.0*, for the definition of well-formed XML documents and fragments

XMLROOT SQL Function

SQL function `XMLRoot` was at one time part of the SQL/XML standard, but it is *deprecated* as a standard function as of SQL/XML 2005. It remains available in Oracle XML DB, as an Oracle SQL function.

You use `XMLRoot` to add a `VERSION` property, and optionally a `STANDALONE` property, to the root information item of an XML value. Typically, this is done to ensure data-model compliance. Figure 18–10 shows the syntax of `XMLRoot`:

Figure 18–10 XMLRoot Syntax



The first argument, *`xml-expression`*, is evaluated, and the indicated properties (`VERSION`, `STANDALONE`) and their values are added to a new prolog for the resulting `XMLType` instance. If the evaluated *`xml-expression`* already contains a prolog, then an error is raised.

Second argument *`string-valued-expression`* (which follows keyword `VERSION`) is evaluated, and the resulting string is used as the value of the prolog `version` property. The value of the prolog `standalone` property (lowercase) is taken from the optional third argument `STANDALONE YES` or `NO` value. If `NOVALUE` is used for `VERSION`, then `"version=1.0"` is used in the resulting prolog. If `NOVALUE` is used for `STANDALONE`, then the `standalone` property is omitted from the resulting prolog.

Function `XMLRoot` returns an instance of `XMLType`. If first argument *`xml-expression`* evaluates to `NULL`, then the function returns `NULL`.

Example 18–19 uses `XMLRoot` to add an XML declaration with `version` and `standalone` attributes.

Example 18–19 Using Oracle SQL Function XMLRoot

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

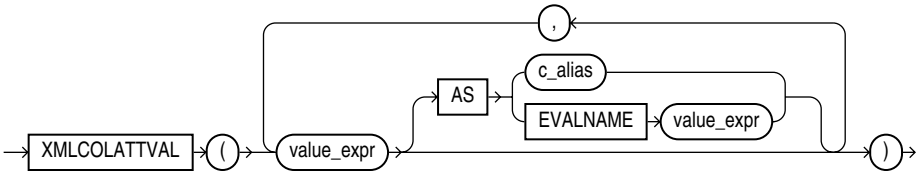
This results in the following output:

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

1 row selected.
```

XMLCOLATTVAL SQL Function

Oracle SQL function `XMLColAttVal` generates a forest of XML `column` elements containing the values of the arguments passed in. This function is an Oracle extension to the SQL/XML ANSI-ISO standard functions. Figure 18–11 shows the `XMLColAttVal` syntax.

Figure 18–11 XMLCOLATTVAL Syntax



The arguments are used as the values of the `name` attribute of the `column` element. The *`c_alias`* values are used as the attribute identifiers.

As an alternative to using keyword `AS` followed by a *literal* string *`c_alias`*, you can use `AS EVALNAME` followed by an expression that evaluates to a string to be used as the attribute identifier.

Because argument values *`value_expr`* are used only as attribute *values*, they need *not* be escaped in any way. This is in contrast to function `XMLForest`. It means that you can use `XMLColAttVal` to transport SQL columns and values without escaping.

Example 18–20 uses `XMLColAttVal` to generate an `Emp` element for each employee, with a `name` attribute, and with `column` elements that have the employee hire date and department as the content.

Example 18–20 XMLCOLATTVAL: Generating Elements with Attribute and Child Elements

```
SELECT XMLElement("Emp",
                  XMLAttributes(e.first_name ||' '||e.last_name AS "fullname" ),
                  XMLColAttVal(e.hire_date, e.department_id AS "department"))
  AS "RESULT"
  FROM hr.employees e
  WHERE e.department_id = 30;
```

This query produces the following XML result. (The result is shown here pretty-printed, for clarity.)

```
RESULT
------------------------------------------------------------
<Emp fullname="Den Raphaely">
  <column name = "HIRE_DATE">2002-12-07</column>
  <column name = "department">30</column>
</Emp>
<Emp fullname="Alexander Khoo">
```

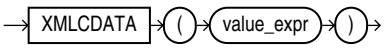
```
  <column name = "HIRE_DATE">2003-05-18</column>
  <column name = "department">30</column>
</Emp>
<Emp fullname="Shelli Baida">
  <column name = "HIRE_DATE">2005-12-24</column>
  <column name = "department">30</column>
</Emp>
<Emp fullname="Sigal Tobias">
  <column name = "HIRE_DATE">2005-07-24</column>
  <column name = "department">30</column>
</Emp>
<Emp fullname="Guy Himuro">
  <column name = "HIRE_DATE">2006-11-15</column>
  <column name = "department">30</column>
</Emp>
<Emp fullname="Karen Colmenares">
  <column name = "HIRE_DATE">2007-08-10</column>
  <column name = "department">30</column>
</Emp>

6 rows selected.
```

See Also: Example 18–8, "XMLFOREST: Generating Elements with Attribute and Child Elements"

XMLCDATA SQL Function

You use Oracle SQL function `XMLCDATA` to generate an XML `CDATA` section. Figure 18–12 shows the syntax:

Figure 18–12 XMLCDATA Syntax



Argument *`value_expr`* is evaluated to a string, and the result is used as the body of the generated XML `CDATA` section, `<![CDATA[`*`string-result`*`]]>`, where *`string-result`* is the result of evaluating *`value_expr`*. If *`string-result`* is the empty string, then the `CDATA` section is empty: `<![CDATA[]]>`.

An error is raised if the constructed XML is not a legal XML `CDATA` section. In particular, *`string-result`* must *not* contain two consecutive right brackets (`]`): `"]]"`.

Function `XMLCDATA` returns an instance of `XMLType`. If *`string-result`* is `NULL`, then the function returns `NULL`.

Example 18–21 uses `XMLCDATA` to generate an XML `CDATA` section.

Example 18–21 Using Oracle SQL Function XMLCDATA

```
SELECT XMLElement("PurchaseOrder",
                  XMLElement("Address",
                             XMLCDATA('100 Pennsylvania Ave.'),
                             XMLElement("City", 'Washington, D.C.')))
  AS RESULT FROM DUAL;
```

This results in the following output. (The result is shown here pretty-printed, for clarity.)

```
RESULT
--------------------------
```

```
<PurchaseOrder>
  <Address>
    <![CDATA[100 Pennsylvania Ave.]]>
    <City>Washington, D.C.</City>
  </Address>
</PurchaseOrder>
```

Generating XML using DBMS_XMLGEN

PL/SQL package DBMS_XMLGEN creates XML documents from SQL query results. It retrieves an XML document as a CLOB or XMLType value.

It provides a *fetch* interface, whereby you can specify the maximum number of rows to retrieve and the number of rows to skip. For example, the first fetch could retrieve a maximum of ten rows, skipping the first four. This is especially useful for pagination requirements in Web applications.

Package 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 and the enclosing tag names.

See Also:

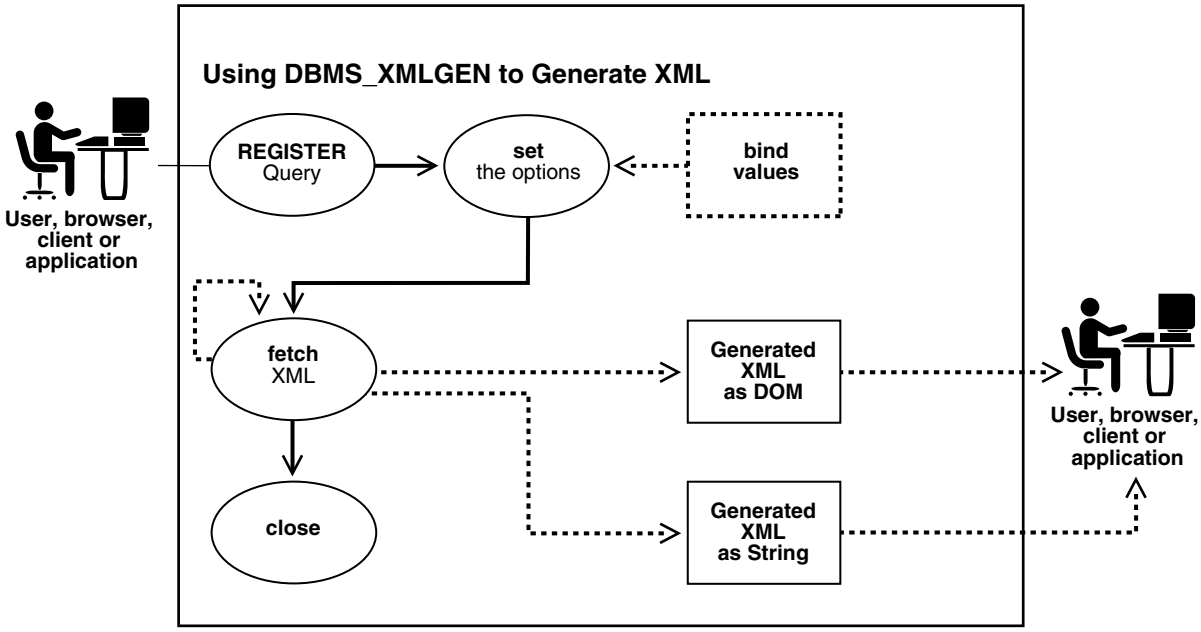
- *Oracle Database PL/SQL Packages and Types Reference*
- *Oracle XML Developer's Kit Programmer's Guide* (compare OracleXMLQuery with DBMS_XMLGEN)

Using PL/SQL Package DBMS_XMLGEN

Figure 18–13 illustrates how to use package DBMS_XMLGEN. The steps are as follows:

1. Get the context from the package by supplying a SQL query and calling PL/SQL function newContext.
2. Pass the context to all procedures or functions in the package to set the various options. For example, to set the ROW element name, use setRowTag(ctx), where ctx is the context got from the previous newContext call.
3. Get the XML result, using PL/SQL function getXML or getXMLType. By setting the maximum number of rows to be retrieved for each fetch using PL/SQL procedure setMaxRows, you can call either of these functions repeatedly, retrieving up to the maximum number of rows for each call. These functions return XML data (as a CLOB value and as an instance of XMLType, respectively), unless there are no rows retrieved. In that case, these functions return NULL. To determine how many rows were retrieved, use PL/SQL function getNumRowsProcessed.
4. You can reset the query to start again and repeat step 3.
5. Call PL/SQL procedure closeContext to free up any previously allocated resources.

Figure 18–13 Using PL/SQL Package DBMS_XMLGEN



In conjunction with a SQL query, PL/SQL method `DBMS_XMLGEN.getXML()` typically returns a result similar to the following, as a `CLOB` value:

```
<?xml version="1.0"?>
<ROWSET>
 <ROW>
  <EMPLOYEE_ID>100</EMPLOYEE_ID>
  <FIRST_NAME>Steven</FIRST_NAME>
  <LAST_NAME>King</LAST_NAME>
  <EMAIL>SKING</EMAIL>
  <PHONE_NUMBER>515.123.4567</PHONE_NUMBER>
  <HIRE_DATE>17-JUN-87</HIRE_DATE>
  <JOB_ID>AD_PRES</JOB_ID>
  <SALARY>24000</SALARY>
  <DEPARTMENT_ID>90</DEPARTMENT_ID>
 </ROW>
 <ROW>
  <EMPLOYEE_ID>101</EMPLOYEE_ID>
  <FIRST_NAME>Neena</FIRST_NAME>
  <LAST_NAME>Kochhar</LAST_NAME>
  <EMAIL>NKOCHHAR</EMAIL>
  <PHONE_NUMBER>515.123.4568</PHONE_NUMBER>
  <HIRE_DATE>21-SEP-89</HIRE_DATE>
  <JOB_ID>AD_VP</JOB_ID>
  <SALARY>17000</SALARY>
  <MANAGER_ID>100</MANAGER_ID>
  <DEPARTMENT_ID>90</DEPARTMENT_ID>
 </ROW>
</ROWSET>
```

The default mapping between relational data and XML data is as follows:

- Each row returned by the SQL query maps to an XML element with the default element name `ROW`.

- Each column returned by the SQL query maps to a child element of the ROW element.
- The entire result is wrapped in a ROWSET element.
- Binary data is transformed to its hexadecimal representation.

Element names ROW and ROWSET can be replaced with names you choose, using DBMS_XMLGEN procedures setRowTagName and setRowSetTagName, respectively.

The CLOB value returned by getXML has the same encoding as the database character set. If the database character set is SHIFTJIS, then the XML document returned is also SHIFTJIS.

Functions and Procedures of Package DBMS_XMLGEN

Table 18–1 describes the functions and procedures of package DBMS_XMLGEN.

Table 18–1 DBMS_XMLGEN Functions and Procedures

Function or Procedure	Description
SUBTYPE ctxHandle IS NUMBER	The context handle used by all functions. Document Type Definition (DTD) or schema specifications: NONE CONSTANT NUMBER:= 0; DTD CONSTANT NUMBER:= 1; SCHEMA CONSTANT NUMBER:= 2; Can be used in function getXML to specify whether to generate a DTD or XML schema or neither (NONE). Only the NONE specification is supported.
newContext()	Given a query string, generate a new context handle to be used in subsequent functions.
newContext(queryString IN VARCHAR2)	Returns a new context *Parameter:* queryString (IN) - the query string, the result of which must be converted to XML *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.
newContext(queryString IN SYS_REFCURSOR) RETURN ctxHandle;	Creates a new context handle from a PL/SQL cursor variable. The context handle can be used for the rest of the functions.
newContextFromHierarchy(queryString IN VARCHAR2) RETURN ctxHandle;	*Parameter:* queryString (IN) - the query string, the result of which must be converted to XML. The query is a hierarchical query typically formed using a CONNECT BY clause, and the result must have the same property as the result set generated by a CONNECT BY query. The result set must have only two columns, the level number and an XML value. The level number is used to determine the hierarchical position of the XML value within the result XML document. *Returns:* Context handle. Call this function first to obtain a handle that you can use in the getXML and other functions to get a hierarchical XML with recursive elements back from the result.
setRowTag()	Sets the name of the element separating all the rows. The default name is ROW.

Table 18–1 (Cont.) DBMS_XMLGEN Functions and Procedures

Function or Procedure	Description
`setRowTag(ctx IN ctxHandle, rowTag IN VARCHAR2);`	*Parameters:* `ctx(IN)` - the context handle obtained from the `newContext` call. `rowTag(IN)` - the name of the `ROW` element. A `NULL` value for `rowTag` indicates that you do not want the `ROW` element to be present. Call this procedure to set the name of the `ROW` element, if you do not want the default `ROW` name to show up. You can also set `rowTag` to `NULL` to suppress the `ROW` element itself. However, since getXML returns complete XML documents, not XML fragments, there must be a (single) root element. Therefore, an error is raised if both the `rowTag` value and the `rowSetTag` value (see `setRowSetTag`, next) are `NULL` and there is more than one column or row in the output.
`setRowSetTag()`	Sets the name of the document root element. The default name is `ROWSET`
`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 root element to be used in the output. A `NULL` value for `rowSetTag` indicates that you do *not* want the `ROWSET` element to be present. Call this procedure to set the name of the document root element, if you do not want the default name `ROWSET` to be used. You can set `rowSetTag` to `NULL` to suppress printing of the document root element. However, since function `getXML` returns complete XML documents, not XML fragments, there must be a (single) root element. Therefore, an error is raised if both the `rowTag` value and the `rowSetTag` value (see `setRowTag`, previous) are `NULL` and there is more than one column or row in the output, or if the `rowSetTag` value is `NULL` and there is more than one row in the output.
`getXML()`	Gets the XML document by fetching the maximum number of rows specified. It appends the XML document to the `CLOB` passed in.
`getXML(ctx IN ctxHandle, clobval IN OUT NCOPY clob, dtdOrSchema IN number:= NONE);`	*Parameters:* `ctx(IN)` - The context handle obtained from calling `newContext`. `clobval(IN/OUT)` - the `CLOB` to which the XML document is to be appended, `dtdOrSchema(IN)` - whether you 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.

Table 18–1 (Cont.) DBMS_XMLGEN Functions and Procedures

Function or Procedure	Description
`getXML()`	Generates the XML document and returns it as a `CLOB`.

`getXML(ctx IN ctxHandle,`
`  dtdOrSchema IN number:= NONE)`
`  RETURN clob;`
	Parameters:
	`ctx(IN)` - The context handle obtained from calling `newContext`.
	`dtdOrSchema(IN)` - whether to generate a DTD or XML schema. This parameter is *not* supported.
	Returns: A temporary `CLOB` containing the document. Free the temporary `CLOB` obtained from this function using the `DBMS_LOB.freeTemporary` call.

`getXMLType(`
`  ctx IN ctxHandle,`
`  dtdOrSchema IN number:= NONE)`
`  RETURN XMLType;`
	Parameters:
	`ctx(IN)` - The context handle obtained from calling `newContext`.
	`dtdOrSchema(IN)` - whether to generate a DTD or XML schema. This parameter is *not* supported.
	Returns: An `XMLType` instance containing the document.

`getXML(`
`  sqlQuery IN VARCHAR2,`
`  dtdOrSchema IN NUMBER := NONE)`
`  RETURN CLOB;`
	Converts the query results from the passed in SQL query string to XML format, and returns the XML as a `CLOB`.

`getXMLType(`
`  sqlQuery IN VARCHAR2,`
`  dtdOrSchema IN NUMBER := NONE)`
`  RETURN XMLType;`
	Converts the query results from the passed in SQL query string to XML format, and returns the XML as a `CLOB`.

`getNumRowsProcessed()`	Gets the number of SQL rows processed when generating XML data using function `getXML`. This count does not include the number of rows *skipped* before generating XML data.
`getNumRowsProcessed(ctx IN ctxHandle)` `  RETURN number;`	*Parameter:* `queryString(IN)`- the query string, the result of which must be converted to XML *Returns:* 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* before generating XML data. Use this function to determine the terminating condition if you are calling `getXML` in a loop. Note that `getXML` would always generate an XML document even if there are no rows present.
`setMaxRows()`	Sets the maximum number of rows to fetch from the SQL query result for every invocation of the `getXML` call. It is an error to call this function on a context handle created by function `newContextFromHierarchy`.

Table 18–1 (Cont.) DBMS_XMLGEN Functions and Procedures

Function or Procedure	Description
`setMaxRows(ctx IN ctxHandle, maxRows IN NUMBER);`	*Parameters:* `ctx(IN)` - the context handle corresponding to the query executed, `maxRows(IN)` - the maximum number of rows to get for each 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. It is an error to call this procedure on a context handle created by function `newContextFromHierarchy`.
`setSkipRows()`	Skips a given number of rows before generating the XML output for every call to `getXML`. It is an error to call this function on a context handle created by function `newContextFromHierarchy`.
`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 for each 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. It is an error to call this function on a context handle created by function `newContextFromHierarchy`.
`setConvertSpecialChars()`	Determines whether or not special characters in the XML data must be converted into their escaped XML equivalent. For example, the < sign is converted to `<`. The default behavior is to perform escape conversions.
`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 **<**, **>**, **"**, **'**, and so on, which must be preceded by an escape character. It is expensive to 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, this function can be called to improve performance.
`useItemTagsForColl()`	Sets the name of the collection elements. The default name for collection elements it 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.
`useItemTagsForColl(ctx IN ctxHandle);`	*Parameter:* `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 action and generate the collection column name with the `_ITEM` tag appended to it, by calling this procedure.

Table 18–1 (Cont.) DBMS_XMLGEN Functions and Procedures

Function or Procedure	Description
`restartQuery()`	Restarts the query and generate the XML from the first row again.
`restartQuery(ctx IN ctxHandle);`	*Parameter:* `ctx(IN)` - the context handle corresponding to the current query. You can call this to start executing the query again, without having to create a new context.
`closeContext()`	Closes a given context and releases all resources associated with that context, including the SQL cursor and bind and define buffers, and so on.
`closeContext(ctx IN ctxHandle);`	*Parameter:* `ctx(IN)` - the context handle to close. Closes all resources associated with this handle. After this you cannot use the handle for any other `DBMS_XMLGEN` function call.
Conversion Functions	

`convert(`
`  xmlData IN varchar2,`
`  flag IN NUMBER := ENTITY_ENCODE)`
`  RETURN VARCHAR2;`
	Encodes or decodes the XML data string argument.
	■ Encoding refers to replacing entity references such as < to their escaped equivalent, such as `<`.
	■ Decoding refers to the reverse conversion.

`convert(`
`  xmlData IN CLOB,`
`  flag IN NUMBER := ENTITY_ENCODE)`
`  RETURN CLOB;`
	Encodes or decodes the passed in XML `CLOB` data.
	■ Encoding refers to replacing entity references such as < to their escaped equivalent, such as `<`.
	■ Decoding refers to the reverse conversion.

NULL Handling
`setNullHandling(ctx IN ctxHandle,`
`  flag IN NUMBER);`
	The `setNullHandling` flag values are:
	■ `DROP_NULLS CONSTANT NUMBER := 0;`
	This is the default setting and leaves out the tag for `NULL` elements.
	■ `NULL_ATTR CONSTANT NUMBER := 1;`
	This sets `xsi:nil = "true"`.
	■ `EMPTY_TAG CONSTANT NUMBER := 2;`
	This sets, for example, `<foo/>`.

`useNullAttributeIndicator(`
`  ctx IN ctxHandle,`
`  attrind IN BOOLEAN := TRUE);`
	`useNullAttributeIndicator` is a shortcut for `setNullHandling(ctx, NULL_ATTR)`.

`setBindValue(`
`  ctx IN ctxHandle,`
`  bindVariableName IN VARCHAR2,`
`  bindValue IN VARCHAR2);`
	Sets bind value for the bind variable appearing in the query string associated with the context handle. The query string with bind variables cannot be executed until all of the bind variables are set values using `setBindValue`.

`clearBindValue(ctx IN ctxHandle);`	Clears all the bind values for all the bind variables appearing in the query string associated with the context handle. Afterwards, all of the bind variables must rebind new values using `setBindValue`.

DBMS_XMLGEN Examples

Example 18–22 uses `DBMS_XMLGEN` to create an XML document by selecting employee data from an object-relational table and putting the resulting `CLOB` value into a table.

Example 18–22 DBMS_XMLGEN: Generating Simple XML

```
CREATE TABLE temp_clob_tab (result CLOB);
```

```
DECLARE
  qryCtx DBMS_XMLGEN.ctxHandle;
  result CLOB;
BEGIN
  qryCtx := DBMS_XMLGEN.newContext(
              'SELECT * FROM hr.employees WHERE employee_id = 101');
  -- Set the row header to be EMPLOYEE
  DBMS_XMLGEN.setRowTag(qryCtx, 'EMPLOYEE');
  -- Get the result
  result := DBMS_XMLGEN.getXML(qryCtx);
  INSERT INTO temp_clob_tab VALUES(result);
  --Close context
  DBMS_XMLGEN.closeContext(qryCtx);
END;
/
```

That generates the following XML document:

```
SELECT * FROM temp_clob_tab;

RESULT
-------------------------------------------------------
<?xml version="1.0"?>
<ROWSET>
 <EMPLOYEE>
  <EMPLOYEE_ID>101</EMPLOYEE_ID>
  <FIRST_NAME>Neena</FIRST_NAME>
  <LAST_NAME>Kochhar</LAST_NAME>
  <EMAIL>NKOCHHAR</EMAIL>
  <PHONE_NUMBER>515.123.4568</PHONE_NUMBER>
  <HIRE_DATE>21-SEP-05</HIRE_DATE>
  <JOB_ID>AD_VP</JOB_ID>
  <SALARY>17000</SALARY>
  <MANAGER_ID>100</MANAGER_ID>
  <DEPARTMENT_ID>90</DEPARTMENT_ID>
 </EMPLOYEE>
</ROWSET>

1 row selected.
```

Instead of generating all of the XML data for all rows, you can use the fetch interface of package `DBMS_XMLGEN` to retrieve a fixed number of rows each time. This speeds up response time and can help in scaling applications that need a Document Object Model (DOM) Application Program Interface (API) on the resulting XML, particularly if the number of rows is large.

Example 18–23 uses `DBMS_XMLGEN` to retrieve results from table `HR.employees`:

Example 18–23 DBMS_XMLGEN: Generating Simple XML with Pagination (Fetch)

```
-- 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 hr.employees');
  -- Set the maximum number of rows to be 2
  DBMS_XMLGEN.setMaxRows(qryCtx, 2);
  LOOP
```

```
    -- Get the result
    result := DBMS_XMLGEN.getXML(qryCtx);
    -- If no rows were processed, then quit
    EXIT WHEN DBMS_XMLGEN.getNumRowsProcessed(qryCtx) = 0;

    -- Do some processing with the lob data
    --   Insert 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
  DBMS_XMLGEN.closeContext(qryCtx);
END;
/

SELECT * FROM temp_clob_tab WHERE rownum < 3;

RESULT
-----------------------------------------------------------
<?xml version="1.0"?>
<ROWSET>
 <ROW>
  <EMPLOYEE_ID>100</EMPLOYEE_ID>
  <FIRST_NAME>Steven</FIRST_NAME>
  <LAST_NAME>King</LAST_NAME>
  <EMAIL>SKING</EMAIL>
  <PHONE_NUMBER>515.123.4567</PHONE_NUMBER>
  <HIRE_DATE>17-JUN-03</HIRE_DATE>
  <JOB_ID>AD_PRES</JOB_ID>
  <SALARY>24000</SALARY>
  <DEPARTMENT_ID>90</DEPARTMENT_ID>
 </ROW>
 <ROW>
  <EMPLOYEE_ID>101</EMPLOYEE_ID>
  <FIRST_NAME>Neena</FIRST_NAME>
  <LAST_NAME>Kochhar</LAST_NAME>
  <EMAIL>NKOCHHAR</EMAIL>
  <PHONE_NUMBER>515.123.4568</PHONE_NUMBER>
  <HIRE_DATE>21-SEP-05</HIRE_DATE>
  <JOB_ID>AD_VP</JOB_ID>
  <SALARY>17000</SALARY>
  <MANAGER_ID>100</MANAGER_ID>
  <DEPARTMENT_ID>90</DEPARTMENT_ID>
 </ROW>
</ROWSET>

<?xml version="1.0"?>
<ROWSET>
 <ROW>
  <EMPLOYEE_ID>102</EMPLOYEE_ID>
  <FIRST_NAME>Lex</FIRST_NAME>
  <LAST_NAME>De Haan</LAST_NAME>
  <EMAIL>LDEHAAN</EMAIL>
  <PHONE_NUMBER>515.123.4569</PHONE_NUMBER>
  <HIRE_DATE>13-JAN-01</HIRE_DATE>
  <JOB_ID>AD_VP</JOB_ID>
  <SALARY>17000</SALARY>
  <MANAGER_ID>100</MANAGER_ID>
  <DEPARTMENT_ID>90</DEPARTMENT_ID>
```

```
 </ROW>
 <ROW>
  <EMPLOYEE_ID>103</EMPLOYEE_ID>
  <FIRST_NAME>Alexander</FIRST_NAME>
  <LAST_NAME>Hunold</LAST_NAME>
  <EMAIL>AHUNOLD</EMAIL>
  <PHONE_NUMBER>590.423.4567</PHONE_NUMBER>
  <HIRE_DATE>03-JAN-06</HIRE_DATE>
  <JOB_ID>IT_PROG</JOB_ID>
  <SALARY>9000</SALARY>
  <MANAGER_ID>102</MANAGER_ID>
  <DEPARTMENT_ID>60</DEPARTMENT_ID>
 </ROW>
</ROWSET>

2 rows selected.
```

Example 18–24 uses DBMS_XMLGEN with object types to represent nested structures.

Example 18–24 DBMS_XMLGEN: Generating XML using Object Types

```
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 new_departments);
CREATE TYPE emp_t AS OBJECT ("@employee_id"   NUMBER,
                             last_name        VARCHAR2(20));
/
INSERT INTO new_departments VALUES (10, 'SALES');
INSERT INTO new_departments VALUES (20, 'ACCOUNTING');
INSERT INTO new_employees   VALUES (30, 'Scott', 10);
INSERT INTO new_employees   VALUES (31, 'Mary',  10);
INSERT INTO new_employees   VALUES (40, 'John',  20);
INSERT INTO new_employees   VALUES (41, 'Jerry', 20);
COMMIT;
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);
/
CREATE TABLE temp_clob_tab (result CLOB);
DECLARE
  qryCtx DBMS_XMLGEN.ctxHandle;
  result CLOB;
BEGIN
  DBMS_XMLGEN.setRowTag(qryCtx, NULL);
  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');
  -- now get the result
  result := DBMS_XMLGEN.getXML(qryCtx);
```

```
  INSERT INTO temp_clob_tab VALUES (result);
  -- close context
  DBMS_XMLGEN.closeContext(qryCtx);
END;
/
SELECT * FROM temp_clob_tab;
```

Here is the resulting XML:

```
RESULT
--------------------------------------------
<?xml version="1.0"?>
<ROWSET>
 <ROW>
  <DEPTXML department_id="10">
   <DEPARTMENT_NAME>SALES</DEPARTMENT_NAME>
   <EMPLIST>
    <EMP_T employee_id="30">
     <LAST_NAME>Scott</LAST_NAME>
    </EMP_T>
    <EMP_T employee_id="31">
     <LAST_NAME>Mary</LAST_NAME>
    </EMP_T>
   </EMPLIST>
  </DEPTXML>
 </ROW>
 <ROW>
  <DEPTXML department_id="20">
   <DEPARTMENT_NAME>ACCOUNTING</DEPARTMENT_NAME>
   <EMPLIST>
    <EMP_T employee_id="40">
     <LAST_NAME>John</LAST_NAME>
    </EMP_T>
    <EMP_T employee_id="41">
     <LAST_NAME>Jerry</LAST_NAME>
    </EMP_T>
   </EMPLIST>
  </DEPTXML>
 </ROW>
</ROWSET>

1 row selected.
```

With relational data, the result is an XML document without nested elements. To obtain nested XML structures, you can use object-relational data, where the mapping is as follows:

- *Object types* map as an XML element – see Chapter 7, "XML Schema Storage and Query: Basic".
- *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.
>
> When used in column names or attribute names, the at-sign (`@`) is translated into an attribute of the enclosing XML element in the mapping.

When you provide a user-defined data-type instance to DBMS_XMLGEN functions, the user-defined data-type instance is mapped to an XML document using canonical mapping: the *attributes* of the user-defined data type are mapped to XML *elements*. Attributes with names starting with an at-sign character (@) are mapped to attributes of the preceding element.

User-defined data-type instances can be used for nesting in the resulting XML document.

For example, consider the tables emp and dept defined in Example 18–25. To generate a hierarchical view of the data, that is, departments with their employees, Example 18–25 defines suitable object types to create the structure inside the database.

Example 18–25 DBMS_XMLGEN: Generating XML using User-Defined Data-Type Instances

```
CREATE TABLE dept (deptno NUMBER PRIMARY KEY, dname VARCHAR2(20));
CREATE TABLE emp (empno   NUMBER PRIMARY KEY, ename VARCHAR2(20),
                  deptno  NUMBER REFERENCES dept);

-- empno is preceded by an at-sign (@) to indicate that it must
-- be mapped as an attribute of the enclosing Employee element.
CREATE TYPE emp_t AS OBJECT ("@empno" NUMBER,  -- empno defined as attribute
                             ename   VARCHAR2(20));
/
INSERT INTO DEPT VALUES (10, 'Sports');
INSERT INTO DEPT VALUES(20, 'Accounting');
INSERT INTO EMP VALUES(200, 'John',  10);
INSERT INTO EMP VALUES(300, 'Jack',  10);
INSERT INTO EMP VALUES(400, 'Mary',  20);
INSERT INTO EMP VALUES(500, 'Jerry', 20);
COMMIT;
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 contains a list of employees.
-- You can now query the employee and department tables and get
-- the result as an XML document, as follows:
CREATE TABLE temp_clob_tab (result CLOB);
DECLARE
  qryCtx DBMS_XMLGEN.ctxHandle;
  RESULT CLOB;
BEGIN
  -- get 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 maximum number of rows to 5
  DBMS_XMLGEN.setMaxRows(qryCtx, 5);
  -- set no row tag for this result, since there is a single ADT column
  DBMS_XMLGEN.setRowTag(qryCtx, NULL);
  LOOP
```

```
    -- get result
    result := DBMS_XMLGEN.getXML(qryCtx);
    -- if there were no rows processed, then quit
    EXIT WHEN DBMS_XMLGEN.getNumRowsProcessed(qryCtx) = 0;
    -- do something with the result
    INSERT INTO temp_clob_tab VALUES (result);
  END LOOP;
END;
/
```

The `MULTISET` keyword for Oracle SQL function `cast` treats the employees working in the department as a list, which `cast` assigns to the appropriate collection type. A department instance is created using constructor `dept_t`, and `DBMS_XMLGEN` routines create the XML data for the object instance.

```
SELECT * FROM temp_clob_tab;

RESULT
--------------------------------
<?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">
  <DNAME>Accounting</DNAME>
  <EMPLIST>
   <EMP_T empno="400">
    <ENAME>Mary</ENAME>
   </EMP_T>
   <EMP_T empno="500">
    <ENAME>Jerry</ENAME>
   </EMP_T>
  </EMPLIST>
 </DEPTXML>
</ROWSET>

1 row selected.
```

The default name `ROW` is not present because it was set to `NULL`. The `deptno` and `empno` have become attributes of the enclosing element.

Example 18–26 uses `DBMS_XMLGEN.getXMLType` to generate a purchase order document in XML format using object views.

Example 18–26 DBMS_XMLGEN: Generating an XML Purchase Order

```
-- Create relational schema and define object views
-- DBMS_XMLGEN maps user-defined data-type attribute names that start
--    with an at-sign (@) 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 ordered collection table
CREATE TYPE lineitems_ntabtyp AS TABLE OF lineitem_typ
/
-- Purchase Order object type
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 Purchase Order relational model tables
-- Customer table
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));
-- Purchase Order table
CREATE TABLE po_tab (PONo       NUMBER,        /* purchase order number */
                     Custno     NUMBER     /*  foreign KEY referencing customer */
                                CONSTRAINT po_cust_fk REFERENCES customer_tab,
                     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 purchase order in XML format
CREATE TABLE po_xml_tab (poid  NUMBER, podoc XMLType)
/
-- 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
  VALUES (2001, 2, '20-APR-1997', '20-MAY-1997',
```

```
          '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 package DBMS_XMLGEN to generate purchase order in XML format
--   and store XMLType in table po_xml
DECLARE
  qryCtx DBMS_XMLGEN.ctxHandle;
  pxml XMLType;
  cxml CLOB;
BEGIN
  -- get query context;
  qryCtx := DBMS_XMLGEN.newContext('SELECT pono,deref(cust_ref) customer,
                                           p.orderdate,
                                           p.shipdate,
                                           lineitems_ntab lineitems,
                                           shiptoaddr
                                      FROM po p');
  -- set 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
    -- get purchase order in XML format
    pxml := DBMS_XMLGEN.getXMLType(qryCtx);
    -- if there were no rows processed, then quit
    EXIT WHEN DBMS_XMLGEN.getNumRowsProcessed(qryCtx) = 0;
    -- Store XMLType po in po_xml table (get the pono out)
    INSERT INTO po_xml_tab(poid, poDoc)
      VALUES(XMLCast(XMLQuery('//PONO/text()' PASSING pxml RETURNING CONTENT)
                     AS NUMBER),
            pxml);
  END LOOP;
END;
/
```

This query then produces two XML purchase-order documents:

```
SELECT XMLSerialize(DOCUMENT x.podoc AS CLOB) xpo FROM po_xml_tab x;

XPO
--------------------------------------------------
 <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>
 <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>
  <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>
   <ZIP>53715</ZIP>
  </SHIPTOADDR>
 </PurchaseOrder>

2 rows selected.
```

Example 18–27 shows how to open a cursor variable for a query and use that cursor variable to create a new context handle for DBMS_XMLGEN.

Example 18–27 DBMS_XMLGEN: Generating a New Context Handle from a REF Cursor

```
CREATE TABLE emp_tab (emp_id       NUMBER PRIMARY KEY,
                      name         VARCHAR2(20),
                      dept_id      NUMBER);
Table created.
INSERT INTO emp_tab VALUES (122, 'Scott',  301);
1 row created.
INSERT INTO emp_tab VALUES (123, 'Mary',   472);
1 row created.
INSERT INTO emp_tab VALUES (124, 'John',   93);
1 row created.
INSERT INTO emp_tab VALUES (125, 'Howard', 488);
1 row created.
INSERT INTO emp_tab VALUES (126, 'Sue',    16);
1 row created.
COMMIT;

DECLARE
  ctx     NUMBER;
  maxrow  NUMBER;
  xmldoc  CLOB;
  refcur  SYS_REFCURSOR;
BEGIN
  DBMS_LOB.createtemporary(xmldoc, TRUE);
  maxrow := 3;
  OPEN refcur FOR 'SELECT * FROM emp_tab WHERE ROWNUM <= :1' USING maxrow;
  ctx := DBMS_XMLGEN.newContext(refcur);
   -- xmldoc will have 3 rows
  DBMS_XMLGEN.getXML(ctx, xmldoc, DBMS_XMLGEN.NONE);
  DBMS_OUTPUT.put_line(xmldoc);
  DBMS_LOB.freetemporary(xmldoc);
  CLOSE refcur;
  DBMS_XMLGEN.closeContext(ctx);
END;
/
<?xml version="1.0"?>
<ROWSET>
 <ROW>
  <EMP_ID>122</EMP_ID>
  <NAME>Scott</NAME>
  <DEPT_ID>301</DEPT_ID>
 </ROW>
 <ROW>
  <EMP_ID>123</EMP_ID>
  <NAME>Mary</NAME>
  <DEPT_ID>472</DEPT_ID>
 </ROW>
```

```
 <ROW>
  <EMP_ID>124</EMP_ID>
  <NAME>John</NAME>
  <DEPT_ID>93</DEPT_ID>
 </ROW>
</ROWSET>

PL/SQL procedure successfully completed.
```

> **See Also:** *Oracle Database PL/SQL Language Reference* for more information about cursor variables (`REF CURSOR`)

Example 18–28 shows how to specify `NULL` handling when using `DBMS_XMLGEN`.

Example 18–28 DBMS_XMLGEN: Specifying NULL Handling

```
CREATE TABLE emp_tab (emp_id       NUMBER PRIMARY KEY,
                      name         VARCHAR2(20),
                      dept_id      NUMBER);
Table created.
INSERT INTO emp_tab VALUES (30, 'Scott', NULL);
1 row created.
INSERT INTO emp_tab VALUES (31, 'Mary', NULL);
1 row created.
INSERT INTO emp_tab VALUES (40, 'John', NULL);
1 row created.
COMMIT;
CREATE TABLE temp_clob_tab (result CLOB);
Table created.

DECLARE
  qryCtx DBMS_XMLGEN.ctxHandle;
  result CLOB;
BEGIN
  qryCtx := DBMS_XMLGEN.newContext('SELECT * FROM emp_tab where name = :NAME');
  -- Set the row header to be EMPLOYEE
  DBMS_XMLGEN.setRowTag(qryCtx, 'EMPLOYEE');
  -- Drop nulls
  DBMS_XMLGEN.setBindValue(qryCtx, 'NAME', 'Scott');
  DBMS_XMLGEN.setNullHandling(qryCtx, DBMS_XMLGEN.DROP_NULLS);
  result := DBMS_XMLGEN.getXML(qryCtx);
  INSERT INTO temp_clob_tab VALUES(result);
  -- Null attribute
  DBMS_XMLGEN.setBindValue(qryCtx, 'NAME', 'Mary');
  DBMS_XMLGEN.setNullHandling(qryCtx, DBMS_XMLGEN.NULL_ATTR);
  result := DBMS_XMLGEN.getXML(qryCtx);
  INSERT INTO temp_clob_tab VALUES(result);
  -- Empty tag
  DBMS_XMLGEN.setBindValue(qryCtx, 'NAME', 'John');
  DBMS_XMLGEN.setNullHandling(qryCtx, DBMS_XMLGEN.EMPTY_TAG);
  result := DBMS_XMLGEN.getXML(qryCtx);
  INSERT INTO temp_clob_tab VALUES(result);
  --Close context
  DBMS_XMLGEN.closeContext(qryCtx);
END;
/

PL/SQL procedure successfully completed.

SELECT * FROM temp_clob_tab;
```

```
RESULT
-----------------------------------------
<?xml version="1.0"?>
<ROWSET>
 <EMPLOYEE>
  <EMP_ID>30</EMP_ID>
  <NAME>Scott</NAME>
 </EMPLOYEE>
</ROWSET>

<?xml version="1.0"?>
<ROWSET xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance">
 <EMPLOYEE>
  <EMP_ID>31</EMP_ID>
  <NAME>Mary</NAME>
  <DEPT_ID xsi:nil = "true"/>
 </EMPLOYEE>
</ROWSET>

<?xml version="1.0"?>
<ROWSET>
 <EMPLOYEE>
  <EMP_ID>40</EMP_ID>
  <NAME>John</NAME>
  <DEPT_ID/>
 </EMPLOYEE>
</ROWSET>

3 rows selected.
```

Function `DBMS_XMLGEN.newContextFromHierarchy` takes as argument a hierarchical query string, which is typically formulated with a `CONNECT BY` clause. It returns a context that can be used to generate a hierarchical XML document with recursive elements.

The hierarchical query returns two columns, the level number (a pseudocolumn generated by `CONNECT BY` query) and an `XMLType`. The level is used to determine the position of the `XMLType` value within the hierarchy of the result XML document.

It is an error to set the skip number of rows or the maximum number of rows for a context created using `newContextFromHierarchy`.

Example 18–29 uses `DBMS_ XMLGEN.newContextFromHierarchy` to generate a manager–employee hierarchy.

Example 18–29 DBMS_XMLGEN: Generating Recursive XML with a Hierarchical Query

```
CREATE TABLE sqlx_display (id NUMBER, xmldoc XMLType);
Table created.

DECLARE
  qryctx DBMS_XMLGEN.ctxhandle;
  result XMLType;
BEGIN
  qryctx :=
    DBMS_XMLGEN.newContextFromHierarchy(
      'SELECT level,
              XMLElement("employees",
                         XMLElement("enumber", employee_id),
                         XMLElement("name", last_name),
```

```
                        XMLElement("Salary", salary),
                        XMLElement("Hiredate", hire_date))
        FROM hr.employees
        START WITH last_name=''De Haan'' CONNECT BY PRIOR employee_id=manager_id
        ORDER SIBLINGS BY hire_date');
  result := DBMS_XMLGEN.getxmltype(qryctx);
  DBMS_OUTPUT.put_line('<result num rows>');
  DBMS_OUTPUT.put_line(to_char(DBMS_XMLGEN.getNumRowsProcessed(qryctx)));
  DBMS_OUTPUT.put_line('</result num rows>');
  INSERT INTO sqlx_display VALUES (2, result);
  COMMIT;
  DBMS_XMLGEN.closecontext(qryctx);
END;
/
<result num rows>
6
</result num rows>
PL/SQL procedure successfully completed.

SELECT xmldoc FROM sqlx_display WHERE id = 2;

XMLDOC
-------------------------------------------------------
<?xml version="1.0"?>
<employees>
  <enumber>102</enumber>
  <name>De Haan</name>
  <Salary>17000</Salary>
  <Hiredate>2001-01-13</Hiredate>
  <employees>
    <enumber>103</enumber>
    <name>Hunold</name>
    <Salary>9000</Salary>
    <Hiredate>2006-01-03</Hiredate>
    <employees>
      <enumber>105</enumber>
      <name>Austin</name>
      <Salary>4800</Salary>
      <Hiredate>2005-06-25</Hiredate>
    </employees>
    <employees>
      <enumber>106</enumber>
      <name>Pataballa</name>
      <Salary>4800</Salary>
      <Hiredate>2006-02-05</Hiredate>
    </employees>
    <employees>
      <enumber>107</enumber>
      <name>Lorentz</name>
      <Salary>4200</Salary>
      <Hiredate>2007-02-07</Hiredate>
    </employees>
    <employees>
      <enumber>104</enumber>
      <name>Ernst</name>
      <Salary>6000</Salary>
      <Hiredate>2007-05-21</Hiredate>
    </employees>
  </employees>
</employees>
```

```
1 row selected.
```

By default, the ROWSET tag is NULL: there is no default ROWSET tag used to enclose the XML result. However, you can explicitly set the ROWSET tag by using procedure setRowSetTag, as follows:

```
CREATE TABLE gg (x XMLType);
Table created.

DECLARE
  qryctx DBMS_XMLGEN.ctxhandle;
  result CLOB;
BEGIN
  qryctx := DBMS_XMLGEN.newContextFromHierarchy(
              'SELECT level,
                      XMLElement("NAME", last_name) AS myname FROM hr.employees
               CONNECT BY PRIOR employee_id=manager_id
               START WITH employee_id = 102');
  DBMS_XMLGEN.setRowSetTag(qryctx, 'mynum_hierarchy');
  result:=DBMS_XMLGEN.getxml(qryctx);
  DBMS_OUTPUT.put_line('<result num rows>');
  DBMS_OUTPUT.put_line(to_char(DBMS_XMLGEN.getNumRowsProcessed(qryctx)));
  DBMS_OUTPUT.put_line('</result num rows>');
  INSERT INTO gg VALUES(XMLType(result));
  COMMIT;
  DBMS_XMLGEN.closecontext(qryctx);
END;
/
<result num rows>
6
</result num rows>
PL/SQL procedure successfully completed.

SELECT * FROM gg;

X
------------------------------------------------------------
<?xml version="1.0"?>
<mynum_hierarchy>
  <NAME>De Haan
    <NAME>Hunold
      <NAME>Ernst</NAME>
      <NAME>Austin</NAME>
      <NAME>Pataballa</NAME>
      <NAME>Lorentz</NAME>
    </NAME>
  </NAME>
</mynum_hierarchy>

1 row selected.
```

If the query string used to create a context contains host variables, you can use PL/SQL method setBindValue() to give the variables values before query execution. Example 18–30 illustrates this.

Example 18–30 DBMS_XMLGEN: Binding Query Variables using SETBINDVALUE()

```
-- Bind one variable
DECLARE
```

```
  ctx NUMBER;
  xmldoc CLOB;
BEGIN
  ctx := DBMS_XMLGEN.newContext(
           'SELECT * FROM employees WHERE employee_id = :NO');
  DBMS_XMLGEN.setBindValue(ctx, 'NO', '145');
  xmldoc := DBMS_XMLGEN.getXML(ctx);
  DBMS_OUTPUT.put_line(xmldoc);
  DBMS_XMLGEN.closeContext(ctx);
EXCEPTION
  WHEN OTHERS THEN DBMS_XMLGEN.closeContext(ctx);
  RAISE;
END;
/
<?xml version="1.0"?>
<ROWSET>
 <ROW>
  <EMPLOYEE_ID>145</EMPLOYEE_ID>
  <FIRST_NAME>John</FIRST_NAME>
  <LAST_NAME>Russell</LAST_NAME>
  <EMAIL>JRUSSEL</EMAIL>
  <PHONE_NUMBER>011.44.1344.429268</PHONE_NUMBER>
  <HIRE_DATE>01-OCT-04</HIRE_DATE>
  <JOB_ID>SA_MAN</JOB_ID>
  <SALARY>14000</SALARY>
  <COMMISSION_PCT>.4</COMMISSION_PCT>
  <MANAGER_ID>100</MANAGER_ID>
  <DEPARTMENT_ID>80</DEPARTMENT_ID>
 </ROW>
</ROWSET>

PL/SQL procedure successfully completed.

-- Bind one variable twice with different values
DECLARE
  ctx NUMBER;
  xmldoc CLOB;
BEGIN
  ctx := DBMS_XMLGEN.newContext('SELECT * FROM employees
                                 WHERE hire_date = :MDATE');
  DBMS_XMLGEN.setBindValue(ctx, 'MDATE', '01-OCT-04');
  xmldoc := DBMS_XMLGEN.getXML(ctx);
  DBMS_OUTPUT.put_line(xmldoc);
  DBMS_XMLGEN.setBindValue(ctx, 'MDATE', '10-MAR-05');
  xmldoc := DBMS_XMLGEN.getXML(ctx);
  DBMS_OUTPUT.put_line(xmldoc);
  DBMS_XMLGEN.closeContext(ctx);
EXCEPTION
  WHEN OTHERS THEN DBMS_XMLGEN.closeContext(ctx);
  RAISE;
END;
/
<?xml version="1.0"?>
<ROWSET>
 <ROW>
  <EMPLOYEE_ID>145</EMPLOYEE_ID>
  <FIRST_NAME>John</FIRST_NAME>
  <LAST_NAME>Russell</LAST_NAME>
  <EMAIL>JRUSSEL</EMAIL>
  <PHONE_NUMBER>011.44.1344.429268</PHONE_NUMBER>
```

```
  <HIRE_DATE>01-OCT-04</HIRE_DATE>
  <JOB_ID>SA_MAN</JOB_ID>
  <SALARY>14000</SALARY>
  <COMMISSION_PCT>.4</COMMISSION_PCT>
  <MANAGER_ID>100</MANAGER_ID>
  <DEPARTMENT_ID>80</DEPARTMENT_ID>
 </ROW>
</ROWSET>

<?xml version="1.0"?>
<ROWSET>
 <ROW>
  <EMPLOYEE_ID>147</EMPLOYEE_ID>
  <FIRST_NAME>Alberto</FIRST_NAME>
  <LAST_NAME>Errazuriz</LAST_NAME>
  <EMAIL>AERRAZUR</EMAIL>
  <PHONE_NUMBER>011.44.1344.429278</PHONE_NUMBER>
  <HIRE_DATE>10-MAR-05</HIRE_DATE>
  <JOB_ID>SA_MAN</JOB_ID>
  <SALARY>12000</SALARY>
  <COMMISSION_PCT>.3</COMMISSION_PCT>
  <MANAGER_ID>100</MANAGER_ID>
  <DEPARTMENT_ID>80</DEPARTMENT_ID>
 </ROW>
 <ROW>
  <EMPLOYEE_ID>159</EMPLOYEE_ID>
  <FIRST_NAME>Lindsey</FIRST_NAME>
  <LAST_NAME>Smith</LAST_NAME>
  <EMAIL>LSMITH</EMAIL>
  <PHONE_NUMBER>011.44.1345.729268</PHONE_NUMBER>
  <HIRE_DATE>10-MAR-97</HIRE_DATE>
  <JOB_ID>SA_REP</JOB_ID>
  <SALARY>8000</SALARY>
  <COMMISSION_PCT>.3</COMMISSION_PCT>
  <MANAGER_ID>146</MANAGER_ID>
  <DEPARTMENT_ID>80</DEPARTMENT_ID>
 </ROW>
</ROWSET>
PL/SQL procedure successfully completed.
-- Bind two variables
DECLARE
  ctx NUMBER;
  xmldoc CLOB;
BEGIN
  ctx := DBMS_XMLGEN.newContext('SELECT * FROM employees
                                  WHERE employee_id = :NO
                                    AND hire_date = :MDATE');
  DBMS_XMLGEN.setBindValue(ctx, 'NO', '145');
  DBMS_XMLGEN.setBindValue(ctx, 'MDATE', '01-OCT-04');
  xmldoc := DBMS_XMLGEN.getXML(ctx);
  DBMS_OUTPUT.put_line(xmldoc);
  DBMS_XMLGEN.closeContext(ctx);
EXCEPTION
  WHEN OTHERS THEN DBMS_XMLGEN.closeContext(ctx);
  RAISE;
END;
/
<?xml version="1.0"?>
<ROWSET>
 <ROW>
```

```
  <EMPLOYEE_ID>145</EMPLOYEE_ID>
  <FIRST_NAME>John</FIRST_NAME>
  <LAST_NAME>Russell</LAST_NAME>
  <EMAIL>JRUSSEL</EMAIL>
  <PHONE_NUMBER>011.44.1344.429268</PHONE_NUMBER>
  <HIRE_DATE>01-OCT-04</HIRE_DATE>
  <JOB_ID>SA_MAN</JOB_ID>
  <SALARY>14000</SALARY>
  <COMMISSION_PCT>.4</COMMISSION_PCT>
  <MANAGER_ID>100</MANAGER_ID>
  <DEPARTMENT_ID>80</DEPARTMENT_ID>
 </ROW>
</ROWSET>
PL/SQL procedure successfully completed.
```

Generating XML using SQL Function SYS_XMLGEN

SQL function `sys_XMLGen` is Oracle-specific. It is similar to the SQL/XML standard function `XMLElement`, except that it takes a single argument and converts the result to an `XMLType` instance. Unlike the other XML generation functions, `sys_XMLGen` always returns a well-formed XML *document*. Unlike package `DBMS_XMLGEN`, which operates at a query level, `sys_XMLGen` operates at the row level, returning an XML *document for each row*.

Example 18–31 uses `sys_XMLGen` to query XML instances, returning an XML document for each row of relational data.

Example 18–31 Creating XML Data using SYS_XMLGEN

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

The resulting XML documents are as follows:

```
result
---------------
<?xml version="1.0"?>
<EMPLOYEE_ID>110</EMPLOYEE_ID>

<?xml version="1.0"?>
<EMPLOYEE_ID>139</EMPLOYEE_ID>

<?xml version="1.0"?>
<EMPLOYEE_ID>145</EMPLOYEE_ID>

3 rows selected.
```

SYS_XMLGEN Syntax

Oracle SQL function `sys_XMLGen` takes as argument a scalar value, object type, or `XMLType` instance to be converted to an XML document. It also takes an optional `XMLFormat` object (previously called `XMLGenFormatType)`, which you can use to specify formatting options for the resulting XML document. The syntax is shown in Figure 18–14.

Figure 18–14 SYS_XMLGEN Syntax

Expression *expr* evaluates to a particular row and column of the database. It can be a scalar value, a user-defined data-type instance, or an `XMLType` instance.

- If *expr* evaluates to a scalar value, then the function returns an XML element containing the scalar value.
- If *expr* evaluates to a user-defined data-type instance, then the function maps the user-defined data-type attributes to XML elements.
- If *expr* evaluates to an `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 *expr*. For example, if *expr* resolves to a column name, then the enclosing XML element has the same name as the column. If you want to format the XML document differently, then specify *fmt*, which is an instance of the `XMLFormat` object.

You can use a `WHERE` clause in a query to suppress `<ROW/>` tags with `sys_XMLGen`, if you do not want `NULL` values represented:

```
SELECT sys_XMLGen(x) FROM table_name WHERE x IS NOT NULL;
```

Example 18–32 retrieves the employee first name from table `HR.employees`, where the `employee_id` value is `110`, and it generates an `XMLType` instance containing an XML document with an `FIRST_NAME` element.

Example 18–32 SYS_XMLGEN: Generating an XML Element from a Database Column

```
SELECT XMLSerialize(DOCUMENT SYS_XMLGEN(first_name))
  FROM employees WHERE employee_id = 110;

XMLSERIALIZE(DOCUMENTSYS_XMLGEN(FIRST_NAME))
--------------------------------------------
<?xml version="1.0"?>
<FIRST_NAME>John</FIRST_NAME>

1 row selected.
```

Advantages of using Oracle SQL Function SYS_XMLGEN

Oracle SQL function `sys_XMLGen` has the following advantages:

- 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. For example, when you use an `XMLType` view that uses `sys_XMLGen` on top of an object type, Oracle XML DB rewrites these queries when possible. See also Chapter 8, "XPath Rewrite for Structured Storage".

`sys_XMLGen` creates an XML document from a user-defined data-type instance, a scalar value, or an `XMLType` instance. It returns an `XMLType` instance.

`sys_XMLGen` also accepts an optional `XMLFormat` object as argument, which you can use to customize the result. A `NULL` format object implies that the default mapping action is to be used.

Using XMLFormat Object Type

You can use the XMLFormat object to specify formatting arguments for Oracle SQL functions sys_XMLGen and sys_XMLAgg.

Function sys_XMLGen returns an XMLType instance containing an XML document. Oracle Database provides the XMLFormat object to format the output of sys_XMLGen.

Table 18–2 lists the attributes of object XMLFormat.

Table 18–2 Attributes of the XMLFormat Object

Attribute	Data Type	Purpose
enclTag	VARCHAR2(100)	The name of the enclosing tag for the result of the sys_XMLGen function. If the input to the function is a column name, then the column name is used as the default value. Otherwise, the default value is ROWSET. When schemaType is set to USE_GIVEN_SCHEMA, this attribute also provides the name of the XML schema element.
schemaType	VARCHAR2(100)	The type of schema generation for the output document. Valid values are 'NO_SCHEMA' and 'USE_GIVEN_SCHEMA'. The default value is 'NO_SCHEMA'.
schemaName	VARCHAR2(4000)	The name of the target schema used if schemaType is 'USE_GIVEN_SCHEMA'. If you specify schemaName, then the enclosing tag is used as the element name.
targetNameSpace	VARCHAR2(4000)	The target namespace if the schema is specified (that is, schemaType is GEN_SCHEMA_*, or USE_GIVEN_SCHEMA)
dburl	VARCHAR2(2000)	The URL to the database to be used if WITH_SCHEMA is specified. If this attribute is not specified, then a relative URL reference is used for the URL to the types.
processingIns	VARCHAR2(4000)	User-provided processing instructions. They are appended to the top of the function output, before the element.

You can use PL/SQL method createFormat() to implement the XMLFormat object. Method createFormat() of object XMLFormat accepts as arguments the enclosing element name, the XML schema type, and the XML schema name. Default values are provided for the other XMLFormat attributes.

See Also:

- Example 18–35 for an example of using createFormat to name the root element that is output by sys_XMLGen
- *Oracle Database SQL Language Reference* for more information about sys_XMLGen and the XMLFormat object

Oracle SQL function sys_XMLGen converts a scalar value to an element that contains the scalar value. The query in Example 18–33 illustrates this. It returns an XML document that contains the employee_id value as the content of element EMPLOYEE_ID.

Example 18–33 SYS_XMLGEN: Converting a Scalar Value to XML Element Contents

```
SELECT sys_XMLGen(employee_id) FROM hr.employees WHERE ROWNUM < 2;

SYS_XMLGEN(EMPLOYEE_ID)
---------------------------
<?xml version="1.0"?>
```

```
<EMPLOYEE_ID>100</EMPLOYEE_ID>

1 row selected.
```

The enclosing element name, in this case EMPLOYEE_ID, is derived from the column name passed to sys_XMLGen. The query result is a single row containing an XMLType instance that corresponds to a complete XML document.

In Example 18–33, the column name EMPLOYEE_ID is used by default for the XML element name. If the column name cannot be derived directly, then the default name ROW is used instead, as shown in Example 18–34.

Example 18–34 SYS_XMLGEN: Default Element Name ROW

```
SELECT sys_XMLGen(employee_id*2) FROM hr.employees WHERE ROWNUM < 2;

SYS_XMLGEN(EMPLOYEE_ID*2)
-------------------------
<?xml version="1.0"?>
<ROW>200</ROW>

1 row selected.
```

In Example 18–34, the argument to sys_XMLGen is not a simple column name, so the name of the output element tag cannot be a column name – the default element name, ROW, is used.

You can override the default ROW tag by supplying an XMLFormat object as the second sys_XMLGen argument. Example 18–35 illustrates this: the query passes a formatting argument to sys_XMLGen, to name the element explicitly.

Example 18–35 Overriding the Default Element Name using SYS_XMLGEN with XMLFormat

```
SELECT XMLSerialize(DOCUMENT
                    SYS_XMLGEN(employee_id*2,
                               XMLFormat.createformat('DOUBLE_ID'))
                    AS CLOB)
 FROM hr.employees WHERE ROWNUM < 2;

XMLSERIALIZE(DOCUMENTSYS_XMLGEN(EMPLOYEE_ID*2,XMLFORMAT.CREATEFORMAT('DOUBLE_ID'
--------------------------------------------------------------------------------
<?xml version="1.0"?>
<DOUBLE_ID>200</DOUBLE_ID>

1 row selected.
```

When you provide a user-defined data-type instance as an argument to sys_XMLGen, the instance is canonically mapped to an XML document. In this mapping, the user-defined data-type attributes are mapped to XML elements.

Any data-type attributes with names that start with an at sign (@) are mapped to attributes of the preceding XML element. User-defined data-type instances can be used to obtain nesting in the resulting XML document.

Example 18–36 shows how to generate hierarchical XML for the employee-and-department example of section "Generating XML using DBMS_XMLGEN" on page 18-22.

Example 18–36 SYS_XMLGEN: Converting a User-Defined Data-Type Instance to XML

```
CREATE OR REPLACE TYPE hr.emp_t AS OBJECT(empno    NUMBER(6),
                                          ename    VARCHAR2(25),
                                          job      VARCHAR2(10),
                                          mgr      NUMBER(6),
                                          hiredate DATE,
                                          sal      NUMBER(8,2),
                                          comm     NUMBER(2,2));
/
Type created.
CREATE OR REPLACE TYPE hr.emplist_t AS TABLE OF emp_t;
/
Type created.
CREATE OR REPLACE TYPE hr.dept_t AS OBJECT(deptno  NUMBER(4),
                                           dname   VARCHAR2(30),
                                           loc     VARCHAR2(4),
                                           emplist emplist_t);
/
Type created.

SELECT XMLSerialize(
         DOCUMENT
         SYS_XMLGEN(
           dept_t(department_id,
                  department_name,
                  d.location_id,
                  cast(MULTISET
                       (SELECT emp_t(e.employee_id, e.last_name,
                                     e.job_id, e.manager_id, e.hire_date,
                                     e.salary, e.commission_pct)
                          FROM hr.employees e
                          WHERE e.department_id = d.department_id)
                       AS emplist_t)))
         AS CLOB)
  AS deptxml
  FROM hr.departments d WHERE department_id = 10 OR department_id = 20;
```

The `MULTISET` keyword for Oracle SQL function `cast` treats the employees working in the department as a list, which `cast` assigns to the appropriate collection type. `sys_XMLGen` creates the XML data for the object instance.

The result is as follows. The default name `ROW` is present because the function cannot deduce the name of the input operand directly.

```
DEPTXML
-------------------------------------
<?xml version="1.0"?>
<ROW>
 <DEPTNO>10</DEPTNO>
 <DNAME>Administration</DNAME>
 <LOC>1700</LOC>
 <EMPLIST>
  <EMP_T>
   <EMPNO>200</EMPNO>
   <ENAME>Whalen</ENAME>
   <JOB>AD_ASST</JOB>
   <MGR>101</MGR>
   <HIREDATE>17-SEP-03</HIREDATE>
   <SAL>4400</SAL>
  </EMP_T>
```

```
 </EMPLIST>
</ROW>

<?xml version="1.0"?>
<ROW>
 <DEPTNO>20</DEPTNO>
 <DNAME>Marketing</DNAME>
 <LOC>1800</LOC>
 <EMPLIST>
  <EMP_T>
   <EMPNO>201</EMPNO>
   <ENAME>Hartstein</ENAME>
   <JOB>MK_MAN</JOB>
   <MGR>100</MGR>
   <HIREDATE>17-FEB-04</HIREDATE>
   <SAL>13000</SAL>
  </EMP_T>
  <EMP_T>
   <EMPNO>202</EMPNO>
   <ENAME>Fay</ENAME>
   <JOB>MK_REP</JOB>
   <MGR>201</MGR>
   <HIREDATE>17-AUG-05</HIREDATE>
   <SAL>6000</SAL>
  </EMP_T>
 </EMPLIST>
</ROW>

2 rows selected.
```

Note: The difference between using SQL function `sys_XMLGen` and PL/SQL package `DBMS_XMLGEN` is apparent from the preceding example. Function `sys_XMLGen` works inside SQL queries, and operates on the expressions and columns within the rows. Package `DBMS_XMLGEN` works on the entire result set.

If you pass an XML document to function `sys_XMLGen`, the function wraps the document (or fragment) with an element whose tag name is the default `ROW`, or the name passed in through the `XMLFormat` formatting object. This functionality can be used to turn XML fragments into well-formed documents. Example 18–37 illustrates this.

Example 18–37 SYS_XMLGEN: Converting an XMLType Instance

```
CREATE TABLE po_xml_tab (podoc XMLType);
Table created.

INSERT INTO po_xml_tab VALUES (XMLType('<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>'));
1 row created.
COMMIT;
```

This query extracts ENAME elements:

```
SELECT XMLQuery('/DOCUMENT/EMPLOYEE/ENAME' PASSING e.podoc RETURNING CONTENT)
  FROM po_xml_tab e;
```

The query result is an XML document fragment (pretty-printed here for clarity):

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

You can make such a fragment into a valid XML document by calling sys_XMLGen to wrap a root element around the fragment, as follows:

```
SELECT XMLSerialize(DOCUMENT
                    sys_XMLGen(XMLQuery('/DOCUMENT/EMPLOYEE/ENAME'
                                        PASSING e.podoc RETURNING CONTENT))
                    AS CLOB)
  FROM po_xml_tab e;
```

This places a ROW element around the fragment, as follows (pretty-printed here for clarity):

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

Note: If the input to sys_XMLGen is a column, then the column name is used as the default element name. You can override the element name using the XMLFormat formatting object as a second argument to sys_XMLGen. See "Using XMLFormat Object Type" on page 18-48.

Example 18–38 shows how to use sys_XMLGen with object views. (For any undefined entities here, refer to the code in Example 18–26 on page 18-34.)

Example 18–38 Using SYS_XMLGEN with Object Views

```
-- 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
SELECT XMLSerialize(DOCUMENT
                    SYS_XMLGEN(OBJECT_VALUE,
                               XMLFormat.createFormat('PurchaseOrder'))
                    AS CLOB) PO
  FROM po p WHERE p.pono=1001;
```

The query returns the purchase order in XML format:

```
PO
----------------------------------------------
<?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>
```

```
 <SHIPTOADDR/>
</PurchaseOrder>

1 row selected.
```

Generating XML using SQL Function SYS_XMLAGG

Oracle SQL function `sys_XMLAgg` aggregates all XML documents or fragments represented by an expression and produces a single XML document. It adds a new enclosing element with a default name, `ROWSET`. To format the XML document differently, use the `fmt` parameter.

Figure 18–15 SYS_XMLAGG Syntax

See Also: *Oracle Database SQL Language Reference*

Generating XML using XSQL Pages Publishing Framework

You can use Oracle XML DB XML to produce data for inclusion in your XSQL pages by using the `<xsql:include-xml>` action element.

The `SELECT` statement that appears inside a `<xsql:include-xml>` element should return a *single row* containing a *single column*. The column can be either a `CLOB` instance or a `VARCHAR2` value. It must contain a well-formed XML document. The XML document is parsed and included in your XSQL page.

> **See Also:** *Oracle XML Developer's Kit Programmer's Guide* for information about element `<xsql:include-xml>` and XSQL pages

Example 18–39 uses nested calls to function `XMLAgg` to aggregate the results of a dynamically-constructed XML document that contains departments and their employees into a single XML result document, which is then wrapped in a `DepartmentList` element.

Example 18–39 Using XSQL Servlet <xsql:include-xml> with Nested XMLAgg Functions

```
<xsql:include-xml connection="orcl92" xmlns:xsql="urn:oracle-xsql">
SELECT
  XMLSerialize(
    DOCUMENT
    XMLElement("DepartmentList",
               XMLAgg(XMLElement(
                        "Department",
                        XMLAttributes(department_id as "Id"),
                        XMLForest(department_name as "Name"),
                        (SELECT XMLElement("Employees",
                                           XMLAgg(XMLElement(
                                             "Employee",
                                             XMLAttributes(employee_id as "Id"),
                                             XMLForest(last_name as "Name",
                                                       salary   as "Salary",
                                                       job_id   as "Job"))))
                           FROM employees e
                           WHERE e.department_id = d.department_id))))
```

```
    AS CLOB)
  FROM departments d
  ORDER BY department_name;
</xsql:include-xml>
```

The query itself produces the following result:

```
XMLSERIALIZE(DOCUMENTXMLELEMENT("DEPARTMENTLIST",XMLAGG(XMLELEMENT("DEPARTMENT",
--------------------------------------------------------------------------------
<DepartmentList><Department Id="10"><Name>Administration</Name><Employees><Emplo
yee Id="200"><Name>Whalen</Name><Salary>4400</Salary><Job>AD_ASST</Job></Employe
e></Employees></Department><Department Id="20"><Name>Marketing</Name><Employees>
<Employee Id="201"><Name>Hartstein</Name><Salary>13000</Salary><Job>MK_MAN</Job>
</Employee><Employee Id="202"><Name>Fay</Name><Salary>6000</Salary><Job>MK_REP</
Job></Employee></Employees></Department>
...
</DepartmentList>

1 row selected.
```

The call to SQL/XML function `XMLSerialize` in Example 18–39 provides XSQL Servlet with a `CLOB` value instead of an `XMLType` instance. To display the results, XSQL Servlet needs a special environment, such as the XSQL Command-Line Utility, XSQL Servlet installed in a Web server, Java Server Pages (JSP), or a Java `XSQLRequest` object.

It is more efficient for the database to aggregate XML fragments into a single result document. Element `<xsql:include-xml>` encourages this approach by retrieving only the first row from the query you provide.

Example 18–40 extracts an aggregate list of all purchase orders of the film *Grand Illusion*. It uses table `purchaseorder` in sample database schema `OE`.

Example 18–40 Using XSQL Servlet <xsql:include-xml> with XMLElement and XMLAgg

```
CONNECT oe
Enter password: password

Connected.

SELECT XMLElement("GrandIllusionOrders",
                  XMLAgg(XMLQuery(
                           '$p/PurchaseOrder/LineItems/*[Part[@Id="37429121924"]]'
                           PASSING po.OBJECT_VALUE AS "p" RETURNING CONTENT)))
  FROM purchaseorder po;
```

This produces the following result.

```
XMLELEMENT("GRANDILLUSIONORDERS",XMLAGG(XMLQUERY('$P/PURCHASEORDER/LINEITEMS/*[P
--------------------------------------------------------------------------------
<GrandIllusionOrders>
  <LineItem ItemNumber="14">
    <Description>Grand Illusion</Description>
    <Part Id="37429121924" UnitPrice="39.95" Quantity="2"/>
  </LineItem>
  <LineItem ItemNumber="14">
    <Description>Grand Illusion</Description>
    <Part Id="37429121924" UnitPrice="39.95" Quantity="2"/>
  </LineItem>
  <LineItem ItemNumber="6">
    <Description>Grand Illusion</Description>
```

```
    <Part Id="37429121924" UnitPrice="39.95" Quantity="2"/>
  </LineItem>
  <LineItem ItemNumber="19">
    <Description>Grand Illusion</Description>
    <Part Id="37429121924" UnitPrice="39.95" Quantity="4"/>
  </LineItem>
  <LineItem ItemNumber="21">
    <Description>Grand Illusion</Description>
    <Part Id="37429121924" UnitPrice="39.95" Quantity="3"/>
  </LineItem>
  <LineItem ItemNumber="15">
    <Description>Grand Illusion</Description>
    <Part Id="37429121924" UnitPrice="39.95" Quantity="3"/>
  </LineItem>
  <LineItem ItemNumber="3">
    <Description>Grand Illusion</Description>
    <Part Id="37429121924" UnitPrice="39.95" Quantity="2"/>
  </LineItem>
  <LineItem ItemNumber="8">
    <Description>Grand Illusion</Description>
    <Part Id="37429121924" UnitPrice="39.95" Quantity="1"/>
  </LineItem>
  <LineItem ItemNumber="17">
    <Description>Grand Illusion</Description>
    <Part Id="37429121924" UnitPrice="39.95" Quantity="4"/>
  </LineItem>
</GrandIllusionOrders>

1 row selected.
```

To include this `XMLType` query result in your XSQL page, paste the query inside an `<xsql:include-xml>` element, and use XMLSerialize to return the result to the client as a `CLOB` value instead of as an `XMLType` instance:

```
<xsql:include-xml connection="orcl92" xmlns:xsql="urn:oracle-xsql">
SELECT XMLSerialize(DOCUMENT
                    XMLElement("GrandIllusionOrders",
                               XMLAgg(XMLQuery(
                                        '$p/PurchaseOrder/LineItems/*[Part[@Id="37429121924"]]'
                                        PASSING po.OBJECT_VALUE AS "p" RETURNING CONTENT)))
                    AS CLOB)
  FROM purchaseorder po;
</xsql:include-xml>
```

SQL/XML functions `XMLElement` and `XMLAgg` are used together here, to aggregate all of the XML fragments identified by the query into a single, well-formed XML document. Failing to do this results in an attempt by the XSQL page processor to parse a `CLOB` value that looks like this:

```
<LineItem>...</LineItem>
<LineItem>...</LineItem>
...
```

This is not well-formed XML because it does not have a single root element as required by the XML 1.0 recommendation. Functions `XMLElement` and `XMLAgg` work together to produce a well-formed result with single root element `GrandIllusionOrders`. This well-formed XML is then parsed and included in your XSQL page.

> **See Also:** *Oracle XML Developer's Kit Programmer's Guide*, the chapter, 'XSQL Page Publishing Framework'

Using XSLT and XSQL

With XSQL Pages, you have control over where XSLT is executed: in the database, the middle-tier, or the client. For database execution, use Oracle SQL function `XMLtransform` (or the equivalent) in your query. For middle-tier execution, add `<?xml-stylesheet?>` at the top of your template page. For client execution, add attribute `client="yes"` to PI `<?xml-stylesheet?>`.

With XSQL Pages, you can build pages that conditionally off-load style-sheet processing to the client, depending, for example, on what browser is used.

To improve performance and throughput, XSQL caches and pools XSLT style sheets (and database connections) in the middle tier. Depending on the application, you can further improve performance by avoiding transformation using Web Cache or other techniques, and you can further optimize to avoid repeatedly transforming the same (or static) data.

XSQL Pages can include a mix of static XML and dynamically produced XML. You can take advantage of this by using the database to create only the dynamic part of the page.

Generating XML using XML SQL Utility (XSU)

You can use Oracle XML SQL Utility (XSU) with Oracle Database to generate XML XML data on the middle tier or a client. XSU also supports generating XML data on tables with `XMLType` columns.

> **See Also:** *Oracle XML Developer's Kit Programmer's Guide* for information about XSU

Guidelines for Generating XML with Oracle XML DB

This section describes additional guidelines for generating XML using Oracle XML DB.

Ordering Query Results Before Aggregating, using XMLAGG ORDER BY Clause

To use the `XMLAgg ORDER BY` clause before aggregation, specify the `ORDER BY` clause following the first `XMLAGG` argument. This is illustrated in Example 18–41.

Example 18–41 Using XMLAGG ORDER BY Clause

```
CREATE TABLE dev_tab (dev         NUMBER,
                      dev_total   NUMBER,
                      devname     VARCHAR2(20));
Table created.
INSERT INTO dev_tab VALUES (16, 5,  'Alexis');
1 row created.
INSERT INTO dev_tab VALUES (2,  14, 'Han');
1 row created.
INSERT INTO dev_tab VALUES (1,  2,  'Jess');
1 row created.
INSERT INTO dev_tab VALUES (9,  88, 'Kurt');
1 row created.
COMMIT;
```

The result of the following query is aggregated according to the order of the `dev` column. (The result is shown here pretty-printed, for clarity.)

```
SELECT XMLAgg(XMLElement("Dev",
                         XMLAttributes(dev AS "id", dev_total AS "total"),
                         devname)
              ORDER BY dev)
  FROM tab1 dev_total;

XMLAGG(XMLELEMENT("DEV",XMLATTRIBUTES(DEVAS"ID",DEV_TOTALAS"TOTAL"),DEVNAME)ORDE
--------------------------------------------------------------------------------
<Dev id="1" total="2">Jess</Dev>
<Dev id="2" total="14">Han</Dev>
<Dev id="9" total="88">Kurt</Dev>
<Dev id="16" total="5">Alexis</Dev>

1 row selected.
```

Returning a Rowset using XMLTABLE

You can use standard SQL/XML function `XMLTable` to return a rowset with relevant portions of a document extracted as multiple rows, as shown in Example 18–42.

Example 18–42 Returning a Rowset using XMLTABLE

```
CONNECT oe
Enter password: password

Connected.

SELECT item.descr, item.partid
  FROM purchaseorder,
       XMLTable('$p/PurchaseOrder/LineItems/LineItem' PASSING OBJECT_VALUE
                COLUMNS descr  VARCHAR2(256) PATH 'Description',
                        partid VARCHAR2(14)  PATH 'Part/@Id') item
  WHERE item.partid = '715515012027'
     OR item.partid = '715515011921'
  ORDER BY partid;
```

This returns a rowset with just the descriptions and part IDs, ordered by part ID.

```
DESCR
--------------
PARTID
--------------
My Man Godfrey
715515011921

My Man Godfrey
715515011921

My Man Godfrey
715515011921

My Man Godfrey
715515011921

My Man Godfrey
715515011921

My Man Godfrey
715515011921
```

```
My Man Godfrey
715515011921

Mona Lisa
715515012027

Mona Lisa
715515012027

Mona Lisa
715515012027

Mona Lisa
715515012027

Mona Lisa
715515012027

Mona Lisa
715515012027

Mona Lisa
715515012027

Mona Lisa
715515012027

Mona Lisa
715515012027

16 rows selected.
```

19

XMLType Views

This chapter describes how to create and use XMLType views. It contains these topics:

- What Are XMLType Views?
- Creating Non-Schema-Based XMLType Views
- Creating XML Schema-Based XMLType Views
- Creating XMLType Views from XMLType Tables
- Referencing XMLType View Objects using SQL Function REF
- DML (Data Manipulation Language) on XMLType Views

What Are XMLType Views?

XMLType views wrap existing relational and object-relational data in XML formats. The major advantages of using XMLType views are:

- You can exploit Oracle XML DB XML features that use XML schema functionality without having to migrate your base legacy data.
- With XMLType views, you can experiment with various other forms of storage, besides the object-relational, CLOB, and binary XML storage available for XMLType tables.

XMLType views are similar to object views. Each row of an XMLType view corresponds to an XMLType instance. The object identifier for uniquely identifying each row in the view can be created using SQL/XML functions XMLCast and XMLQuery.

Throughout this chapter XML schema refers to the W3C XML Schema 1.0 recommendation, http://www.w3.org/XML/Schema.

There are two types of XMLType views:

- **Non-schema-based XMLType views.** These views do not confirm to a particular XML schema.
- **XML schema-based XMLType views.** As with XMLType tables, XMLType views that conform to a particular XML schema are called XML schema-based XMLType views. These provide stronger typing than non-schema-based XMLType views.

XPath rewrite of queries over XMLType views is enabled for both XML schema-based and non-schema-based XMLType views. XPath rewrite is described in Chapter 8, "XPath Rewrite for Structured Storage".

To create an XML schema-based XMLType view, first register your XML schema. If the view is an object view, that is, if it is constructed using an object type, then the XML

schema should have annotations that represent the bidirectional mapping from XML to SQL object types. `XMLType` views conforming to this registered XML schema can then be created by providing an underlying query that constructs instances of the appropriate SQL object type.

See Also:

- "Accessing XML Data in Oracle XML DB using Relational Views" on page 3-48
- Chapter 7, "XML Schema Storage and Query: Basic"

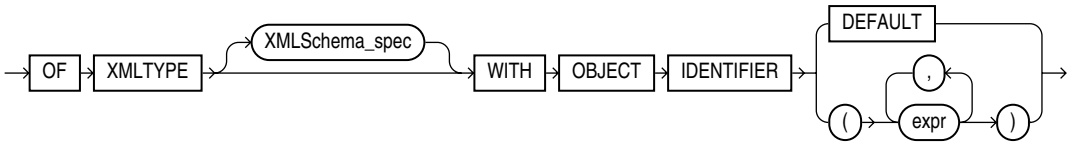
You can create `XMLType` views in any of the following ways:

- Based on SQL/XML publishing functions, such as `XMLElement`, `XMLForest`, `XMLConcat`, and `XMLAgg`. SQL/XML publishing functions can be used to construct both non-schema-based `XMLType` views and XML schema-based `XMLType` views. This enables construction of `XMLType` view from the underlying relational tables directly without physically migrating those relational legacy data into XML. However, to construct XML schema-based `XMLType` view, the XML schema must be registered and the XML value generated by SQL/XML publishing functions must be constrained to the XML schema.
- Based on object types, object views, and Oracle SQL function `sys_XMLGen`. Non-schema-based `XMLType` views can be constructed using object types, object views, and function `sys_XMLGen` and XML schema-based `XMLType` view can be constructed using object types and object views. This enables the construction of the `XMLType` view from underlying relational or object relational tables directly without physically migrating the relational or object relational legacy data into XML. Creating non-schema-based `XMLType` view requires the use of `sys_XMLGen` over existing object types or object views. Creating XML-schema-based `XMLType` view requires to annotate the XML schema with a mapping to existing object types or to generate the XML schema from the existing object types.
- Directly from an `XMLType` table.

Creating XMLType Views: Syntax

Figure 19–1 shows the `CREATE VIEW` clause for creating `XMLType` views. See *Oracle Database SQL Language Reference* for details on the `CREATE VIEW` syntax.

Figure 19–1 Creating XMLType Views Clause: Syntax



Creating Non-Schema-Based XMLType Views

Non-schema-based `XMLType` views are `XMLType` views whose resultant XML value is not constrained to be a particular element in a registered XML schema. You can create a non-schema-based `XMLType` view in either of these ways:

- Using SQL/XML publishing functions.

See Also: Chapter 18, "Generating XML Data from the Database", for details on SQL/XML publishing functions

- Using object types or object views, together with Oracle SQL function `sys_XMLGen`. This is convenient when you already have object types, views, and tables that you want to map to XML data.

  See Also: "Creating XML Schema-Based XMLType Views using Object Types or Object Views" on page 19-10

Creating Non-Schema-Based XMLType Views using SQL/XML Publishing Functions

Example 19–1 shows how to create an `XMLType` view using SQL/XML function `XMLELement`.

Example 19–1 Creating an XMLType View using XMLELEMENT

```
CREATE OR REPLACE VIEW emp_view OF XMLType
  WITH OBJECT ID (XMLCast(XMLQuery('/Emp/@empno'
                                   PASSING OBJECT_VALUE RETURNING CONTENT)
                          AS BINARY_DOUBLE)) AS
  SELECT XMLElement("Emp",
                    XMLAttributes(employee_id AS "empno"),
                    XMLForest(e.first_name ||' '|| e.last_name AS "name",
                              e.hire_date AS "hiredate"))
    AS "result" FROM employees e WHERE salary > 15000;

SELECT * FROM emp_view;

SYS_NC_ROWINFO$
--------------------------------------------------------------------------------
<Emp empno="100"><name>Steven King</name><hiredate>2003-06-17</hiredate></Emp>
<Emp empno="101"><name>Neena Kochhar</name><hiredate>2005-09-21</hiredate></Emp>
<Emp empno="102"><name>Lex De Haan</name><hiredate>2001-01-13</hiredate></Emp>
```

Existing data in relational tables or views can be exposed as XML this way. If a view is generated using a SQL/XML publishing function, then queries that access that view using XPath expressions can often be rewritten. These optimized queries can then directly access the underlying relational columns. See Chapter 8, "XPath Rewrite for Structured Storage" for details.

You can perform DML operations on these `XMLType` views, but, in general, you must write instead-of triggers to handle the DML operation.

Creating Non-Schema-Based XMLType Views using Object Types and SYS_XMLGEN

You can also create an `XMLType` view using object types and Oracle SQL function `sys_XMLGen`. Function `sys_XMLGen` accepts as argument an instance of an object type, and it generates a corresponding instance of `XMLType`. The query in Example 19–2 uses `sys_XMLGen` and produces the same result as the query of Example 19–1.

Example 19–2 Creating an XMLType View using Object Types and SYS_XMLGEN

```
CREATE TYPE emp_t AS OBJECT ("@empno"  NUMBER(6),
                             fname     VARCHAR2(20),
                             lname     VARCHAR2(25),
                             hiredate  DATE);
/
```

```
CREATE OR REPLACE VIEW employee_view OF XMLType
  WITH OBJECT ID (XMLCast(XMLQuery('/Emp/@empno'
                                   PASSING OBJECT_VALUE RETURNING CONTENT)
                          AS BINARY_DOUBLE)) AS
  SELECT sys_XMLGen(emp_t(e.employee_id, e.first_name, e.last_name, e.hire_date),
                    XMLFormat('EMP'))
    FROM employees e WHERE salary > 15000;

SELECT * FROM employee_view;

SYS_NC_ROWINFO$
-------------------------------------------------------
<?xml version="1.0"?
<EMP empno="100">
  <FNAME>Steven</FNAME>
  <LNAME>King</LNAME>
  <HIREDATE>17-JUN-03</HIREDATE>
</EMP>

<?xml version="1.0"?>
<EMP empno="101">
  <FNAME>Neena</FNAME>
  <LNAME>Kochhar</LNAME>
  <HIREDATE>21-SEP-05</HIREDATE>
</EMP>

<?xml version="1.0"?>
<EMP empno="102">
  <FNAME>Lex</FNAME>
  <LNAME>De Haan</LNAME>
  <HIREDATE>13-JAN-01</HIREDATE>
</EMP>
```

Existing relational or object-relational data can be exposed as XML data using this mechanism.

Creating XML Schema-Based XMLType Views

XML schema-based `XMLType` views are views whose data is constrained to conform to an XML schema. You can create an XML schema-based `XMLType` view in either of these ways:

- Using SQL/XML publishing functions.

  See Also: "Creating XML Schema-Based XMLType Views using SQL/XML Publishing Functions" on page 19-4

- Using object types or object views. This is convenient when you already have object types, views, and tables that you want to map to XML data.

  See Also: "Creating XML Schema-Based XMLType Views using Object Types or Object Views" on page 19-10

Creating XML Schema-Based XMLType Views using SQL/XML Publishing Functions

You can use SQL/XML publishing functions to create XML schema-based `XMLType` views in a similar way as for the non-schema-based case described in section "Creating Non-Schema-Based XMLType Views":

1. Create and register the XML schema document that contains the necessary XML structures. You do not need to annotate the XML schema to define the mapping between XML types and SQL object types.
2. Use SQL/XML publishing functions to create an `XMLType` view that conforms to the XML schema.

These two steps are illustrated in Example 19–3 and Example 19–4, respectively.

Example 19–3 Registering XML Schema emp_simple.xsd

```
BEGIN
  DBMS_XMLSCHEMA.registerSchema(
   SCHEMAURL => 'http://www.oracle.com/emp_simple.xsd',
   SCHEMADOC => '<schema xmlns="http://www.w3.org/2001/XMLSchema"
                         targetNamespace="http://www.oracle.com/emp_simple.xsd"
                         version="1.0"
                         xmlns:xdb="http://xmlns.oracle.com/xdb"
                         elementFormDefault="qualified">
                    <element name = "Employee">
                      <complexType>
                        <sequence>
                          <element name = "EmployeeId" type = "positiveInteger"/>
                          <element name = "Name" type = "string"/>
                          <element name = "Job" type = "string"/>
                          <element name = "Manager" type = "positiveInteger"/>
                          <element name = "HireDate" type = "date"/>
                          <element name = "Salary" type = "positiveInteger"/>
                          <element name = "Commission" type = "positiveInteger"/>
                          <element name = "Dept">
                            <complexType>
                              <sequence>
                                <element name = "DeptNo" type = "positiveInteger" />
                                <element name = "DeptName" type = "string"/>
                                <element name = "Location" type = "positiveInteger"/>
                              </sequence>
                            </complexType>
                          </element>
                        </sequence>
                      </complexType>
                    </element>
                  </schema>',
   LOCAL     => TRUE,
   GENTYPES  => TRUE);
END;
```

Example 19–3 assumes that you have an XML schema `emp_simple.xsd` that contains XML structures defining an employee. It registers the XML schema with the target location `http://www.oracle.com/emp_simple.xsd`.

When using SQL/XML publishing functions to generate XML schema-based content, you must specify the appropriate namespace information for all of the elements and also indicate the location of the schema using attribute `xsi:schemaLocation`. These can be specified using the `XMLAttributes` clause. Example 19–4 illustrates this.

Example 19–4 Creating an XMLType View using SQL/XML Publishing Functions

```
CREATE OR REPLACE VIEW emp_simple_xml OF XMLType
  XMLSCHEMA "http://www.oracle.com/emp_simple.xsd" ELEMENT "Employee"
   WITH OBJECT ID (XMLCast(XMLQuery('/Employee/EmployeeId/text()'
                                    PASSING OBJECT_VALUE
```

```
                                  RETURNING CONTENT)
                          AS BINARY_DOUBLE)) AS
  SELECT
    XMLElement("Employee",
               XMLAttributes(
                 'http://www.oracle.com/emp_simple.xsd' AS "xmlns" ,
                 'http://www.w3.org/2001/XMLSchema-instance' AS "xmlns:xsi",
                 'http://www.oracle.com/emp_simple.xsd
                  http://www.oracle.com/emp_simple.xsd'
                 AS "xsi:schemaLocation"),
               XMLForest(e.employee_id    AS "EmployeeId",
                         e.last_name      AS "Name",
                         e.job_id         AS "Job",
                         e.manager_id     AS "Manager",
                         e.hire_date      AS "HireDate",
                         e.salary         AS "Salary",
                         e.commission_pct AS "Commission",
                         XMLForest(
                           d.department_id   AS "DeptNo",
                           d.department_name AS "DeptName",
                           d.location_id     AS "Location") AS "Dept"))
    FROM employees e, departments d
    WHERE e.department_id = d.department_id;
```

In Example 19–4, function `XMLElement` creates XML element `Employee`. Function `XMLForest` creates the children of element `Employee`. The `XMLAttributes` clause inside `XMLElement` constructs the required XML `namespace` and schema `location` attributes, so that the XML data that is generated conforms to the XML schema of the view. The innermost call to `XMLForest` creates the children of element `department`, which is a child of element `Employee`.

By default, the XML generation functions create a non-schema-based XML instance. However, when the schema location is specified, using attribute `xsi:schemaLocation` or `xsi:noNamespaceSchemaLocation`, Oracle XML DB generates XML schema-based XML data. For `XMLType` views, as long as the names of the elements and attributes match those in the XML schema, the XML data is converted implicitly into a valid XML schema-based document. Any errors in the generated XML data are caught later, when operations such as validation or extraction operations are performed on the XML instance.

Example 19–5 queries the `XMLType` view, returning an XML result from tables `employees` and `departments`. The result of the query is shown pretty-printed, for clarity.

Example 19–5 Querying an XMLType View

```
SELECT OBJECT_VALUE AS RESULT FROM emp_simple_xml WHERE ROWNUM < 2;

RESULT
----------------------------------------------------------------------
<Employee xmlns="http://www.oracle.com/emp_simple.xsd"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://www.oracle.com/emp_simple.xsd
                              http://www.oracle.com/emp_simple.xsd">
  <EmployeeId>200</EmployeeId>
  <Name>Whalen</Name>
  <Job>AD_ASST</Job>
  <Manager>101</Manager>
  <HireDate>2003-09-17</HireDate>
  <Salary>4400</Salary>
```

```
  <Dept>
    <DeptNo>10</Deptno>
    <DeptName>Administration</DeptName>
    <Location>1700</Location>
  </Dept>
</Employee>
```

Using Namespaces with SQL/XML Publishing Functions

If you have complex XML schemas involving namespaces, you must use the partially escaped mapping provided by the SQL/XML publishing functions and create elements with appropriate namespaces and prefixes.

The query in Example 19–6 creates XML instances that have the correct namespace, prefixes, and target schema location. It can be used as the query in the definition of view `emp_simple_xml`.

Example 19–6 Using Namespace Prefixes with SQL/XML Publishing Functions

```
SELECT XMLElement("ipo:Employee",
          XMLAttributes('http://www.oracle.com/emp_simple.xsd' AS "xmlns:ipo",
                        'http://www.oracle.com/emp_simple.xsd
                         http://www.oracle.com/emp_simple.xsd' AS "xmlns:xsi"),
            XMLForest(e.employee_id                     AS "ipo:EmployeeId",
                      e.last_name                       AS "ipo:Name",
                      e.job_id                          AS "ipo:Job",
                      e.manager_id                      AS "ipo:Manager",
                      TO_CHAR(e.hire_date,'YYYY-MM-DD') AS "ipo:HireDate",
                      e.salary                          AS "ipo:Salary",
                      e.commission_pct                  AS "ipo:Commission",
                 XMLForest(d.department_id   AS "ipo:DeptNo",
                           d.department_name AS "ipo:DeptName", d.location_id
       AS "ipo:Location") AS "ipo:Dept"))
       FROM employees e, departments d
       WHERE e.department_id = d.department_id AND d.department_id = 20;
BEGIN
  -- Delete schema if it already exists (else error)
  DBMS_XMLSCHEMA.deleteSchema('emp-noname.xsd', 4);
END;

XMLELEMENT("IPO:EMPLOYEE",XMLATTRIBUTES('HTTP://WWW.ORACLE.COM/EMP_SIMPLE.XSD'AS
--------------------------------------------------------------------------------
<ipo:Employee
xmlns:ipo="http://www.oracle.com/emp_simple.xsd"
 xmlns:xsi="http://www.oracle.com/emp_simple.xsd
 http://www.oracle.com/emp_simple.xsd">
<ipo:EmployeeId>201</ipo:EmployeeId><ipo:Name>Hartstein</ipo:Name>
<ipo:Job>MK_MAN</ipo:Job><ipo:Manager>100</ipo:Manager>
<ipo:HireDate>2004-02-17</ipo:HireDate><ipo:Salary>13000</ipo:Salary>
<ipo:Dept><ipo:DeptNo>20</ipo:DeptNo><ipo:DeptName>Marketing</ipo:DeptName>
<ipo:Location>1800</ipo:Location></ipo:Dept></ipo:Employee>
<ipo:Employee xmlns:ipo="http://www.oracle.com/emp_simple.xsd"
 xmlns:xsi="http://www.oracle.com/emp_simple.xsd
 http://www.oracle.com/emp_simple.xsd"><ipo:EmployeeId>202</ipo:EmployeeId>
<ipo:Name>Fay</ipo:Name><ipo:Job>MK_REP</ipo:Job><ipo:Manager>201</ipo:Manager>
<ipo:HireDate>2005-08-17</ipo:HireDate><ipo:Salary>6000</ipo:Salary>
<ipo:Dept><ipo:DeptNo>20</ipo:Dept
No><ipo:DeptName>Marketing</ipo:DeptName><ipo:Location>1800</ipo:Location>
</ipo:Dept>
</ipo:Employee>
```

If the XML schema had no target namespace, then you could use attribute `xsi:noNamespaceSchemaLocation` to indicate that. Example 19–7 shows such an XML schema.

Example 19–7 XML Schema with No Target Namespace

```
BEGIN
  DBMS_XMLSCHEMA.registerSchema(
    SCHEMAURL => 'emp-noname.xsd',
    SCHEMADOC => '<schema xmlns="http://www.w3.org/2001/XMLSchema"
                          xmlns:xdb="http://xmlns.oracle.com/xdb">
                    <element name = "Employee">
                      <complexType>
                        <sequence>
                          <element name = "EmployeeId" type = "positiveInteger"/>
                          <element name = "Name" type = "string"/>
                          <element name = "Job" type = "string"/>
                          <element name = "Manager" type = "positiveInteger"/>
                          <element name = "HireDate" type = "date"/>
                          <element name = "Salary" type = "positiveInteger"/>
                          <element name = "Commission" type = "positiveInteger"/>
                          <element name = "Dept">
                            <complexType>
                              <sequence>
                                <element name = "DeptNo" type = "positiveInteger" />
                                <element name = "DeptName" type = "string"/>
                                <element name = "Location" type = "positiveInteger"/>
                              </sequence>
                            </complexType>
                          </element>
                        </sequence>
                      </complexType>
                    </element>
                  </schema>',
    LOCAL     => TRUE,
    GENTYPES  => TRUE);
END;
```

Example 19–8 creates a view that conforms to the XML schema in Example 19–7. The `XMLAttributes` clause creates an XML element that contains the `noNamespace` schema location attribute.

Example 19–8 Creating a View for an XML Schema with No Target Namespace

```
CREATE OR REPLACE VIEW emp_xml OF XMLType
     XMLSCHEMA "emp-noname.xsd" ELEMENT "Employee"
     WITH OBJECT ID (XMLCast(XMLQuery('/Employee/EmployeeId/text()'
                                      PASSING OBJECT_VALUE
                                      RETURNING CONTENT)
                             AS BINARY_DOUBLE)) AS
     SELECT XMLElement(
       "Employee",
       XMLAttributes('http://www.w3.org/2001/XMLSchema-instance' AS "xmlns:xsi",
                     'emp-noname.xsd' AS "xsi:noNamespaceSchemaLocation"),
       XMLForest(e.employee_id    AS "EmployeeId",
                 e.last_name      AS "Name",
                 e.job_id         AS "Job",
                 e.manager_id     AS "Manager",
                 e.hire_date      AS "HireDate",
                 e.salary         AS "Salary",
```

```
                                e.commission_pct AS "Commission",
                                XMLForest(d.department_id   AS "DeptNo",
                                          d.department_name AS "DeptName",
                                          d.location_id     AS "Location") AS "Dept"))
                        FROM employees e, departments d
                        WHERE e.department_id = d.department_id;
```

Example 19–9 creates view `dept_xml`, which conforms to XML schema `dept.xsd`.

Example 19–9 Using SQL/XML Functions in XML Schema-Based XMLType Views

```
BEGIN
  -- Delete schema if it already exists (else error)
  DBMS_XMLSCHEMA.deleteSchema('http://www.oracle.com/dept.xsd', 4);
END;
/

BEGIN
  DBMS_XMLSCHEMA.registerSchema(
    SCHEMAURL => 'http://www.oracle.com/dept.xsd',
    SCHEMADOC => '<schema xmlns="http://www.w3.org/2001/XMLSchema"
                          targetNamespace="http://www.oracle.com/dept.xsd"
                          version="1.0"
                          xmlns:xdb="http://xmlns.oracle.com/xdb"
                          elementFormDefault="qualified">
                    <element name = "Department">
                      <complexType>
                        <sequence>
                          <element name = "DeptNo" type = "positiveInteger"/>
                          <element name = "DeptName" type = "string"/>
                          <element name = "Location" type = "positiveInteger"/>
                          <element name = "Employee" maxOccurs = "unbounded">
                            <complexType>
                              <sequence>
                                <element name = "EmployeeId" type = "positiveInteger"/>
                                <element name = "Name" type = "string"/>
                                <element name = "Job" type = "string"/>
                                <element name = "Manager" type = "positiveInteger"/>
                                <element name = "HireDate" type = "date"/>
                                <element name = "Salary" type = "positiveInteger"/>
                                <element name = "Commission" type = "positiveInteger"/>
                             </sequence>
                            </complexType>
                          </element>
                        </sequence>
                      </complexType>
                    </element>
                  </schema>',
    LOCAL     => TRUE,
    GENTYPES  => FALSE);
  END;
/

CREATE OR REPLACE VIEW dept_xml OF XMLType
  XMLSCHEMA "http://www.oracle.com/dept.xsd" ELEMENT "Department"
  WITH OBJECT ID (XMLCast(XMLQuery('/Department/DeptNo'
                                   PASSING OBJECT_VALUE RETURNING CONTENT)
                          AS BINARY_DOUBLE)) AS
  SELECT XMLElement(
    "Department",
```

```
    XMLAttributes(
      'http://www.oracle.com/emp.xsd' AS "xmlns" ,
      'http://www.w3.org/2001/XMLSchema-instance' AS "xmlns:xsi",
      'http://www.oracle.com/dept.xsd
       http://www.oracle.com/dept.xsd' AS "xsi:schemaLocation"),
    XMLForest(d.department_id "DeptNo",
              d.department_name "DeptName",
              d.location_id "Location"),
    (SELECT XMLagg(
              XMLElement("Employee",
                         XMLForest(
                           e.employee_id  "EmployeeId",
                           e.last_name "Name",
                           e.job_id "Job",
                           e.manager_id "Manager",
                           to_char(e.hire_date,'YYYY-MM-DD') "Hiredate",
                           e.salary "Salary",
                           e.commission_pct "Commission")))
       FROM employees e
       WHERE e.department_id = d.department_id))
     FROM departments d;
```

This is the `XMLType` instance that results:

```
SELECT OBJECT_VALUE AS result FROM dept_xml WHERE ROWNUM < 2;

RESULT
----------------------------------------------------------------
<Department
    xmlns="http://www.oracle.com/emp.xsd"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.oracle.com/dept.xsd
                        http://www.oracle.com/dept.xsd">
  <DeptNo>10</DeptNo>
  <DeptName>Administration</DeptName>
  <Location>1700</Location>
  <Employee>
    <EmployeeId>200</EmployeeId>
    <Name>Whalen</Name>
    <Job>AD_ASST</Job>
    <Manager>101</Manager>
    <Hiredate>2003-09-17</Hiredate>
    <Salary>4400</Salary>
  </Employee>
</Department>
```

Creating XML Schema-Based XMLType Views using Object Types or Object Views

To create an XML schema-based `XMLType` view from object types or object views, do the following:

1. Create the object types, if they do not yet exist.
2. Create and then register the XML schema, annotating it to define the mapping between XML types and SQL object types and attributes.

   You can annotate the XML schema *before* registering it. You typically do this when you wrap existing data to create an `XMLType` view.

   See: Chapter 7, "XML Schema Storage and Query: Basic"

You can use PL/SQL functions `DBMS_XMLSchema.generateSchema` and `DBMS_XMLSchema.generateSchemas` to generate the default XML mapping for specified object types. The generated XML schema has the requisite annotations `SQLType`, `SQLSchema`, and so on. When such an XML schema document is registered, the following validation can occur:

- `SQLType` for attributes or elements based on `simpleType`. The SQL type must be compatible with the XML type of the corresponding `XMLType` data. For example, an XML `string` data type can be mapped only to a `VARCHAR2` or a Large Object (LOB) data type.
- `SQLType` specified for elements based on `complexType`. This is either a LOB or an object type whose structure must be compatible with the declaration of the `complexType`, that is, the object type must have the correct number of attributes with the correct data types.

3. Create the `XMLType` view, specifying the XML schema URL and the root element name. The query defining the view first constructs the object instances and then converts them to XML.
   a. Create an object view.
   b. Create an `XMLType` view over the object view.

The following sections present examples of creating XML schema-based `XMLType` views using object types or object views. They are based on relational tables that contain employee and department data.

- "Creating XMLType Employee View, with Nested Department Information"
- "Creating XMLType Department View, with Nested Employee Information"

The same relational data is used to create each of two `XMLType` views. In the employee view, `emp_xml`, the XML document describes an employee, with the employee's department as nested information. In the department view, `dept_xml`, the XML data describes a department, with the department's employees as nested information.

Creating XMLType Employee View, with Nested Department Information

This section describes how to create `XMLType` view `emp_xml` based on object views. For the last step, there are two *alternatives*:

- "Step 3a. Create XMLType View emp_xml using Object Type emp_t" – create `XMLType` view `emp_xml` using object type `emp_t`
- "Step 3b. Create XMLType View emp_xml using Object View emp_v" – create `XMLType` view `emp_xml` using object view `emp_v`

Step 1. Create Object Types Example 19–10 creates the object types used in the other steps.

Example 19–10 Creating Object Types for Schema-Based XMLType Views

```
CREATE TYPE dept_t AS OBJECT
      (deptno NUMBER(4),
       dname  VARCHAR2(30),
       loc    NUMBER(4));
/

CREATE TYPE emp_t AS OBJECT
      (empno     NUMBER(6),
       ename     VARCHAR2(25),
```

```
        job       VARCHAR2(10),
        mgr       NUMBER(6),
        hiredate  DATE,
        sal       NUMBER(8,2),
        comm      NUMBER(2,2),
        dept      dept_t);
/
```

Step 2. Create and Register XML Schema emp_complex.xsd You can create an XML schema manually, or you can use package `DBMS_XMLSCHEMA` to generate an XML schema automatically from existing object types, as shown in Example 19–11.

Example 19–11 Generating an XML Schema with DBMS_XMLSCHEMA.GENERATESCHEMA

```
SELECT DBMS_XMLSCHEMA.generateSchema('HR','EMP_T') AS result FROM DUAL;
```

Example 19–11 generates the XML schema for type `emp_t`. You can supply various arguments to PL/SQL function `DBMS_XMLSCHEMA.generateSchemas`, to add namespaces, and so on. You can also edit the XML schema to change the default mappings that are generated. Function `generateSchemas` generates a list of XML schemas, one for each SQL database schema that is referenced by the object type and its object attributes.

Example 19–12 shows how to register XML schema `emp_complex.xsd`, which specifies how XML elements and attributes are mapped to corresponding object attributes in the object types (the `xdb:SQLType` annotations).

Example 19–12 Registering XML Schema emp_complex.xsd

```
BEGIN
  -- Delete schema if it already exists (else error)
  DBMS_XMLSCHEMA.deleteSchema('http://www.oracle.com/emp_complex.xsd', 4);
END;
/

COMMIT;

BEGIN
  DBMS_XMLSCHEMA.registerSchema(
    SCHEMAURL => 'http://www.oracle.com/emp_complex.xsd',
    SCHEMADOC => '<?xml version="1.0"?>
                  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                              xmlns:xdb="http://xmlns.oracle.com/xdb"
                              xsi:schemaLocation="http://xmlns.oracle.com/xdb
                                                  http://xmlns.oracle.com/xdb/XDBSchema.xsd">
                    <xsd:element name="Employee" type="EMP_TType" xdb:SQLType="EMP_T"
                                 xdb:SQLSchema="HR"/>
                    <xsd:complexType name="EMP_TType" xdb:SQLType="EMP_T" xdb:SQLSchema="HR"
                                     xdb:maintainDOM="false">
                      <xsd:sequence>
                        <xsd:element name="EMPNO" type="xsd:double" xdb:SQLName="EMPNO"
                                     xdb:SQLType="NUMBER"/>
                        <xsd:element name="ENAME" xdb:SQLName="ENAME" xdb:SQLType="VARCHAR2">
                          <xsd:simpleType>
                            <xsd:restriction base="xsd:string">
                              <xsd:maxLength value="25"/>
                            </xsd:restriction>
                          </xsd:simpleType>
```

```
                    </xsd:element>
                    <xsd:element name="JOB" xdb:SQLName="JOB" xdb:SQLType="VARCHAR2">
                      <xsd:simpleType>
                        <xsd:restriction base="xsd:string">
                          <xsd:maxLength value="10"/>
                        </xsd:restriction>
                      </xsd:simpleType>
                    </xsd:element>
                    <xsd:element name="MGR" type="xsd:double" xdb:SQLName="MGR"
                                 xdb:SQLType="NUMBER"/>
                    <xsd:element name="HIREDATE" type="xsd:date" xdb:SQLName="HIREDATE"
                                 xdb:SQLType="DATE"/>
                    <xsd:element name="SAL" type="xsd:double" xdb:SQLName="SAL"
                                 xdb:SQLType="NUMBER"/>
                    <xsd:element name="COMM" type="xsd:double" xdb:SQLName="COMM"
                                 xdb:SQLType="NUMBER"/>
                    <xsd:element name="DEPT" type="DEPT_TType" xdb:SQLName="DEPT"
                                 xdb:SQLSchema="HR" xdb:SQLType="DEPT_T"/>
                  </xsd:sequence>
                </xsd:complexType>
                <xsd:complexType name="DEPT_TType" xdb:SQLType="DEPT_T" xdb:SQLSchema="HR"
                                 xdb:maintainDOM="false">
                  <xsd:sequence>
                    <xsd:element name="DEPTNO" type="xsd:double" xdb:SQLName="DEPTNO"
                                 xdb:SQLType="NUMBER"/>
                    <xsd:element name="DNAME" xdb:SQLName="DNAME" xdb:SQLType="VARCHAR2">
                      <xsd:simpleType>
                        <xsd:restriction base="xsd:string">
                          <xsd:maxLength value="30"/>
                        </xsd:restriction>
                      </xsd:simpleType>
                    </xsd:element>
                   <xsd:element name="LOC" type="xsd:double" xdb:SQLName="LOC"
                                xdb:SQLType="NUMBER"/>
                  </xsd:sequence>
                </xsd:complexType>
               </xsd:schema>',
    LOCAL     => TRUE,
    GENTYPES  => FALSE);
END;
/
```

Example 19–12 registers the XML schema using the target location `http://www.oracle.com/emp_complex.xsd`.

Step 3a. Create XMLType View emp_xml using Object Type emp_t Example 19–13 creates an `XMLType` view using object type `emp_t`.

Example 19–13 Creating XMLType View emp_xml

```
CREATE OR REPLACE VIEW emp_xml OF XMLType
  XMLSCHEMA "http://www.oracle.com/emp_complex.xsd"
  ELEMENT "Employee"
    WITH OBJECT ID (XMLCast(XMLQuery('/Employee/EMPNO'
                                     PASSING OBJECT_VALUE RETURNING CONTENT)
                            AS BINARY_DOUBLE)) AS
  SELECT emp_t(e.employee_id, e.last_name, e.job_id, e.manager_id, e.hire_date,
               e.salary, e.commission_pct,
               dept_t(d.department_id, d.department_name, d.location_id))
    FROM employees e, departments d
```

```
    WHERE e.department_id = d.department_id;
```

Example 19–13 uses SQL/XML function `XMLCast` in the `OBJECT ID` clause to convert the XML employee number to SQL data type `BINARY_DOUBLE`.

Step 3b. Create XMLType View emp_xml using Object View emp_v Example 19–14 creates an `XMLType` view based on an object view.

Example 19–14 Creating an Object View and an XMLType View on the Object View

```
CREATE OR REPLACE VIEW emp_v OF emp_t WITH OBJECT ID (empno) AS
  SELECT emp_t(e.employee_id, e.last_name, e.job_id, e.manager_id, e.hire_date,
               e.salary, e.commission_pct,
               dept_t(d.department_id, d.department_name, d.location_id))
    FROM employees e, departments d
    WHERE e.department_id = d.department_id;

CREATE OR REPLACE VIEW emp_xml OF XMLType
  XMLSCHEMA "http://www.oracle.com/emp_complex.xsd" ELEMENT "Employee"
  WITH OBJECT ID DEFAULT AS
  SELECT VALUE(p) FROM emp_v p;
```

Creating XMLType Department View, with Nested Employee Information

This section describes how to create `XMLType` view `dept_xml`. Each department in this view contains nested employee information. For the last step, there are two *alternatives*:

- "Step 3a. Create XMLType View dept_xml using Object Type dept_t" – create `XMLType` view `dept_xml` using the object type for a department, `dept_t`
- "Step 3b. Create XMLType View dept_xml using Relational Data Directly" – create `XMLType` view `dept_xml` using relational data directly

Step 1. Create Object Types Example 19–15 creates the object types used in the other steps.

Example 19–15 Creating Object Types

```
CREATE TYPE emp_t AS OBJECT (empno    NUMBER(6),
                             ename    VARCHAR2(25),
                             job      VARCHAR2(10),
                             mgr      NUMBER(6),
                             hiredate DATE,
                             sal      NUMBER(8,2),
                             comm     NUMBER(2,2));
/

CREATE OR REPLACE TYPE emplist_t AS TABLE OF emp_t;
/

CREATE TYPE dept_t AS OBJECT (deptno NUMBER(4),
                              dname  VARCHAR2(30),
                              loc    NUMBER(4),
                              emps   emplist_t);
/
```

Step 2. Register XML Schema dept_complex.xsd You can either use a pre-existing XML schema or generate an XML schema from the object type with function `DBMS_XMLSCHEMA.generateSchema` or `DBMS_XMLSCHEMA.generateSchemas` (see

Example 19–11 on page 19-12). Example 19–16 registers the XML schema `dept_complex.xsd`.

Example 19–16 Registering XML Schema dept_complex.xsd

```
BEGIN
  -- Delete schema if it already exists (else error)
  DBMS_XMLSCHEMA.deleteSchema('http://www.oracle.com/dept_complex.xsd', 4);
END;
/

BEGIN
  DBMS_XMLSCHEMA.registerSchema(
    SCHEMAURL => 'http://www.oracle.com/dept_complex.xsd',
    SCHEMADOC => '<?xml version="1.0"?>
                  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                              xmlns:xdb="http://xmlns.oracle.com/xdb"
                              xsi:schemaLocation="http://xmlns.oracle.com/xdb
                                                  http://xmlns.oracle.com/xdb/XDBSchema.xsd">
                    <xsd:element name="Department" type="DEPT_TType" xdb:SQLType="DEPT_T"
                                 xdb:SQLSchema="HR"/>
                    <xsd:complexType name="DEPT_TType" xdb:SQLType="DEPT_T" xdb:SQLSchema="HR"
                                     xdb:maintainDOM="false">
                      <xsd:sequence>
                        <xsd:element name="DEPTNO" type="xsd:double" xdb:SQLName="DEPTNO"
                                     xdb:SQLType="NUMBER"/>
                        <xsd:element name="DNAME" xdb:SQLName="DNAME" xdb:SQLType="VARCHAR2">
                          <xsd:simpleType>
                            <xsd:restriction base="xsd:string">
                              <xsd:maxLength value="30"/>
                            </xsd:restriction>
                          </xsd:simpleType>
                        </xsd:element>
                        <xsd:element name="LOC" type="xsd:double" xdb:SQLName="LOC"
                                     xdb:SQLType="NUMBER"/>
                        <xsd:element name="EMPS" type="EMP_TType" maxOccurs="unbounded"
                                     minOccurs="0" xdb:SQLName="EMPS"
                                     xdb:SQLCollType="EMPLIST_T" xdb:SQLType="EMP_T"
                                     xdb:SQLSchema="HR" xdb:SQLCollSchema="HR"/>
                      </xsd:sequence>
                    </xsd:complexType>
                    <xsd:complexType name="EMP_TType" xdb:SQLType="EMP_T" xdb:SQLSchema="HR"
                                     xdb:maintainDOM="false">
                      <xsd:sequence>
                        <xsd:element name="EMPNO" type="xsd:double" xdb:SQLName="EMPNO"
                                     xdb:SQLType="NUMBER"/>
                        <xsd:element name="ENAME" xdb:SQLName="ENAME" xdb:SQLType="VARCHAR2">
                          <xsd:simpleType>
                            <xsd:restriction base="xsd:string">
                              <xsd:maxLength value="25"/>
                            </xsd:restriction>
                          </xsd:simpleType>
                        </xsd:element>
                        <xsd:element name="JOB" xdb:SQLName="JOB" xdb:SQLType="VARCHAR2">
                          <xsd:simpleType>
                            <xsd:restriction base="xsd:string">
                              <xsd:maxLength value="10"/>
                            </xsd:restriction>
                          </xsd:simpleType>
```

```
                    </xsd:element>
                    <xsd:element name="MGR" type="xsd:double" xdb:SQLName="MGR"
                                 xdb:SQLType="NUMBER"/>
                    <xsd:element name="HIREDATE" type="xsd:date" xdb:SQLName="HIREDATE"
                                 xdb:SQLType="DATE"/>
                    <xsd:element name="SAL" type="xsd:double" xdb:SQLName="SAL"
                                 xdb:SQLType="NUMBER"/>
                    <xsd:element name="COMM" type="xsd:double" xdb:SQLName="COMM"
                                 xdb:SQLType="NUMBER"/>
                  </xsd:sequence>
                 </xsd:complexType>
                </xsd:schema>',
    LOCAL     => TRUE,
    GENTYPES  => FALSE);
END;
/
```

Step 3a. Create XMLType View dept_xml using Object Type dept_t Example 19–17 creates `XMLType` view `dept_xml` using object type `dept_t`.

Example 19–17 Creating XMLType View dept_xml using Object Type dept_t

```
CREATE OR REPLACE VIEW dept_xml OF XMLType
  XMLSCHEMA "http://www.oracle.com/dept_complex.xsd" ELEMENT "Department"
  WITH OBJECT ID (XMLCast(XMLQuery('/Department/DEPTNO'
                                   PASSING OBJECT_VALUE RETURNING CONTENT)
                            AS BINARY_DOUBLE)) AS
  SELECT dept_t(d.department_id, d.department_name, d.location_id,
                cast(MULTISET
                     (SELECT emp_t(e.employee_id, e.last_name, e.job_id,
                                   e.manager_id, e.hire_date,
                                   e.salary, e.commission_pct)
                        FROM employees e WHERE e.department_id = d.department_id)
                     AS emplist_t))
    FROM departments d;
```

Step 3b. Create XMLType View dept_xml using Relational Data Directly Alternatively, you can use SQL/XML publishing functions to create `XMLType` view `dept_xml` from the relational tables without using object type `dept_t`. Example 19–18 illustrates this.

Example 19–18 Creating XMLType View dept_xml using Relational Data Directly

```
CREATE OR REPLACE VIEW dept_xml OF XMLType
  XMLSCHEMA "http://www.oracle.com/dept_complex.xsd" ELEMENT "Department"
  WITH OBJECT ID (XMLCast(XMLQuery('/Department/DEPTNO'
                                   PASSING OBJECT_VALUE RETURNING CONTENT)
                            AS BINARY_DOUBLE)) AS
  SELECT
    XMLElement(
      "Department",
      XMLAttributes('http://www.oracle.com/dept_complex.xsd' AS "xmlns",
                    'http://www.w3.org/2001/XMLSchema-instance' AS "xmlns:xsi",
                    'http://www.oracle.com/dept_complex.xsd
                     http://www.oracle.com/dept_complex.xsd'
                      AS "xsi:schemaLocation"),
      XMLForest(d.department_id "DeptNo", d.department_name "DeptName",
                d.location_id "Location"),
      (SELECT XMLAgg(XMLElement("Employee",
                                XMLForest(e.employee_id "EmployeeId",
                                          e.last_name "Name",
```

```
                                        e.job_id "Job",
                                        e.manager_id "Manager",
                                        e.hire_date "Hiredate",
                                        e.salary "Salary",
                                        e.commission_pct "Commission")))
                        FROM employees e WHERE e.department_id = d.department_id))
    FROM departments d;
```

> **Note:** XML schema and element information must be specified at the view level, because the `SELECT` list could arbitrarily construct XML of a different XML schema from the underlying table.

Creating XMLType Views from XMLType Tables

An `XMLType` view can be created on an `XMLType` table, for example, to transform the XML data or to restrict the rows returned.

Example 19–19 creates an `XMLType` view by restricting the rows included from an underlying `XMLType` table. It uses XML schema `dept_complex.xsd` to create the underlying table—see "Creating XMLType Department View, with Nested Employee Information".

Example 19–19 Creating an XMLType View by Restricting Rows from an XMLType Table

```
CREATE TABLE dept_xml_tab OF XMLType
    XMLSchema "http://www.oracle.com/dept_complex.xsd" ELEMENT "Department"
    NESTED TABLE XMLDATA."EMPS" STORE AS dept_xml_tab_tab1;

CREATE OR REPLACE VIEW dallas_dept_view OF XMLType
    XMLSchema "http://www.oracle.com/dept.xsd" ELEMENT "Department"
        AS SELECT OBJECT_VALUE FROM dept_xml_tab
           WHERE XMLCast(XMLQuery('/Department/LOC'
                                  PASSING OBJECT_VALUE RETURNING CONTENT)
                          AS VARCHAR2(20))
                 = 'DALLAS';

CREATE OR REPLACE VIEW dallas_dept_view OF XMLType
```

Here, `dallas_dept_view` restricts the `XMLType` table rows to those departments whose location is Dallas.

Example 19–20 shows how you can create an `XMLType` view by transforming XML data using a style sheet.

Example 19–20 Creating an XMLType View by Transforming an XMLType Table

```
CREATE OR REPLACE VIEW hr_po_tab OF XMLType
  ELEMENT "PurchaseOrder" WITH OBJECT ID DEFAULT AS
  SELECT XMLtransform(OBJECT_VALUE, x.col1)
    FROM purchaseorder p, xsl_tab x;
```

Referencing XMLType View Objects using SQL Function REF

You can reference an `XMLType` view object using SQL function `ref`:

```
SELECT ref(d) FROM dept_xml_tab d;
```

An `XMLType` view reference is based on one of the following object IDs:

- System-generated OID — for views on XMLType tables or object views
- Primary key based OID -- for views with OBJECT ID expressions

These REFs can be used to fetch OCIXMLType instances in the OCI Object cache, or they can be used in SQL queries. These REFs act the same as REFs to object views.

DML (Data Manipulation Language) on XMLType Views

A given XMLType view might not be implicitly updatable. In that case you must write INSTEAD-OF TRIGGERS to handle all data manipulation (DML). One way to determine whether a given XMLType view is implicitly updatable is to query the view to see whether it is based on an object view or an object constructor that is itself inherently updatable. Example 19–21 illustrates this.

Example 19–21 Determining Whether an XMLType View is Implicitly Updatable

```
CREATE TYPE dept_t AS OBJECT
      (deptno NUMBER(4),
       dname  VARCHAR2(30),
       loc    NUMBER(4));
/

BEGIN
  -- Delete schema if it already exists (else error)
  DBMS_XMLSCHEMA.deleteSchema('http://www.oracle.com/dept.xsd', 4);
END;
/
COMMIT;

BEGIN
  DBMS_XMLSCHEMA.registerSchema(
    SCHEMAURL => 'http://www.oracle.com/dept_t.xsd',
    SCHEMADOC => '<?xml version="1.0"?>
                  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                              xmlns:xdb="http://xmlns.oracle.com/xdb"
                              xsi:schemaLocation="http://xmlns.oracle.com/xdb
                                                  http://xmlns.oracle.com/xdb/XDBSchema.xsd">
                    <xsd:element name="Department" type="DEPT_TType" xdb:SQLType="DEPT_T"
                                 xdb:SQLSchema="HR"/>
                    <xsd:complexType name="DEPT_TType" xdb:SQLType="DEPT_T" xdb:SQLSchema="HR"
                                     xdb:maintainDOM="false">
                      <xsd:sequence>
                        <xsd:element name="DEPTNO" type="xsd:double" xdb:SQLName="DEPTNO"
                                     xdb:SQLType="NUMBER"/>
                        <xsd:element name="DNAME" xdb:SQLName="DNAME" xdb:SQLType="VARCHAR2">
                          <xsd:simpleType>
                            <xsd:restriction base="xsd:string">
                              <xsd:maxLength value="30"/>
                            </xsd:restriction>
                          </xsd:simpleType>
                        </xsd:element>
                        <xsd:element name="LOC" type="xsd:double" xdb:SQLName="LOC"
                                     xdb:SQLType="NUMBER"/>
                      </xsd:sequence>
                    </xsd:complexType>
                  </xsd:schema>',
    LOCAL     => TRUE,
    GENTYPES  => FALSE);
```

```
END;
/

CREATE OR REPLACE VIEW dept_xml of XMLType
  XMLSchema "http://www.oracle.com/dept_t.xsd" element "Department"
  WITH OBJECT ID (XMLCast(XMLQuery('/Department/DEPTNO'
                                   PASSING OBJECT_VALUE RETURNING CONTENT)
                          AS BINARY_DOUBLE)) AS
  SELECT dept_t(d.department_id, d.department_name, d.location_id)
    FROM departments d;

INSERT INTO dept_xml
  VALUES (
    XMLType.createXML(
      '<Department
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:noNamespaceSchemaLocation="http://www.oracle.com/dept_t.xsd" >
         <DEPTNO>300</DEPTNO>
         <DNAME>Processing</DNAME>
         <LOC>1700</LOC>
       </Department>'));

UPDATE dept_xml d
 SET d.OBJECT_VALUE = updateXML(d.OBJECT_VALUE, '/Department/DNAME/text()',
                                'Shipping')
   WHERE XMLExists('/Department[DEPTNO=300]' PASSING OBJECT_VALUE);
```

20

Accessing Data Through URIs

This chapter describes how to generate and store URLs in the database and how to retrieve data pointed to by those URLs. Three kinds of URIs are discussed:

- DBUris – addresses to relational data in the database
- XDBUris – addresses to data in Oracle XML DB Repository
- HTTPUris – Web addresses that use the Hyper Text Transfer Protocol (HTTP(S))

This chapter contains these topics:

- Overview of Oracle XML DB URL Features
- URIs and URLs
- URIType and its Subtypes
- Accessing Data using URIType Instances
- XDBUris: Pointers to Repository Resources
- DBUris: Pointers to Database Data
- Creating New Subtypes of URIType using Package URIFACTORY
- SYS_DBURIGEN SQL Function
- DBUriServlet

Overview of Oracle XML DB URL Features

The two main features described in this chapter are these:

- *Using paths as an indirection mechanism* – You can store a path in the database and then access its target *indirectly* by referring to the path. The paths in question are various kinds of Uniform Resource Identifier (URI).
- *Using paths that target database data to produce XML documents* – One kind of URI that you can use for indirection in particular, a *DBUri*, provides a convenient XPath notation for addressing *database data*. You can use a DBUri to construct an *XML document* that contains database data and whose structure reflects the database structure.

URIs and URLs

In developing Web-based XML applications, you often refer to data located on a network using **Uniform Resource Identifiers**, or **URI**s. A **URL**, or **Uniform Resource Locator**, is a URI that accesses an object using an Internet protocol.

A URI has two parts, separated by a number sign (**#**):

- A URL part, that identifies a document.
- A fragment part, that identifies a fragment within the document. The notation for the fragment depends on the document type. For HTML documents, it is an anchor name. For XML documents, it is an XPath expression.

These are typical URIs:

- *For HTML* – `http://www.example.com/document1#some_anchor`, where `some_anchor` is a named anchor in the HTML document.
- *For XML* – `http://www.example.com/xml_doc#/po/cust/custname`, where:
  - `http://www.example.com/xml_doc` identifies the location of the XML document.
  - `/po/cust/custname` identifies a fragment within the document. This portion is defined by the W3C XPointer recommendation.

    See Also:

    - `http://www.w3.org/2002/ws/Activity.html` an explanation of HTTP(S) URL notation
    - `http://www.w3.org/TR/xpath` for an explanation of the XML XPath notation
    - `http://www.w3.org/TR/xptr/` for an explanation of the XML XPointer notation
    - `http://xml.coverpages.org/xmlMediaMIME.html` for a discussion of MIME types

URIType and its Subtypes

Oracle XML DB can represent paths of various kinds as database objects. These are the available path object types:

- **`HTTPURIType`** – An object of this type is called an **HTTPUri** and represents a URL that begins with **`http://`**. With `HTTPURIType`, you can create objects that represent links to remote *Web pages* (or files) and retrieve those Web pages by calling object methods. Applications using `HTTPUriType` must have the proper access privileges. `HTTPUriType` implements the Hyper Text Transfer Protocol (HTTP(S)) for accessing remote Web pages. `HTTPURIType` uses package `UTL_HTTP` to fetch data, so session settings and access control for this package can also be used to influence HTTP fetches.

  See Also:

  - "HTTPURIType PL/SQL Method GETCONTENTTYPE()" on page 20-5
  - *Oracle Database Security Guide* for information about managing fine-grained access to external network services

- **`DBURIType`** – An object of this type is called a **DBUri** and represents a URI that targets database data – a table, one or more rows, or a single column. With `DBURIType`, you can create objects that represent links to *database data*, and retrieve such data *as XML* by calling object methods. A DBUri uses a simple form

of XPath expression as its URI syntax – for example, the following XPath expression is a DBUri reference to the row of table `HR.employees` where column `first_name` has value `Jack`:

```
/HR/EMPLOYEES/ROW[FIRST_NAME="Jack"]
```

See Also : DBUris: Pointers to Database Data on page 20-12

- **`XDBURIType`** – An object of this type is called an **XDBUri**, and represents a URI that targets a resource in Oracle XML DB Repository. With `XDBURIType`, you can create objects that represent links to *repository resources*, and retrieve all or part of any resource by calling object methods. The URI syntax for an XDBUri is a repository resource address optionally followed by an XPath expression. For example, `/public/hr/doc1.xml#/purchaseOrder/lineItem` is an XDBUri reference to the `lineItem` child element of the root element `purchaseOrder` in repository file `doc1.xml` in folder `/public/hr`.

  See Also : XDBUris: Pointers to Repository Resources on page 20-10

Each of these object types is derived from an *abstract* object type, **`URIType`**. As an abstract type, it has *no* instances (objects). Only its subtypes have instances.

Type `URIType` provides the following features:

- *Unified access to data stored inside and outside the server.* Because you can use `URIType` values to store pointers to HTTP(S) and DBUris, you can create queries and indexes without worrying about where the data resides.
- *Mapping of URIs in XML Documents to Database Columns.* When an XML document is broken up and stored in object-relational tables and columns, any URIs contained in the document are mapped to database columns of the appropriate `URIType` subtype.

You can reference data stored in relational columns and expose it to the external world using URIs. Oracle Database provides a standard servlet, DBUriServlet, that interprets DBUris. It also provides PL/SQL package `UTL_HTTP` and Java class `java.net.URL`, which you can use to fetch URL references.

`URIType` columns can be indexed natively in Oracle Database using Oracle Text – no special data store is needed.

See Also:

- "Creating New Subtypes of URIType using Package URIFACTORY" on page 20-20 for information about defining new `URIType` subtypes
- Chapter 6, "Indexing XMLType Data"

DBUris and XDBUris – What For?

The following are typical uses of DBUris and XDBUris:

- You can reference XSLT style sheets from within database-generated Web pages. PL/SQL package `DBMS_METADATA` uses DBUris to reference XSL style sheets. An XDBUri can be used to reference XSLT style sheets stored in Oracle XML DB Repository.
- You can reference HTML text, images and other data stored in the database. URLs can be used to point to data stored in database tables or in repository folders.

- You can improve performance by bypassing the Web server. Replace a global URL in your XML document with a reference to the database, and use a servlet, a DBUri, or an XDBUri to retrieve the targeted content. Using a DBUri or an XDBUri generally provides better performance than using a servlet, because you interact directly with the database rather than through a Web server.
- With a DBUri, you can access an XML document in the database without using SQL.
- Whenever a repository resource is stored in a database table to which you have access, you can use either an XDBUri or a DBUri to access its content.

> **See Also:** *Oracle Database PL/SQL Packages and Types Reference*, "DBMS_METADATA package"

URIType Methods

Abstract object type `URIType` includes PL/SQL methods that can be used with each of its subtypes. Each of these methods can be overridden by any of the subtypes. Table 20–1 lists the `URIType` PL/SQL methods. In addition, each of the subtypes has a constructor with the same name as the subtype.

Table 20–1 URIType PL/SQL Methods

URIType Method	Description
`getURL()`	Returns the URL of the `URIType` instance. Use this method instead of referencing a URL directly. `URIType` subtypes override this method to provide the correct URL. For example, `HTTPURIType` stores a URL without prefix `http://`. Method `getURL()` then prepends the prefix and returns the entire URL.
`getExternalURL()`	Similar to `getURL()`, but `getExternalURL()` escapes characters in the URL, to conform with the URL specification. For example, spaces are converted to the escaped value `%20`.
`getContentType()`	Returns the MIME content type for the URI. *HTTPUri:* To return the content type, the URL is followed and the MIME header examined. *DBUri:* The returned content type is either `text/plain` (for a scalar value) or `text/xml` (otherwise). *XDBUri:* The value of the `ContentType` metadata property of the repository resource is returned.
`getCLOB()`	Returns the target of the URI as a `CLOB` value. The database character set is used for encoding the data. *DBUri: XML* data is returned (unless node-test `text()` is used, in which case the targeted data is returned as is). When a `BLOB` column is targeted, the binary data in the column is *translated as hexadecimal character data*.
`getBLOB()`	Returns the target of the URI as a `BLOB` value. No character conversion is performed, and the character encoding is that of the URI target. This method can also be used to fetch binary data. *DBUri:* When applied to a DBUri that targets a `BLOB` column, `getBLOB()` returns the binary data *translated as hexadecimal character data*. When applied to a DBUri that targets *non*-binary data, the data is returned in the database character set.

Table 20–1 (Cont.) URIType PL/SQL Methods

URIType Method	Description
`getXML()`	Returns the target of the URI as an `XMLType` instance. Using this, an application that performs operations other than `getCLOB()` and `getBLOB()` can use `XMLType` methods to do those operations. This throws an exception if the URI does not target a well-formed XML document.
`createURI()`	Constructs an instance of one of the `URIType` subtypes.

HTTPURIType PL/SQL Method GETCONTENTTYPE()

`HTTPURIType` PL/SQL method `getContentType()` returns the MIME information for its targeted document. You can use this information to decide whether to retrieve the document as a `BLOB` value or a `CLOB` value. For example, you might treat a Web page with a MIME type of `x/jpeg` as a `BLOB` value, and one with a MIME type of `text/plain` or `text/html` as a `CLOB` value.

Example 20–1 tests the HTTP content type to determine whether to retrieve data as a `CLOB` or `BLOB` value. The content-type data is the HTTP header, for `HTTPURIType`, or the metadata of the database column, for `DBURIType`.

Example 20–1 Using HTTPURIType PL/SQL Method GETCONTENTTYPE()

```
DECLARE
  httpuri HTTPURIType;
  y CLOB;
  x BLOB;
BEGIN
  httpuri := HTTPURIType('http://www.oracle.com/index.html');
  DBMS_OUTPUT.put_line(httpuri.getContentType());
  IF httpuri.getContentType() = 'text/html'
  THEN
     y := httpuri.getCLOB();
  END IF;
  IF httpuri.getContentType() = 'application-x/bin'
  THEN
     x := httpuri.getBLOB();
  END IF;
END;
/
text/html
```

DBURIType PL/SQL Method GETCONTENTTYPE()

PL/SQL method `getContentType()` returns the MIME information for a URL. If a DBUri targets a scalar value, then the MIME content type returned is `text/plain`. Otherwise, the type returned is `text/xml`.

```
CREATE TABLE dbtab (a VARCHAR2(20), b BLOB);
```

DBUris corresponding to the following XPath expressions have content type `text/xml`, because each targets a complete column of XML data.

- `/HR/DBTAB/ROW/A`
- `/HR/DBTAB/ROW/B`

DBUris corresponding to the following XPath expressions have content type `text/plain`, because each targets a scalar value.

- /HR/DBTAB/ROW/A/text()
- /HR/DBTAB/ROW/B/text()

DBURIType PL/SQL Method GETCLOB()

When PL/SQL method getCLOB() is applied to a DBUri, the targeted data is returned as *XML* data, using the targeted column or table name as an XML element name. If the target XPath uses node-test text(), then the data is returned as text without an enclosing XML tag. In both cases, the returned data is in the database character set.

For example: If applied to a DBUri with XPath /HR/DBTAB/ROW/A/text(), where A is a non-binary column, the data in column A is returned as is. Without XPath node-test text(), the result is the data wrapped in XML:

```
<HR><DBTAB><ROW><A>...data_in_column_A...</A></ROW></DBTAB></HR>
```

When applied to a DBUri that targets a *binary* (BLOB) column, the binary data in the column is *translated as hexadecimal character data*.

For example: If applied to a DBUri with XPath /HR/DBTAB/ROW/B/text(), where B is a BLOB column, the targeted binary data is translated to hexadecimal character data and returned. Without XPath node-test text(), the result is the translated data wrapped in XML:

```
<HR><DBTAB><ROW><B>...data_translated_to_hex...</B></ROW></DBTAB></HR>
```

DBURIType PL/SQL Method GETBLOB()

When applied to a DBUri that targets a BLOB column, getBLOB() returns the binary data *translated as hexadecimal character data*. When applied to a DBUri that targets *non*-binary data, getBLOB() returns the data (as a BLOB value) in the database character set.

For example, consider table dbtab:

```
CREATE TABLE dbtab (a VARCHAR2(20), b BLOB);
```

When getBLOB() is applied to a DBUri corresponding to XPath expression /HR/DBTAB/ROW/B, it returns a BLOB value containing an XML document with root element B whose content is the hexadecimal-character translation of the binary data of column B.

When getBLOB() is applied to a DBUri corresponding to XPath expression /HR/DBTAB/ROW/B/text(), it returns a BLOB value containing only the hexadecimal-character translation of the binary data of column B.

When getBLOB() is applied to a DBUri corresponding to XPath expression /HR/DBTAB/ROW/**A**/text(), which targets *non*-binary data, it returns a BLOB value containing the data of column A, in the database character set.

Accessing Data using URIType Instances

To use instances of URIType subtypes for indirection, you generally store such instances in the database and then use them in queries with a PL/SQL method such as getCLOB() to retrieve the targeted data. This section illustrates how to do this.

You can create database columns using URIType or any of its subtypes, or you can store just the text of each URI as a string and then create the needed URIType

instances on demand, when the URIs are accessed. You can store objects of different URIType subtypes in the same URIType database column.

You can also define your own object types that inherit from the URIType subtypes. Deriving new types lets you use custom techniques to retrieve, transform, or filter data.

See Also:

- "Creating New Subtypes of URIType using Package URIFACTORY" on page 20-20 for information about defining new URIType subtypes
- "XSL Transformation and Oracle XML DB" on page 3-64 for information about transforming XML data

Example 20–2 stores an HTTPUri and a DBUri (instances of URIType subtypes HTTPURIType and DBURIType) in the same database column of type URIType. A query retrieves the data addressed by each of the URIs. The first URI is a Web-page URL. The second URI references data in table employees of standard database schema HR. (For brevity, only the beginning of the Web page is shown.)

Example 20–2 Creating and Querying a URI Column

```
CREATE TABLE uri_tab (url URIType);
Table created.

INSERT INTO uri_tab VALUES (HTTPURIType.createURI('http://www.oracle.com'));
1 row created.

INSERT INTO uri_tab VALUES (DBURIType.createURI(
                              '/HR/EMPLOYEES/ROW[FIRST_NAME="Jack"]'));
1 row created.

SELECT e.url.getCLOB() FROM uri_tab e;

E.URL.GETCLOB()
--------------------------------------------------------------------
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
. . .

<?xml version="1.0"?>
 <ROW>
  <EMPLOYEE_ID>177</EMPLOYEE_ID>
  <FIRST_NAME>Jack</FIRST_NAME>
  <LAST_NAME>Livingston</LAST_NAME>
  <EMAIL>JLIVINGS</EMAIL>
  <PHONE_NUMBER>011.44.1644.429264</PHONE_NUMBER>
  <HIRE_DATE>23-APR-06</HIRE_DATE>
  <JOB_ID>SA_REP</JOB_ID>
  <SALARY>8400</SALARY>
  <COMMISSION_PCT>.2</COMMISSION_PCT>
  <MANAGER_ID>149</MANAGER_ID>
  <DEPARTMENT_ID>80</DEPARTMENT_ID>
 </ROW>

2 rows selected.
```

To use URIType PL/SQL method createURI(), you must know the particular URIType subtype to use. PL/SQL method getURI() of package URIFACTORY lets you instead use the flexibility of late binding, determining the particular type information at run time.

URIFACTORY.getURI() takes as argument a URI string. It returns a URIType instance of the appropriate subtype (HTTPURIType, DBURIType, or XDBURIType), based on the form of the URI string:

- If the URI starts with http://, then getURI() creates and returns an HTTPUri.
- If the URI starts with either /oradb/ or /dburi/, then getURI() creates and returns a DBUri.
- Otherwise, getURI() creates and returns an XDBUri.

Example 20–3 is similar to Example 20–2, but it uses two different ways to obtain documents targeted by URIs:

- PL/SQL method SYS.URIFACTORY.getURI() with *absolute* URIs:
  - an HTTPUri that targets HTTP address http://www.oracle.com
  - a DBUri that targets database address /oradb/HR/EMPLOYEES/ROW[EMPLOYEE_ID=200]
- Constructor SYS.HTTPURIType() with a *relative* URL (no http://). The same HTTPUri is used as for the absolute URI: the Oracle home page.

In Example 20–3, the URI strings passed to getURI() are hard-coded, but they could just as easily be string values that are obtained by an application at run time.

Example 20–3 Using Different Kinds of URI, Created in Different Ways

```
CREATE TABLE uri_tab (docUrl SYS.URIType, docName VARCHAR2(200));
Table created.

-- Insert an HTTPUri with absolute URL into SYS.URIType using URIFACTORY.
-- The target is Oracle home page.
INSERT INTO uri_tab VALUES
  (SYS.URIFACTORY.getURI('http://www.oracle.com'), 'AbsURL');
1 row created.

-- Insert an HTTPUri with relative URL using constructor SYS.HTTPURIType.
-- Note the absence of prefix http://. The target is the same.
INSERT INTO uri_tab VALUES (SYS.HTTPURIType('www.oracle.com'), 'RelURL');
1 row created.

-- Insert a DBUri that targets employee data from table HR.employees.
INSERT INTO uri_tab VALUES
  (SYS.URIFACTORY.getURI('/oradb/HR/EMPLOYEES/ROW[EMPLOYEE_ID=200]'), 'Emp200');
1 row created.

-- Extract all of the documents.
SELECT e.docUrl.getCLOB(), docName FROM uri_tab e;

E.DOCURL.GETCLOB()
------------------
DOCNAME
------------------------------------
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
```

```
. . .
AbsURL

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
. . .
RelURL

<?xml version="1.0"?>
 <ROW>
  <EMPLOYEE_ID>200</EMPLOYEE_ID>
  <FIRST_NAME>Jennifer</FIRST_NAME>
  <LAST_NAME>Whalen</LAST_NAME>
  <EMAIL>JWHALEN</EMAIL>
  <PHONE_NUMBER>515.123.4444</PHONE_NUMBER>
  <HIRE_DATE>17-SEP-03</HIRE_DATE>
  <JOB_ID>AD_ASST</JOB_ID>
  <SALARY>4400</SALARY>
  <MANAGER_ID>101</MANAGER_ID>
  <DEPARTMENT_ID>10</DEPARTMENT_ID>
 </ROW>
Emp200

3 rows selected.

-- In PL/SQL
CREATE OR REPLACE FUNCTION returnclob
  RETURN CLOB
  IS a SYS.URIType;
BEGIN
  SELECT docUrl INTO a FROM uri_Tab WHERE docName LIKE 'Emp200%';
  RETURN a.getCLOB;
END;
/
Function created.

SELECT returnclob() FROM DUAL;

RETURNCLOB()
---------------------------------------------------------------
<?xml version="1.0"?>
 <ROW>
  <EMPLOYEE_ID>200</EMPLOYEE_ID>
  <FIRST_NAME>Jennifer</FIRST_NAME>
  <LAST_NAME>Whalen</LAST_NAME>
  <EMAIL>JWHALEN</EMAIL>
  <PHONE_NUMBER>515.123.4444</PHONE_NUMBER>
  <HIRE_DATE>17-SEP-03</HIRE_DATE>
  <JOB_ID>AD_ASST</JOB_ID>
  <SALARY>4400</SALARY>
  <MANAGER_ID>101</MANAGER_ID>
  <DEPARTMENT_ID>10</DEPARTMENT_ID>
 </ROW>

1 row selected.
```

XDBUris: Pointers to Repository Resources

XDBURIType is a subtype of URIType that provides a way to expose resources in Oracle XML DB Repository using URIs. Instances of type XDBURIType are called **XDBUris**.

XDBUri URI Syntax

The URL portion of an XDBUri URI is the hierarchical address of the targeted repository resource – it is a *repository* path (*not* an XPath expression).

The optional fragment portion of the URI uses the XPath syntax, and is separated from the URL part by a number-sign (#). It is appropriate only if the targeted resource is an *XML* document, in which case the fragment portion targets one or more parts of the XML document. If the targeted resource is not an XML document, then omit the fragment and number-sign.

The following are examples of XDBUri URIs:

- /public/hr/image27.jpg
- /public/hr/doc1.xml#/PurchaseOrder/LineItem

Based on the form of these URIs:

- /public/hr is a folder resource in Oracle XML DB Repository.
- image27.jpg and doc1.xml are resources in folder /public/hr.
- Resource doc1.xml is a file resource, and it contains an XML document.
- The XPath expression /PurchaseOrder/LineItem refers to the LineItem child element in element PurchaseOrder of XML document doc1.xml.

You can create an XDBUri using PL/SQL method getURI() of package URIFACTORY.

XDBURIType is the *default* URIType used when generating instances using URIFACTORY PL/SQL method getURI(), unless the URI has one of the recognized prefixes http://, /dburi, or /oradb.

For example, if resource doc1.xml is present in repository folder /public/hr, then the following query returns an XDBUri that targets that resource.

```
SELECT SYS.URIFACTORY.getURI('/public/hr/doc1.xml') FROM DUAL;
```

It is the lack of a special prefix that determines that the type is XDBURIType, not any particular resource file extension or the presence of # followed by an XPath expression. Even if the resource were named foo.bar instead of doc1.xml, the returned URIType instance would still be an XDBUri.

XDBUri Examples

Example 20–4 creates an XDBUri, inserts values into a purchase-order table, and then selects all of the purchase orders. Because there is no special prefix used in the URI passed to URIFACTORY.getURI(), the created URIType instance is an XDBUri.

Example 20–4 Access a Repository Resource by URI using an XDBUri

```
DECLARE
res BOOLEAN;
postring VARCHAR2(100):= '<?xml version="1.0"?>
<ROW>
<PO>999</PO>
```

```
</ROW>';
BEGIN
res:=DBMS_XDB.createFolder('/public/orders/');
res:=DBMS_XDB.createResource('/public/orders/po1.xml', postring);
END;
/
PL/SQL procedure successfully completed.

CREATE TABLE uri_tab (poUrl SYS.URIType, poName VARCHAR2(1000));
Table created.

-- Create an abstract type column so any type of URI can be used
-- Insert an absolute URL into poUrl.
-- The factory will create an XDBURIType because there is no prefix.
-- Here, po1.xml is an XML file that is stored in /public/orders/
-- of the XML repository.
INSERT INTO uri_tab VALUES
  (URIFACTORY.getURI('/public/orders/po1.xml'), 'SomePurchaseOrder');
1 row created.

-- Get all the purchase orders
SELECT e.poUrl.getCLOB(), poName FROM uri_tab e;

E.POURL.GETCLOB()
-----------------
PONAME
------
<?xml version="1.0"?>
<ROW>
<PO>999</PO>
</ROW>
SomePurchaseOrder

1 row selected.

-- Using PL/SQL, you can access table uri_tab as follows:
CREATE OR REPLACE FUNCTION returnclob
  RETURN CLOB
  IS a URIType;
BEGIN
  -- Get absolute URL for purchase order named like 'Some%'
   SELECT poUrl INTO a FROM uri_tab WHERE poName LIKE 'Some%';
   RETURN a.getCLOB();
END;
/
Function created.

SELECT returnclob() FROM DUAL;

RETURNCLOB()
---------------------
<?xml version="1.0"?>
<ROW>
<PO>999</PO>
</ROW>

1 row selected.
```

Because PL/SQL method `getXML()` returns an `XMLType` instance, you can use it with SQL/XML functions such as `XMLQuery`. The query in Example 20–5 illustrates this. The query retrieves all purchase orders numbered 999.

Example 20–5 Using PL/SQL Method GETXML() with XMLCAST and XMLQUERY

```
SELECT e.poUrl.getCLOB() FROM uri_tab e
  WHERE XMLCast(XMLQuery('$po/ROW/PO'
                         PASSING e.poUrl.getXML() AS "po"
                         RETURNING CONTENT)
                 AS VARCHAR2(24))
        = '999';

E.POURL.GETCLOB()
---------------------
<?xml version="1.0"?>
<ROW>
<PO>999</PO>
</ROW>

1 row selected.
```

DBUris: Pointers to Database Data

A DBUri is a URI that targets *database data*. As for all instances of `URIType` subtypes, a DBUri provides an indirection mechanism for accessing data. In addition, `DBURIType` lets you do the following:

- Address database data using XPath notation. This, in effect, lets you visualize and access the database as if it were XML data.

  For example, a DBUri can use an expression such as `/HR/EMPLOYEES/ROW[FIRST_NAME="Jack"]` to target the row of table `HR.employees` where column `first_name` has value `Jack`.

- *Construct an XML document* that contains database data targeted by a DBUri and whose structure reflects the database structure.

  For example: A DBUri with XPath `/HR/DBTAB/ROW/A` can be used to construct an XML document that wraps the data of column `A` in XML elements that reflect the database structure and are named accordingly:

  `<HR><DBTAB><ROW><A>...data_in_column_A...</A></ROW></DBTAB></HR>`

A DBUri does not reference a global location as does an HTTPUri. You can, however, also access objects addressed by a DBUri in a global manner, by appending the DBUri to an HTTPUri that identifies a servlet that handles DBUris – see "DBUriServlet" on page 20-26.

Viewing the Database as XML Data

You can access only those database schemas to which you have been granted access privileges. This portion of the database is, in effect, your own view of the database.

Using `DBURIType`, you can have corresponding *XML* views of the database, which are portions of the database to which you have access, presented *in the form of XML data*. This means all kinds database data, not just data that is stored as XML. When visualized this way, the database data is effectively wrapped in XML elements, resulting in one or more XML documents.

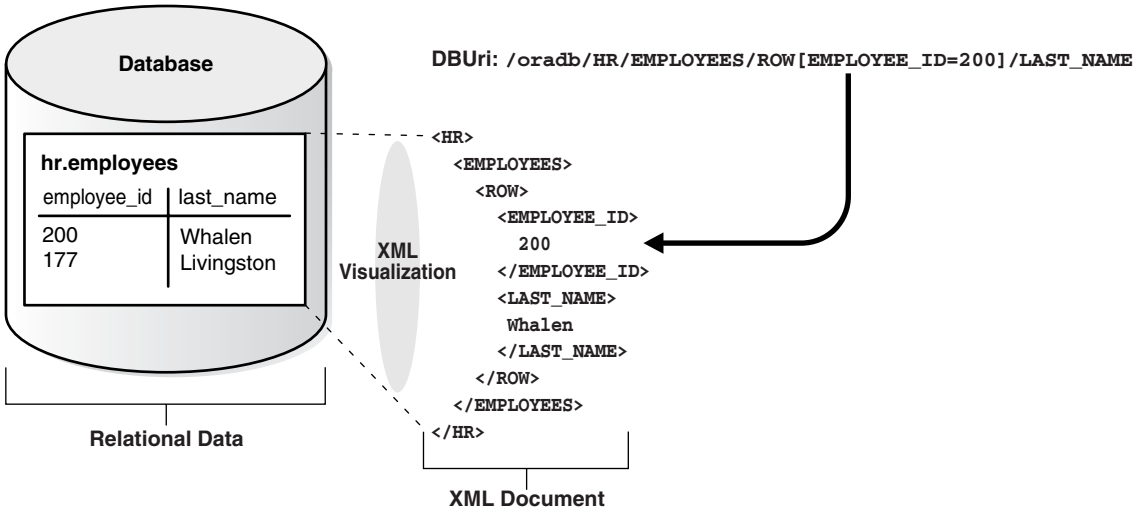
Such "XML views" are not database views, in the technical sense of the term. "View" here means only an abstract perspective that can be useful for understanding `DBURIType`. You can think of `DBURIType` as providing a way to visualize and access the database *as if it were* XML data.

However, `DBURIType` does not just provide an exercise in visualization and an additional means to access database data. Each "XML view" can be realized as an XML document – that is, you can use `DBURIType` to generate XML documents using database data.

All of this is another way of saying that `DBURIType` lets you use XPath notation to 1) address and access any database data to which you have access and 2) construct XML representations of that data.

Figure 20–1 illustrates the relation between a relational table, `HR.employees`, a corresponding XML view of a portion of that table, and the corresponding DBUri URI (a simple XPath expression). In this case, the portion of the data exposed as XML is the row where `employee_id` is `200`. The URI can be used to access the data and construct an XML document that reflects the "XML view".

Figure 20–1 A DBUri Corresponds to an XML Visualization of Relational Data



The XML elements in the "XML view" and the steps in the URI XPath expression both reflect the database table and column names. Note the use of **`ROW`** to indicate a row in the database table – both in the "XML view" and in the URI XPath expression.

Note also that the XPath expression contains a root-element step, **`oradb`**. This is used to indicate that the URI corresponds to a DBUri, not an HTTPUri or an XDBUri. Whenever this correspondence is understood from context, this XPath step can be skipped. For example, if it is known that the path in question is a path to database data, the following URIs are equivalent:

- **`/oradb`**`/HR/EMPLOYEES/ROW[EMPLOYEE_ID=200]/LAST_NAME`
- `/HR/EMPLOYEES/ROW[EMPLOYEE_ID=200]/LAST_NAME`

Whenever the URI context is not clear, however, you must use the prefix `/oradb` to distinguish a URI as corresponding to a DBUri. In particular, you must supply the prefix to `URIFACTORY` PL/SQL methods and to DBUriServlet.

See Also:

- "Creating New Subtypes of URIType using Package URIFACTORY" on page 20-20
- "DBUriServlet" on page 20-26
- Chapter 18, "Generating XML Data from the Database" for other ways to generate XML from database data

DBUri URI Syntax

An XPath expression is a path into XML data that addresses one or more XML nodes. A DBUri exploits the notion of a virtual XML user visualization of the database to use a *simple form* of XPath expression as a URI to address database data. This is so, regardless of the type of data, in particular, whether or not the data is XML.

Thus, for `DBURIType`, Oracle Database supports only a subset of the full XPath or XPointer syntax. There are no syntax restrictions for XDBUri XPath expressions. There is also an exception in the DBUri case: data in `XMLType` tables. For an `XMLType` table, the simple XPath form is used to address the table itself within the database. Then, to address particular XML data in the table, the remainder of the XPath expression can use the full XPath syntax. This exception applies only to `XMLType` *tables*, not to `XMLType` columns.

In any case, unlike an XDBUri, a DBUri URI does not use a number-sign (`#`) to separate the URL portion of a URI from a fragment (XPath) portion. `DBURIType` does not use URI fragments. Instead, the entire URI is treated as a (simple) XPath expression.

You can create DBUris to any database data to which you have access. XPath expressions such as the following are allowed:

- `/database_schema/table`
- `/database_schema/table/ROW[predicate_expression]/column`
- `/database_schema/table/ROW[predicate_expression]/object_column/attribute`
- `/database_schema/`***`XMLType_table`***`/ROW/XPath_expression`

In the last case, `XMLType_table` is an `XMLType` table, and `XPath_expression` is *any* XPath expression. For tables that are *not* `XMLType`, a DBUri XPath expression must end at a column (it cannot address specific data inside a column). This restriction includes `XMLType` columns, LOB columns, and `VARCHAR2` columns that contain XML data.

A DBUri XPath expression can do any of the following:

- Target an entire table.

  For example, `/HR/EMPLOYEES` targets table `employees` of database schema `HR`.
- Include XPath predicates at any step in the path, except the database schema and table steps.

  For example, `/HR/EMPLOYEES/ROW[EMPLOYEE_ID=200]/EMAIL` targets column `email` of table `HR.employees`, where `employee_id` is `200`.
- Use the `text()` XPath node test on data with scalar content. This is the *only* node test that can be used, and it cannot be used with the table or row step.

The following can be used in DBUri (XPath) *predicate* expressions:

- Boolean operators `and`, `or`, and `not`
- Relational operators `<`, `>`, `<=`, `!=`, `>=`, `=`, `mod`, `div`, `*` (multiply)

A DBUri XPath expression *must* do all of the following:

- Use only the *child* XPath axis – other axes, such as parent, are not allowed.
- Either specify a database schema or specify `PUBLIC` to resolve the table name without a specific schema.
- Specify a database view or table name.
- Include a `ROW` step, if a database column is targeted.
- Identify a *single* data value, which can be an object-type instance or a collection.
- Result in well-formed XML when it is used to generate XML data using database data.

  An example of a DBUri that does *not* result in well-formed XML is `/HR/EMPLOYEES/ROW/LAST_NAME`. It returns more than one `<LAST_NAME>` element fragment, with no single root element.
- Use *none* of the following:
  - `*` (wildcard)
  - `.` (self)
  - `..` (parent)
  - `//` (descendant or self)
  - XPath functions, such as `count`

A DBUri XPath expression can optionally be prefixed by **`/oradb`** or **`/dburi`** (the two are equivalent) to distinguish it. This prefix is case-insensitive. However, the rest of the DBUri XPath expression is *case-sensitive*, as are XPath expressions generally. Thus, for example, to specify table `HR.employees` as a DBUri XPath expression, you must use `HR/EMPLOYEES`, not `hr/employees` (or a mixed-case combination), because table and column names are uppercase, by default.

See Also: `http://www.w3.org/TR/xpath` on XPath notation

DBUris are Scoped to a Database and Session

The content of the XML views you have of the database, and hence of the XML documents that you can construct, reflects the permissions you have for accessing particular database data at a given time. That is, a DBUri is scoped to a given database session, so the same DBUri can give different results in the same query, depending on the session context (which user is connected and what privileges the user has).

To complicate things a bit, there is also an XML element `PUBLIC`, under which database data is accessible without any database-schema qualification. This is a convenience feature, but it can also lead to some confusion if you forget that the XML views of the database for a given user depend on the specific access the user has to the database at a given time.

XML element `PUBLIC` corresponds to the use of a *public synonym*. For example, when queried by user `quine`, the following query tries to match table `foo` under database schema `quine`, but if no such table exists, it tries to match a public synonym named `foo`.

```
SELECT * FROM foo;
```

In the same way, XML element `PUBLIC` contains all of the database data visible to a given user and all of the data visible to that user through public synonyms. So, the same DBUri URI `/PUBLIC/FOO` can resolve to `quine.foo` when user `quine` is connected, and resolve to `curry.foo` when user `curry` is connected.

DBUri Examples

A DBUri can identify a table, a row, a column in a row, or an attribute of an object column. The following sections describe how to target different object types.

Targeting a Table

You can target a complete database table, using this syntax:

```
/database_schema/table
```

Example 20–6 uses a DBUri that targets a complete table. An XML document is returned that corresponds to the table contents. The top-level XML element is named for the table. The values of each row are enclosed in a `ROW` element.

Example 20–6 Targeting a Complete Table using a DBUri

```
CREATE TABLE uri_tab (url URIType);
Table created.

INSERT INTO uri_tab VALUES
       (DBURIType.createURI('/HR/EMPLOYEES'));
1 row created.

SELECT e.url.getCLOB() FROM uri_tab e;

E.URL.GETCLOB()
---------------
<?xml version="1.0"?>
<EMPLOYEES>
 <ROW>
  <EMPLOYEE_ID>100</EMPLOYEE_ID>
  <FIRST_NAME>Steven</FIRST_NAME>
  <LAST_NAME>King</LAST_NAME>
  <EMAIL>SKING</EMAIL>
  <PHONE_NUMBER>515.123.4567</PHONE_NUMBER>
  <HIRE_DATE>17-JUN-03</HIRE_DATE>
  <JOB_ID>AD_PRES</JOB_ID>
  <SALARY>24000</SALARY>
  <DEPARTMENT_ID>90</DEPARTMENT_ID>
 </ROW>
 <ROW>
  <EMPLOYEE_ID>101</EMPLOYEE_ID>
  <FIRST_NAME>Neena</FIRST_NAME>
  <LAST_NAME>Kochhar</LAST_NAME>
  <EMAIL>NKOCHHAR</EMAIL>
  <PHONE_NUMBER>515.123.4568</PHONE_NUMBER>
  <HIRE_DATE>21-SEP-05</HIRE_DATE>
  <JOB_ID>AD_VP</JOB_ID>
  <SALARY>17000</SALARY>
  <MANAGER_ID>100</MANAGER_ID>
  <DEPARTMENT_ID>90</DEPARTMENT_ID>
 </ROW>
 . . .
```

```
1 row selected.
```

Targeting a Row in a Table

You can target one or more specific rows of a table, using this syntax:

*/database_schema/table/*ROW[*predicate_expression*]

Example 20–7 uses a DBUri that targets a single table row. The XPath predicate expression identifies the single table row that corresponds to employee number 200. The result is an XML document with ROW as the top-level element.

Example 20–7 Targeting a Particular Row in a Table using a DBUri

```
CREATE TABLE uri_tab (url URIType);
Table created.

INSERT INTO uri_tab VALUES
       (DBURIType.createURI('/HR/EMPLOYEES/ROW[EMPLOYEE_ID=200]'));
1 row created.

SELECT e.url.getCLOB() FROM uri_tab e;

E.URL.GETCLOB()
-------------------------------------------------------
<?xml version="1.0"?>
 <ROW>
  <EMPLOYEE_ID>200</EMPLOYEE_ID>
  <FIRST_NAME>Jennifer</FIRST_NAME>
  <LAST_NAME>Whalen</LAST_NAME>
  <EMAIL>JWHALEN</EMAIL>
  <PHONE_NUMBER>515.123.4444</PHONE_NUMBER>
  <HIRE_DATE>17-SEP-03</HIRE_DATE>
  <JOB_ID>AD_ASST</JOB_ID>
  <SALARY>4400</SALARY>
  <MANAGER_ID>101</MANAGER_ID>
  <DEPARTMENT_ID>10</DEPARTMENT_ID>
 </ROW>

1 row selected.
```

Targeting a Column

You can target a specific column, using this syntax:

*/database_schema/table/*ROW[*predicate_expression*]*/column*

You can target a specific attribute of an object column, using this syntax:

*/database_schema/table/*ROW[*predicate_expression*]*/object_column/attribute*

You can target a specific object column whose attributes have specific values, using this syntax:

*/database_schema/table/*ROW[*predicate_expression_with_attributes*]*/object_column*

Example 20–8 uses a DBUri that targets column `last_name` for the same employee as in Example 20–7. The top-level XML element is named for the targeted column.

Example 20–8 Targeting a Specific Column using a DBUri

```
CREATE TABLE uri_tab (url URIType);
Table created.

INSERT INTO uri_tab VALUES
       (DBURIType.createURI('/HR/EMPLOYEES/ROW[EMPLOYEE_ID=200]/LAST_NAME'));
1 row created.

SELECT e.url.getCLOB() FROM uri_tab e;

E.URL.GETCLOB()
------------------------------
<?xml version="1.0"?>
 <LAST_NAME>Whalen</LAST_NAME>

1 row selected.
```

Example 20–9 uses a DBUri that targets a `CUST_ADDRESS` object column containing city and postal code attributes with certain values. The top-level XML element is named for the column, and it contains child elements for each of the object attributes.

Example 20–9 Targeting an Object Column with Specific Attribute Values using a DBUri

```
CREATE TABLE uri_tab (url URIType);
Table created.

INSERT INTO uri_tab VALUES
  (DBURIType.createURI(
     '/OE/CUSTOMERS/ROW[CUST_ADDRESS/CITY="Poughkeepsie" and
                        CUST_ADDRESS/POSTAL_CODE=12601]/CUST_ADDRESS'));
1 row created.

SELECT e.url.getCLOB() FROM uri_tab e;

E.URL.GETCLOB()
---------------
<?xml version="1.0"?>
 <CUST_ADDRESS>
  <STREET_ADDRESS>33 Fulton St</STREET_ADDRESS>
  <POSTAL_CODE>12601</POSTAL_CODE>
  <CITY>Poughkeepsie</CITY>
  <STATE_PROVINCE>NY</STATE_PROVINCE>
  <COUNTRY_ID>US</COUNTRY_ID>
 </CUST_ADDRESS>

1 row selected.
```

The DBUri identifies the object that has a `CITY` attribute with `Poughkeepsie` as value and a `POSTAL_CODE` attribute with `12601` as value.

Retrieving the Text Value of a Column

In many cases, it can be useful to retrieve only the text values of a column and not the enclosing tags. For example, if XSLT style sheets are stored in a `CLOB` column, you can retrieve the document text without having any enclosing column-name tags. You can use the `text()` XPath node test for this. It specifies that you want only the text value of the node. Use the following syntax:

/oradb/*database_schema*/*table*/ROW[*predicate_expression*]/*column*/**text()**

Example 20–10 retrieves the text value of the employee `last_name` column for employee number 200, without the XML tags.

Example 20–10 Retrieve Only the Text Value of a Node using a DBUri

```
CREATE TABLE uri_tab (url URIType);
Table created.

INSERT INTO uri_tab VALUES
       (DBURIType.createURI(
          '/HR/EMPLOYEES/ROW[EMPLOYEE_ID=200]/LAST_NAME/text()'));

1 row created.

SELECT e.url.getCLOB() FROM uri_tab e;

E.URL.GETCLOB()
---------------
Whalen

1 row selected.
```

Targeting a Collection

You can target a database collection, such as an ordered collection table. You must, however, target the entire collection – you cannot target individual members of a collection. When a collection is targeted, the XML document produced by the DBUri contains each collection member as an XML element, with all such elements enclosed in a element named for the *type* of the collection.

Example 20–11 uses a DBUri that targets a collection of numbers. The top-level XML element is named for the collection, and its children are named for the collection *type* (`NUMBER`).

Example 20–11 Targeting a Collection using a DBUri

```
CREATE TYPE num_collection AS VARRAY(10) OF NUMBER;
/
Type created.

CREATE TABLE orders (item VARCHAR2(10), quantities num_collection);
Table created.

INSERT INTO orders VALUES ('boxes', num_collection(3, 7, 4, 9));
1 row created.

SELECT * FROM orders;

ITEM
----
QUANTITIES
----------
boxes
NUM_COLLECTION(3, 7, 4, 9)

1 row selected.

SELECT DBURIType('/HR/ORDERS/ROW[ITEM="boxes"]/QUANTITIES').getCLOB() FROM DUAL;

DBURITYPE('/HR/ORDERS/ROW[ITEM="BOXES"]/QUANTITIES').GETCLOB()
```

```
------------------------------------------------------------
<?xml version="1.0"?>
 <QUANTITIES>
  <NUMBER>3</NUMBER>
  <NUMBER>7</NUMBER>
  <NUMBER>4</NUMBER>
  <NUMBER>9</NUMBER>
 </QUANTITIES>

1 row selected.
```

Creating New Subtypes of URIType using Package URIFACTORY

You can use PL/SQL package `URIFACTORY` to do more than create `URIType` instances. Additional PL/SQL methods are listed in Table 20–2.

Table 20–2 URIFACTORY PL/SQL Methods

PL/SQL Method	Description
`getURI()`	Returns the URL of the `URIType` instance.
`escapeURI()`	Escapes the URI string by replacing characters that are not permitted in URIs by their equivalent escape sequence.
`unescapeURI()`	Removes escaping from a given URI.
`registerURLHandler()`	Registers a particular type name for handling a particular URL. This is called by `getURI()` to generate an instance of the type. A Boolean argument can be used to indicate that the prefix must be stripped off before calling the appropriate type constructor.
`unregisterURLHandler()`	Unregisters a URL handler.

Of particular note is that you can use package `URIFACTORY` to define new subtypes of type `URIType`. You can then use those subtypes to provide specialized processing of URIs. In particular, you can define `URIType` subtypes that correspond to particular protocols – `URIFACTORY` then recognizes and processes instances of those subtypes accordingly.

Defining new types and creating database columns specific to the new types has these advantages:

- It provides an implicit *constraint* on the columns to contain only instances of those types. This can be useful for implementing specialized indexes on a column for specific protocols. For a DBUri, for instance, you can implement specialized indexes that fetch data directly from disk blocks, rather than executing SQL queries.
- You can have different constraints on different columns, based on the type. For a HTTPUri, for instance, you can define proxy and firewall constraints on a column, so that any access through the HTTP uses the proxy server.

Registering New URIType Subtypes with Package URIFACTORY

To provide specialized processing of URIs, you define and register a new URIType subtype, as follows:

1. Create the new type using SQL statement `CREATE TYPE`. The type must implement PL/SQL method `createURI()`.

2. Optionally override the default methods, to perform specialized processing when retrieving data or to transform the XML data before displaying it.
3. Choose a new URI prefix, to identify URIs that use this specialized processing.
4. Register the new prefix using PL/SQL method `registerURLHandler()`, so that package `URIFACTORY` can create an instance of your new subtype when it receives a URI starting with the new prefix you defined.

After the new subtype is defined, a URI with the new prefix is recognized by `URIFACTORY` methods, and you can create and use instances of the new type.

For example, suppose that you define a new protocol prefix, `ecom://`, and define a subtype of `URIType` to handle it. Perhaps the new subtype implements some special logic for PL/SQL method `getCLOB()`, or perhaps it makes some changes to XML tags or data in method `getXML()`. After you register prefix `ecom://` with `URIFACTORY`, a call to `getURI()` generates an instance of the new `URIType` subtype for a URI with that prefix.

Example 20–12 creates a new type, `ECOMURIType`, to handle a new protocol, `ecom://`. The example stores three different kinds of URIs in a single table: an HTTPUri, a DBUri, and an instance of the new type, `ECOMURIType`. To run this example, you would need to define each of the `ECOMURIType` member functions.

Example 20–12 URIFACTORY: Registering the ECOM Protocol

```
CREATE TABLE url_tab (urlcol varchar2(80));
Table created.

-- Insert an HTTP URL reference
INSERT INTO url_tab VALUES ('http://www.oracle.com/');
1 row created.

-- Insert a DBUri
INSERT INTO url_tab VALUES ('/oradb/HR/EMPLOYEES/ROW[FIRST_NAME="Jack"]');
1 row created.

-- Create a new type to handle a new protocol called ecom://
-- This is just an example template. For this to run, the implementations
-- of these functions must be specified.
CREATE OR REPLACE TYPE ECOMURIType UNDER SYS.URIType (
  OVERRIDING MEMBER FUNCTION getCLOB RETURN CLOB,
  OVERRIDING MEMBER FUNCTION getBLOB RETURN BLOB,
  OVERRIDING MEMBER FUNCTION getExternalURL RETURN VARCHAR2,
  OVERRIDING MEMBER FUNCTION getURI RETURN VARCHAR2,
  -- Must have this for registering with the URL handler
  STATIC FUNCTION createURI(url IN VARCHAR2) RETURN ECOMURIType);
/
-- Register a new handler for the ecom:// prefixes
BEGIN
  -- The handler type name is ECOMURIType; schema is HR
  -- Ignore the prefix case, so that URIFACTORY creates the same subtype
  -- for URIs beginning with ECOM://, ecom://, eCom://, and so on.
  -- Strip the prefix before calling PL/SQL method createURI(),
  -- so that the string 'ecom://' is not stored inside the
  -- ECOMURIType object. It is added back automatically when
  -- you call ECOMURIType.getURI().
  URIFACTORY.registerURLHandler (prefix => 'ecom://',
                                 schemaname => 'HR',
                                 typename => 'ECOMURITYPE',
                                 ignoreprefixcase => TRUE,
```

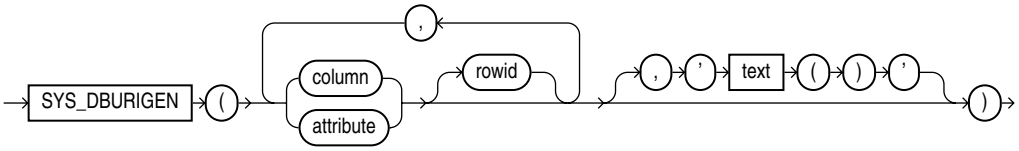
```
                                  stripprefix => TRUE);
END;
/
PL/SQL procedure successfully completed.

-- Insert this new type of URI into the table
INSERT INTO url_tab VALUES ('ECOM://company1/company2=22/comp');
1 row created.

-- Use the factory to generate an instance of the appropriate
-- subtype for each URI in the table.

-- You would need to define the member functions for this to work:
SELECT urifactory.getURI(urlcol) FROM url_tab;

-- This would generate:
HTTPURIType('www.oracle.com'); -- an HTTPUri
DBURIType('/oradb/HR/EMPLOYEES/ROW[FIRST_NAME="Jack"]', null); -- a DBUri
ECOMURIType('company1/company2=22/comp'); -- an ECOMURIType instance
```

SYS_DBURIGEN SQL Function

You can create a DBUri by providing an XPath expression to constructor `DBURIType` or to appropriate `URIFACTORY` PL/SQL methods. With Oracle SQL function `sys_DburiGen`, you can alternatively create a DBUri with an *XPath that is composed* from database columns and their values.

Oracle SQL function `sys_DburiGen` takes as its argument one or more database columns or attributes, and optionally a rowid, and generates a DBUri that targets a particular column or row object. Function `sys_DburiGen` takes an additional parameter that indicates whether the text value of the node is needed. See Figure 20–2.

Figure 20–2 SYS_DBURIGEN Syntax



All columns or attributes referenced must reside in the same table. They must each reference a unique value. If you specify multiple columns, then the initial columns identify the row, and the last column identifies the column within that row. If you do not specify a database schema, then the table name is interpreted as a public synonym.

> **See Also:** *Oracle Database SQL Language Reference*

Example 20–13 uses Oracle SQL function `sys_DburiGen` to generate a DBUri that targets column `email` of table `HR.employees` where `employee_id` is `206`:

Example 20–13 SYS_DBURIGEN: Generating a DBUri that Targets a Column

```
SELECT sys_DburiGen(employee_id, email)
  FROM employees
  WHERE employee_id = 206;

SYS_DBURIGEN(EMPLOYEE_ID,EMAIL)(URL, SPARE)
-------------------------------------------------------------------
```

```
DBURITYPE('/PUBLIC/EMPLOYEES/ROW[EMPLOYEE_ID = "206"]/EMAIL', NULL)

1 row selected.
```

Rules for Passing Columns or Object Attributes to SYS_DBURIGEN

A column or attribute passed to Oracle SQL function `sys_DburiGen` must obey the following rules:

- *Same table:* All columns referenced in function `sys_DburiGen` must come from the same table or view.
- *Unique mapping:* The column or object attribute must be uniquely mappable back to the table or view from which it came. The only virtual columns allowed are those produced with `value` or `ref`. The column can come from a subquery with a SQL `TABLE` collection expression, that is, `TABLE(...)`, or from an inline view (as long as the inline view does not rename the columns).

  See *Oracle Database SQL Language Reference* for information about the SQL `TABLE` collection expression.
- *Key columns:* Either the rowid or a set of key columns must be specified. The list of key columns is not required to be declared as a unique or primary key, as long as the columns uniquely identify a particular row in the result.
- **`PUBLIC`** *element:* If the table or view targeted by the rowid or key columns does not specify a database schema, then the `PUBLIC` keyword is used. When a DBUri is accessed, the table name resolves to the same table, synonym, or database view that was visible by that name when the DBUri was created.
- *Optional* **`text()`** *argument:* By default, `DBURIType` constructs an XML document. Use `text()` as the third argument to `sys_DburiGen` to create a DBUri that targets a text node (no XML elements). For example:

  ```
  SELECT sys_DburiGen(employee_id, last_name, 'text()') FROM hr.employees,
    WHERE employee_id=200;
  ```

  This constructs a DBUri with the following URI:

  ```
  /HR/EMPLOYEES/ROW[EMPLOYEE_ID=200]/LAST_NAME/text()
  ```

- *Single-column argument:* If there is a single-column argument, then the column is used as both the key column to identify the row and the referenced column.

The query in Example 20–14 uses `employee_id` as both the key column and the referenced column. It generates a DBUri that targets the row with `employee_id` 200.

Example 20–14 Passing Columns with Single Arguments to SYS_DBURIGEN

```
SELECT sys_DburiGen(employee_id) FROM employees
  WHERE employee_id=200;

SYS_DBURIGEN(EMPLOYEE_ID)(URL, SPARE)
-------------------------------------
DBURITYPE('/PUBLIC/EMPLOYEES/ROW[EMPLOYEE_ID=''200'']/EMPLOYEE_ID', NULL)

1 row selected.
```

SYS_DBURIGEN SQL Function: Examples

Example 20–15 Inserting Database References using SYS_DBURIGEN

```
CREATE TABLE doc_list_tab (docno NUMBER PRIMARY KEY, doc_ref SYS.DBURIType);
Table created.

-- Insert a DBUri that targets the row with employee_id=177
INSERT INTO doc_list_tab VALUES(1001, (SELECT sys_DburiGen(rowid, employee_id)
                                        FROM employees WHERE employee_id=177));
1 row created.

-- Insert a DBUri that targets the last_name column of table employees
INSERT INTO doc_list_tab VALUES(1002,
                                (SELECT sys_DburiGen(employee_id, last_name)
                                   FROM employees WHERE employee_id=177));
1 row created.

SELECT * FROM doc_list_tab;

     DOCNO
----------
DOC_REF(URL, SPARE)
---------------------------------------------------
      1001
DBURITYPE('/PUBLIC/EMPLOYEES/ROW[ROWID=''AAAQCcAAFAAAABSABN'']/EMPLOYEE_ID', NULL)

      1002
DBURITYPE('/PUBLIC/EMPLOYEES/ROW[EMPLOYEE_ID=''177'']/LAST_NAME', NULL)

2 rows selected.
```

Returning Partial Results

When selecting from a large column, you might sometimes want to retrieve only a portion of the result, and create a URL to the column instead. For example, consider the case of a travel story Web site. If travel stories are stored in a table, and users search for a set of relevant stories, then you do not want to list each entire story in the search-result page. Instead, you might show just the first 20 characters of each story, to represent the gist, and then return a URL to the full story. This can be done as follows:

Example 20–16 creates the travel story table.

Example 20–16 Creating the Travel Story Table

```
CREATE TABLE travel_story (story_name VARCHAR2(100), story CLOB);
Table created.

INSERT INTO travel_story
  VALUES ('Egypt', 'This is the story of my time in Egypt....');
1 row created.
```

Example 20–17 creates a function that returns only the first 20 characters from the story.

Example 20–17 A Function that Returns the First 20 Characters

```
CREATE OR REPLACE FUNCTION charfunc(clobval IN CLOB) RETURN VARCHAR2 IS
  res VARCHAR2(20);
  amount NUMBER := 20;
```

```
BEGIN
  DBMS_LOB.read(clobval, amount, 1, res);
  RETURN res;
END;
/
Function created.
```

Example 20–18 creates a view that selects only the first twenty characters from the travel story, and returns a DBUri to the story column.

Example 20–18 Creating a Travel View for Use with SYS_DBURIGEN

```
CREATE OR REPLACE VIEW travel_view AS
  SELECT story_name, charfunc(story) short_story,
         sys_DburiGen(story_name, story, 'text()') story_link
  FROM travel_story;
View created.

SELECT * FROM travel_view;

STORY_NAME
----------
SHORT_STORY
-----------
STORY_LINK(URL, SPARE)
----------------------
Egypt
This is the story of
DBURITYPE('/PUBLIC/TRAVEL_STORY/ROW[STORY_NAME=''Egypt'']/STORY/text()', NULL)

1 row selected.
```

RETURNING URLs to Inserted Objects

You can use Oracle SQL function `sys_DburiGen` in the `RETURNING` clause of DML statements to retrieve the URL of an object as it is inserted.

In Example 20–19, whenever a document is inserted into table `clob_tab`, its URL is inserted into table `uri_tab`. This is done using Oracle SQL function `sys_DburiGen` in the `RETURNING` clause of the `INSERT` statement.

Example 20–19 Retrieving a URL using SYS_DBURIGEN in RETURNING Clause

```
CREATE TABLE clob_tab (docid NUMBER, doc CLOB);
Table created.

CREATE TABLE uri_tab (docs SYS.DBURIType);
Table created.
```

In PL/SQL, specify the storage of the URL of the inserted document as part of the insertion operation, using the `RETURNING` clause and `EXECUTE IMMEDIATE`:

```
DECLARE
  ret SYS.DBURIType;
BEGIN
  -- execute the insert operation and get the URL
  EXECUTE IMMEDIATE
    'INSERT INTO clob_tab VALUES (1, ''TEMP CLOB TEST'')
       RETURNING sys_DburiGen(docid, doc, ''text()'') INTO :1'
    RETURNING INTO ret;
  -- Insert the URL into uri_tab
```

```
  INSERT INTO uri_tab VALUES (ret);
END;
/

SELECT e.docs.getURL() FROM hr.uri_tab e;
E.DOCS.GETURL()
-----------------------------------------------
/ORADB/PUBLIC/CLOB_TAB/ROW[DOCID='1']/DOC/text()

1 row selected.
```

DBUriServlet

Oracle XML DB Repository resources can be retrieved using the HTTP server that is incorporated in Oracle XML DB. Oracle Database also includes a servlet, **DBUriServlet**, that makes any kind of database data available through HTTP(S) URLs. The data can be returned as plain text, HTML, or XML.

A Web client or application can access such data without using SQL or a specialized database API. You can retrieve the data by linking to it on a Web page or by requesting it through HTTP-aware APIs of Java, PL/SQL, and Perl. You can display or process the data using an application such as a Web browser or an XML-aware spreadsheet. DBUriServlet can generate content that is XML data or not, and it can transform the result using XSLT style sheets.

You make database data Web-accessible by using a URI that is composed of a servlet address (URL) plus a DBUri URI that specifies which database data to retrieve. This is the syntax, where `http://server:port` is the URL of the servlet (server and port), and `/oradb/database_schema/table` is the DBUri URI (any DBUri URI can be used):

```
http://server:port/oradb/database_schema/table
```

When using XPath notation in a URL for the servlet, you might need to escape certain characters. You can use `URIType` PL/SQL method `getExternalURL()` to do this.

You can either use `DBUriServlet`, which is pre-installed as part of Oracle XML DB, or write your own servlet that runs on a servlet engine. The servlet reads the URI portion of the invoking URL, creates a DBUri using that URI, calls `URIType` PL/SQL methods to retrieve the data, and returns the values in a form such as a Web page, an XML document, or a plain-text document.

The MIME type to use is specified to the servlet through the URI:

- By default, the servlet produces MIME types `text/xml` and `text/plain`. If the DBUri path ends in `text()`, then `text/plain` is used. Otherwise, an XML document is generated with MIME type `text/xml`.
- You can override the default MIME type, setting it to `binary/x-jpeg` or some other value, by using the `contenttype` argument to the servlet.

> **See Also:** Chapter 32, "Writing Oracle XML DB Applications in Java", for information about Oracle XML DB servlets

Table 20–3 describes each of the optional URL parameters you can pass to DBUriServlet to customize its output.

Table 20–3 DBUriServlet: Optional Arguments

Argument	Description
rowsettag	Changes the default root tag name for the XML document. For example: http://*server*:8080/oradb/HR/EMPLOYEES**?rowsettag=OracleEmployees**
contenttype	Specifies the MIME type of the generated document. For example: http://*server*:8080/oradb/HR/EMPLOYEES**?contenttype=text/plain**
transform	Passes a URL to URIFACTORY, which retrieves the XSL style sheet at that location. This style sheet is then applied to the XML document being returned by the servlet. For example: http://*server*:8080/oradb/HR/EMPLOYEES**?transform=/oradb/QUINE/XSLS/DOC/text()**&contenttype=text/html

Overriding the MIME Type using a URL

To retrieve the employee_id column of the employee table, you can use a URL such as one of the following, where computer *server.oracle.com* is running Oracle Database with a Web service listening to requests on port 8080. Step oradb is the virtual path that maps to the servlet.

http://*server.oracle.com*:8080/oradb/QUINE/A/ROW[B=200]/C/text()
Produces a content type of text/plain

http://*server.oracle.com*:8080/oradb/QUINE/A/ROW[B=200]/C
Produces a content type of text/xml

To override the content type, you can use a URL that passes text/html to the servlet as the contenttype parameter:

http://*server.oracle.com*:8080/oradb/QUINE/A/ROW[B=200]/C**?contenttype=text/html**
Produces a content type of text/html

Customizing DBUriServlet

DBUriServlet is built into the database – to customize the servlet, you must edit the Oracle XML DB configuration file, xdbconfig.xml. You can edit it with database schema (user account) XDB, using WebDAV, FTP, Oracle Enterprise Manager, or PL/SQL. To update the file using FTP or WebDAV, download the document, edit it, and save it back into the database.

See Also:

- Chapter 32, "Writing Oracle XML DB Applications in Java"
- Chapter 34, "Administering Oracle XML DB"
- *Oracle Database 2 Day + Security Guide* for information about database schema XDB

DBUriServlet is installed at /oradb/*, which is the address specified in the servlet-pattern tag of xdbconfig.xml. The asterisk (*) is necessary to indicate that any path following oradb is to be mapped to the same servlet. oradb is published as the virtual path. You can change the path that is used to access the servlet.

In Example 20–20, the configuration file is modified to install DBUriServlet under /dburi/*.

Example 20–20 Changing the Installation Location of DBUriServlet

```
DECLARE
  doc XMLType;
  doc2 XMLType;
BEGIN
  doc := DBMS_XDB.cfg_get();
  SELECT
    updateXML(doc,
'/xdbconfig/sysconfig/protocolconfig/httpconfig/webappconfig/servletconfig/
servlet-mappings/servlet-mapping[servlet-name="DBUriServlet"]/servlet-pattern/
text()',
              '/dburi/*')
    INTO doc2 FROM DUAL;
  DBMS_XDB.cfg_update(doc2);
  COMMIT;
END;
/
```

Security parameters, the servlet display-name, and the description can also be customized in configuration file `xdbconfig.xml`. The servlet can be removed by deleting its servlet-pattern. This can also be done using Oracle SQL function `updateXML` to update the servlet-mapping element to `NULL`.

DBUriServlet Security

Servlet security is handled by Oracle Database using roles. When users log in to the servlet, they use their database user name and password. The servlet checks to ensure that the user logging has one of the roles specified in the configuration file using parameter `security-role-ref`). By default, the servlet is available to role `authenticatedUser`, and any user who logs into the servlet with a valid database password has this role.

The role parameter can be changed to restrict access to any specific database roles. To change from the default `authenticatedUser` role to a role that you have created, you modify the Oracle XML DB configuration file.

Example 20–21 changes the default role `authenticatedUser` to role `servlet-users` (which you must have created).

Example 20–21 Restricting Servlet Access to a Database Role

```
DECLARE
  doc  XMLType;
  doc2 XMLType;
  doc3 XMLType;
BEGIN
  doc := DBMS_XDB.cfg_get();
  SELECT updateXML(doc,
'/xdbconfig/sysconfig/protocolconfig/httpconfig/webappconfig/servletconfig/
servlet-list/servlet[servlet-name="DBUriServlet"]/security-role-ref/role-name/
text()',
                   'servlet-users')
    INTO doc2 FROM DUAL;
  SELECT updateXML(doc2,
'/xdbconfig/sysconfig/protocolconfig/httpconfig/webappconfig/servletconfig/
servlet-list/servlet[servlet-name="DBUriServlet"]/security-role-ref/role-link/
text()',
                   'servlet-users')
    INTO doc3 FROM DUAL;
```

```
  DBMS_XDB.cfg_update(doc3);
  COMMIT;
END;
/
```

Configuring Package URIFACTORY to Handle DBUris

A URL such as `http://server/servlets/`**`oradb`** is handled by DBUriServlet (or by a custom servlet). When a URL such as this is stored as a `URIType` instance, it is generally desirable to use subtype `DBURIType`, since this URI targets database data.

However, if a `URIType` instance is created using PL/SQL methods of package `URIFACTORY`, then, by default, the subtype used is `HTTPURIType`, not `DBURIType`. This is because `URIFACTORY` looks only at the URI prefix, sees `http://`, and assumes that the URI targets a Web page. This results in unnecessary layers of communication and perhaps extra character conversions.

To make things more efficient, you can teach `URIFACTORY` that URIs of the given form represent database accesses and so should be realized as DBUris, not HTTPUris. You do this by registering a handler for this URI as a prefix, specifying `DBURIType` as the type of instance to generate.

Example 20–22 effectively tells `URIFACTORY` that any URI string starting with `http://server/servlets/oradb` corresponds to a database access.

Example 20–22 Registering a Handler for a DBUri Prefix

```
BEGIN
  URIFACTORY.registerURLHandler('http://server/servlets/oradb',
                              'SYS', 'DBURIType', true, true);
END;
/
```

After you execute this code, all `getURI()` calls in the same session automatically create DBUris for any URI strings with prefix `http://server/servlets/oradb`.

> **See Also:** *Oracle Database PL/SQL Packages and Types Reference* for information about `URIFACTORY` functions

Part V

Oracle XML DB Repository

Part V of this manual describes Oracle XML DB repository. It includes how to version your data, implement and manage security, and how to use the associated Oracle XML DB APIs to access and manipulate repository data.

Part V contains the following chapters:

- Chapter 21, "Accessing Oracle XML DB Repository Data"
- Chapter 22, "Configuring Oracle XML DB Repository"
- Chapter 23, "Using XLink and XInclude with Oracle XML DB"
- Chapter 24, "Managing Resource Versions"
- Chapter 25, "Accessing the Repository using RESOURCE_VIEW and PATH_VIEW"
- Chapter 26, "Accessing the Repository using PL/SQL"
- Chapter 27, "Repository Access Control"
- Chapter 28, "Accessing the Repository using Protocols"
- Chapter 29, "User-Defined Repository Metadata"
- Chapter 30, "Oracle XML DB Repository Events"
- Chapter 31, "Using Oracle XML DB Content Connector"
- Chapter 32, "Writing Oracle XML DB Applications in Java"
- Chapter 33, "Using Native Oracle XML DB Web Services"

21

Accessing Oracle XML DB Repository Data

This chapter describes how to access data in Oracle XML DB Repository using standard protocols such as FTP and HTTP(S)/WebDAV, and other Oracle XML DB resource Application Program Interfaces (APIs). It also introduces you to using `RESOURCE_VIEW` and `PATH_VIEW` as the SQL mechanism for accessing and manipulating repository data. It includes a table for comparing repository operations through the various resource APIs.

This chapter contains these topics:

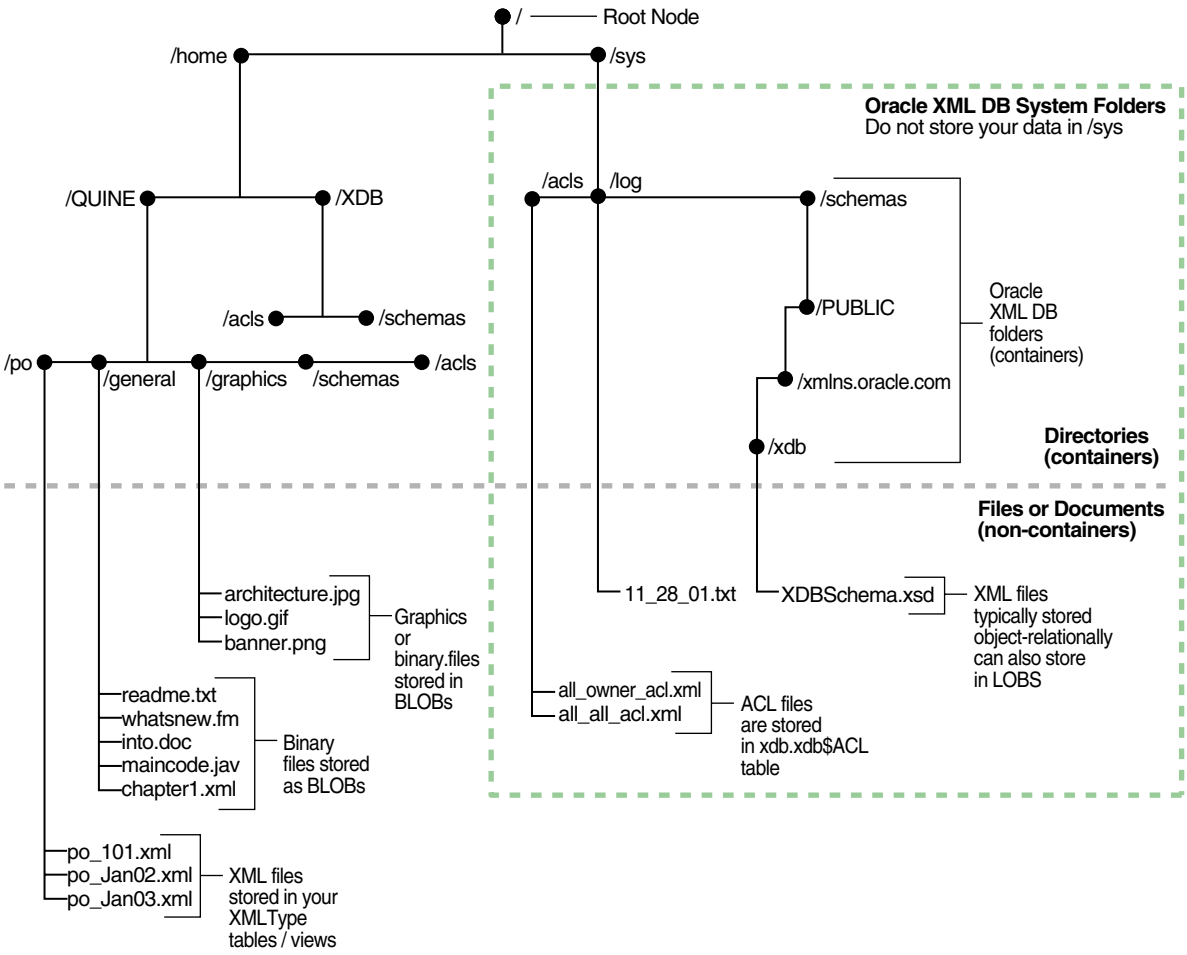
- Overview of Oracle XML DB Repository
- Repository Terminology and Supplied Resources
- Oracle XML DB Repository Resources
- Two Ways to Access Oracle XML DB Repository Resources
- Navigational or Path Access to Repository Resources
- Query-Based Access to Repository Resources
- Servlet Access to Repository Resources
- Operations on Repository Resources

Overview of Oracle XML DB Repository

Using Oracle XML DB Repository you can store content in the database in hierarchical structures, as opposed to traditional relational database structures.

Figure 21–1 is an example of a hierarchical structure that shows a typical tree of folders and files in Oracle XML DB Repository. The top of the tree shows `/`, the root folder.

Figure 21–1 A Folder Tree, Showing Hierarchical Structures in the Repository



Note: Folder `/sys` is used by Oracle XML DB to maintain *system-defined* XML schemas, access control lists (ACLs), and so on. Do *not* add or modify any data in folder `/sys`.

Foldering lets applications access hierarchically indexed content in the database using the FTP, HTTP(S), and WebDAV protocol standards as if the database content were stored in a file system. You can set access control privileges on repository folders and resources.

This chapter provides an overview of how to access data in Oracle XML DB Repository folders using the standard protocols. It discusses APIs that you can use to access the repository object hierarchy using Java, SQL, and PL/SQL.

See Also:

- Chapter 25, "Accessing the Repository using RESOURCE_VIEW and PATH_VIEW"
- Chapter 26, "Accessing the Repository using PL/SQL"
- Chapter 27, "Repository Access Control"
- Chapter 28, "Accessing the Repository using Protocols"

Two Ways to Access Oracle XML DB Repository Resources

There are two ways to access Oracle XML DB Repository resources:

- Navigational or path-based access. This uses a hierarchical index of resources. Each resource has one or more unique path names that reflect its location in the hierarchy. You can navigate, using XPath expressions, to any repository resource.

  A repository resource can be created as a reference to an existing `XMLType` object in the database. You can navigate to any such database object using XPath. See "Navigational or Path Access to Repository Resources" on page 21-9.

- SQL access to the repository. This is done using special views that expose resource properties and path names, and map hierarchical access operators onto the Oracle XML DB schema. See "Query-Based Access to Repository Resources" on page 21-13.

  See Also:

  - "Oracle XML DB Repository Access" on page 2-3 for guidance on selecting an access method
  - Table 21–3, " Accessing Oracle XML DB Repository: API Options" for a summary comparison of the access methods

A Uniform Resource Locator (URL) is used to access an Oracle XML DB resource. A URL includes the host name, protocol information, path name, and resource name of the object.

Repository Terminology and Supplied Resources

Oracle XML DB Repository is the set of database objects, across all XML and database schemas, that are mapped to path names. It is a connected, directed, acyclic[1] graph of resources, with a single root node (`/`). Each resource in the graph has one or more associated path names: the repository supports multiple links to a given resource. The repository can be thought of as a file system of objects rather than files.

Repository Terminology

The following list describes terms used in Oracle XML DB Repository:

- **resource** – Any object or node in the repository hierarchy. Resources are identified by URLs.

  See Also:

  "Overview of Oracle XML DB Repository" on page 1-7

  "Oracle XML DB Repository Resources" on page 21-5

- **folder** – A resource that can contain other resources. Sometimes called a **directory**.
- **path name** – A hierarchical name representing an absolute path to a resource. It is composed of a slash (`/`) representing the repository root, followed by zero or more **path components** separated by slashes. A path component cannot be only `.` or `..`, but a period (`.`) can otherwise be used in a path component. A path component is composed of any characters in the database character set *except* slash (`/`), backslash

[1] The graph is established by the hard links that define the repository structure, and cycles are not permitted using hard links. You can, however, introduce cycles using weak links. See "Hard Links and Weak Links" on page 21-7.

(\), and those characters specified in the Oracle XML DB configuration file, `/xdbconfig.xml`, by configuration parameter `/xdbconfig/sysconfig/invalid-pathname-chars`.

- **resource name** (or **link name**) – The name of a resource within its parent folder. This is the rightmost path component of a path name. Resource names must be unique within their immediately containing folder, and they are case-sensitive.
- **resource content** – The body, or data, of a resource. This is what you get when you treat the resource as a file and ask for its content. This is always of type `XMLType`.
- **XDBBinary element** – An XML element that contains binary data. It is defined by the Oracle XML DB XML schema. `XDBBinary` elements are stored in the repository whenever unstructured binary data is uploaded into Oracle XML DB.
- **access control list** (ACL) – A set of principals (users or roles) that are allowed access to one or more specific resources.

See Also: Chapter 27, "Repository Access Control"

Many terms used by Oracle XML DB have common synonyms in other contexts, as shown in Table 21–1.

Table 21–1 Synonyms for Oracle XML DB Repository Terms

Synonym	Repository Term	Usage
collection	folder	WebDAV
directory	folder	operating systems
privilege	privilege	permission
right	privilege	various
WebDAV folder	folder	Web folder
role	group	access control
revision	version	RCS, CVS
file system	repository	operating systems
hierarchy	repository	various
file	resource	operating systems
binding	link	WebDAV

Supplied Files and Folders

The list of supplied Oracle XML DB Repository files and folders is as follows. In addition to using these, you can create your own folders and files wherever you want.

```
/public
/sys
/sys/acls
/sys/acls/all_all_acl.xml
/sys/acls/all_owner_acl.xml
/sys/acls/bootstrap_acl.xml
/sys/acls/ro_all_acl.xml
/sys/apps
/sys/asm
/sys/log
/sys/schemas
/sys/schemas/PUBLIC
```

```
/sys/schemas/PUBLIC/www.w3.org
/sys/schemas/PUBLIC/www.w3.org/2001
/sys/schemas/PUBLIC/www.w3.org/2001/xml.xsd
/sys/schemas/PUBLIC/xmlns.oracle.com
/sys/schemas/PUBLIC/xmlns.oracle.com/xdb
/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/XDBFolderListing.xsd
/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/XDBResource.xsd
/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/XDBSchema.xsd
/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/XDBStandard.xsd
/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/acl.xsd
/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/dav.xsd
/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/log
/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/log/ftplog.xsd
/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/log/httplog.xsd
/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/log/xdblog.xsd
/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/stats.xsd
/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/xdbconfig.xsd
/xdbconfig.xml
```

Oracle XML DB Repository Resources

Oracle XML DB Repository resources conform to the Oracle XML DB XML schema `XDBResource.xsd`. The elements in a resource include those needed to persistently store WebDAV-defined properties, such as creation date, modification date, WebDAV locks, owner, ACL, language, and character set.

> **See Also:** "XDBResource.xsd: XML Schema for Oracle XML DB Resources" on page A-1

A resource index has a special element called **`Contents`** that contains the contents of the resource.

The XML schema for a resource also defines an **`any`** element, with `maxoccurs` attribute unbounded. An `any` element can contain any element outside of the Oracle XML DB XML namespace. Arbitrary instance-defined properties can be associated with the resource.

Where Is Repository Data Stored?

Oracle XML DB stores Oracle XML DB Repository data in a set of tables and indexes to which you have access. If you register an XML schema and request that the tables be generated by Oracle XML DB, then the tables are created in your database schema. You are then able to see or modify them. Other users cannot see your tables unless you grant them permission to do so.

Names of Generated Tables

The names of the generated tables are assigned by Oracle XML DB, and can be obtained by finding the `xdb:defaultTable` attribute in your XML schema document (or in the default XML schema document). When you register an XML schema, you can alternatively provide your own table name, instead of using the default name supplied by Oracle XML DB. If the table specifies binary XML storage, then a document is encoded in binary XML format before storing it in the table.

> **See Also:** "Default Tables Created During XML Schema Registration" on page 7-10

Defining Structured Storage for Resources

Applications that need to define structured storage for resources can do so in one of these ways:

- Subclass the Oracle XML DB resource type. Subclassing Oracle XML DB resources requires privileges on the table `XDB$RESOURCE`.
- Store data that conforms to a visible, registered XML schema.

See Also: Chapter 7, "XML Schema Storage and Query: Basic"

Oracle ASM Virtual Folder

The Oracle Automatic Storage Management (Oracle ASM) virtual folder, `/sys/asm`, is an exception to the description of the previous sections: its contents are Oracle ASM files and folders that are managed automatically by Oracle ASM.

See Also:

- "Accessing Oracle ASM Files using Protocols and Resource APIs – For DBAs" on page 21-12
- *Oracle Automatic Storage Management Administrator's Guide*

Path-Name Resolution

The data relating a folder to its contents is managed by the Oracle XML DB hierarchical repository index. This provides a fast mechanism for evaluating path names, similar to the directory mechanisms that are used by operating-system file systems.

Resources that are folders have the `Container` attribute of element `Resource` set to `true`.

To resolve a resource name in a folder, the current user must have the following privileges:

- `resolve` privilege on the folder
- `read-properties` on the resource in that folder

If the user does not have these privileges, then the user receives an `access denied` error. Folder listings and other queries do not return a row when the `read-properties` privilege is denied on its resource.

Caution: Error handling in path-name resolution differentiates between invalid resource names and resources that are not folders, for compatibility with file systems. Because Oracle XML DB resources are accessible from outside Oracle XML DB Repository (using SQL), denying read access on a folder that contains a resource does *not prevent* read access to that resource.

See Also: "XDBResource.xsd: XML Schema for Oracle XML DB Resources" on page A-1 for the definition of element `Resource` and its attribute `Container`

Link Types

Links in Oracle XML DB can be repository links or document links. Repository links can be hard links or weak links. Document links can also be hard links or weak links, when their targets are repository resources. These terms are explained further in the following sections.

Repository and Document Links

In addition to containing resources, a folder resource can contain links to other resources (files or folders). These **repository links**, sometimes called **folder links**, are not to be confused with **document links**, which correspond to the links provided by the XLink and XInclude standards, and which are also supported by Oracle XML DB. Repository links are navigational, folder–child links among repository resources. Document links are arbitrary links among documents that are not necessarily repository resources.

Repository links represent repository hierarchical relationships. Document links represent arbitrary relationships whose semantics derives from the applications that use them. Because they represent repository hierarchical relationships, repository links can be navigated using file system-related protocols. This is not true of document links. Because document links can represent arbitrary relationships, they can also represent repository relationships. When document links thus target resources, they can also be hard or weak.

> **See Also:** Chapter 23, "Using XLink and XInclude with Oracle XML DB" for information about document links

Hard Links and Weak Links

Links that target repository resources can be **hard links** or **weak links**. Both hard and weak links are references, or pointers, to physical data—(internal) repository resource identifiers. They do not point to symbolic names or paths to other links. Their targets are resolved at the time of link creation. Because they point directly to resource identifiers, hard and weak links cannot dangle: they remain valid even when their targets are renamed or moved. You need the same privileges to create or delete hard and weak links.

The difference between hard and weak links lies in their relationship to target resource deletion. A target resource is dependent on its hard links, in the sense that it cannot be deleted as long as it remains the target of a hard link. Deletion of a hard link also deletes the resource targeted by the link, if the following are both true:

- The resource is not versioned.
- The hard link that was deleted was the last (that is, the only) hard link to the resource.

A weak link has no such hold on a resource: you can delete a resource, even if it is the target of a weak link (as long as it is not the target of a hard link). Because of this, weak links can be used as shortcuts to frequently accessed resources, without impacting deletion of those resources.

There is a dependency in the other direction, however: If you delete a resource that is the target of one or more weak links, then those links are automatically deleted, as well. In this sense, too, weak links cannot dangle. Both hard and weak links provide referential integrity: if a link exists, then so does its target.

Another difference between hard and weak links is this: Hard links to ancestor folders are not permitted, because they introduce cycles. There is no such restriction for weak

links: a weak link can target any folder, possibly creating a cycle. It is the set of hard links that define the (acyclic) structure of Oracle XML DB Repository. Weak links represent an additional mapping on top of that basic structure.

You can query the repository path view, `PATH_VIEW`, to determine the type of a repository link: the link information contains the link type. `XMLType` column `LINK` of `PATH_VIEW` contains this information in element `LinkType`, which is a child of the root element, `LINK`. Example 21–1 illustrates this. You can also determine the type of a repository link by using the `getLink()` callback function on an event handler (`LinkIn`, `LinkTo`, `UnlinkIn`, or `UnlinkFrom`).

Example 21–1 Querying PATH_VIEW to Determine Link Type

```
SELECT RESID, XMLCast(XMLQuery('/LINK/LinkType' PASSING LINK RETURNING CONTENT)
                      AS VARCHAR2(24)) link_type
  FROM PATH_VIEW WHERE equals_path(RES, '/home/QUINE/purchaseOrder.xml') = 1;

RESID                              LINK_TYPE
--------------------------------   ---------
DF9856CF2FE0829EE030578CCE0639C5   Weak
```

See Also:

- "Deleting Repository Resources: Examples" on page 25-14
- "Query-Based Access to Repository Resources" on page 21-13 for information about `PATH_VIEW`
- *Oracle Database PL/SQL Packages and Types Reference* for information on PL/SQL function `getLink`

Creating a Weak Link with No Knowledge of Folder Hierarchy

Suppose that you want to read a file resource that belongs to one of your colleagues. You cannot create a hard link to that resource, to make it accessible for your use, unless you have the privilege `<xdb:resolve>` on *all* of the ancestor folders of that file. Having that privilege would mean that you could see all of your colleague's folder names and the structure of the hierarchy down to the target resource.

However, because weak links essentially represent a mapping on top of the real repository structure, which structure is determined by the set of hard links, you can create a weak link to a resource using just its OID rather than its full, named path (URL). Your colleague can determine the OID path to the file, send you that instead of the named path, and you can create a weak link to the document using that OID path. Example 21–2 and Example 21–3 illustrate this.

Example 21–2 prints the OID path for the file resource `/home/QUINE/purchaseOrder.xml`. User `quine` can use this to obtain the OID path to the resource, and then send that path to user `curry`, who can create a weak link to the resource (Example 21–3).

Example 21–2 Obtaining the OID Path of a Resource

```
DECLARE
  resoid  RAW(16);
  oidpath VARCHAR2(100);
BEGIN
  SELECT RESID INTO :resoid FROM RESOURCE_VIEW
    WHERE equals_path(RES, '/home/QUINE/purchaseOrder.xml') = 1;
    oidpath := DBMS_XDB.createOIDPath(resoid)
  DBMS_OUTPUT.put_line(oidpath);
```

```
END;
```

In Example 21–3, user `curry` creates a weak link named `quinePurchaseOrder.xml` in folder `/home/CURRY`. The target of the link is the OID path that corresponds to the URL `/home/QUINE/purchaseOrder.xml`. User `curry` need not be aware of the repository structure that is visible to user `quine`.

Example 21–3 Creating a Weak Link using an OID Path

```
CALL DBMS_XDB.Link(/sys/oid/1BDCB46477B59C20E040578CCE0623D3
                   '/home/CURRY', 'quinePurchaseOrder.xml',
                   DBMS_XDB.LINK_TYPE_WEAK);
```

Restricting Multiple Hard Links

Sometimes, it is useful to restrict the creation of hard links, disallowing multiple hard links to folders or files (or both). In particular, allowing multiple hard links to file resources, but disallowing multiple hard links to folder resources, provides behavior that is similar to that for some file systems, including UNIX and Linux. This can simplify application design, by, in effect, ensuring that each file resource has a unique, canonical hard-link path to it. In addition, preventing multiple hard links to a resource can lead to query performance improvements.

You can configure the prevention of multiple hard links using the following Boolean parameters in configuration file `xdbconfig.xml`. The default value of each parameter is `true`, meaning that multiple hard links can be created.

- `folder-hard-links` – Prevent the creation of multiple hard links to a folder resource, if `false`.
- `non-folder-hard-links` – Prevent the creation of multiple hard links to a file resource, if `false`.

> **See Also:** "Configuring Oracle XML DB using xdbconfig.xml" on page 34-5

Navigational or Path Access to Repository Resources

Oracle XML DB Repository folders support the same protocol standards used by many operating systems. This lets a repository folder act like a native folder (directory) in supported operating-system environments. For example, you can:

- Use Windows Explorer to open and access Oracle XML DB folders and resources the same way you access other directories or resources in the Windows NT file system, as shown in Figure 21–2.
- Access Oracle XML DB Repository data using HTTP(S)/WebDAV from an Internet Explorer browser, such as when viewing Web Folders, as shown in Figure 21–3. Figure 21–3 shows a Web browser visiting URL `http://xdbdemo:8080/`: the server it is connected to is `xdbdemo`, and its HTTP port number is `8080`.

> **See Also:** "Configuring Protocol Server Parameters" on page 28-3 for information about configuring the HTTP port number

Figure 21–2 Oracle XML DB Folders in Windows Explorer

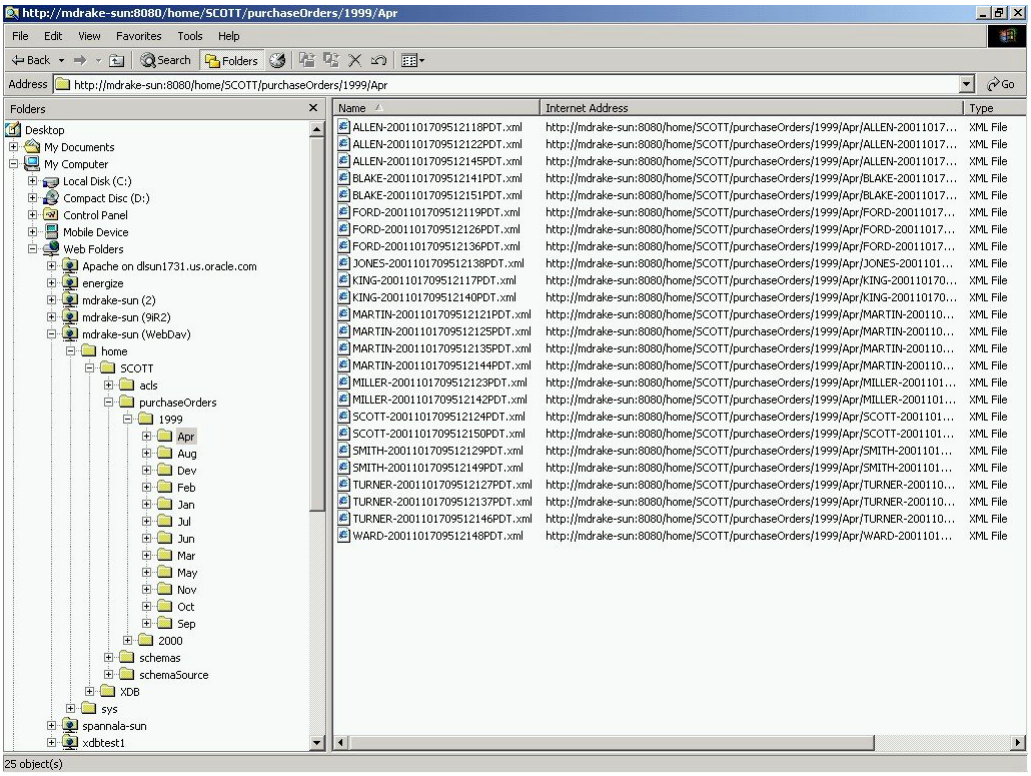
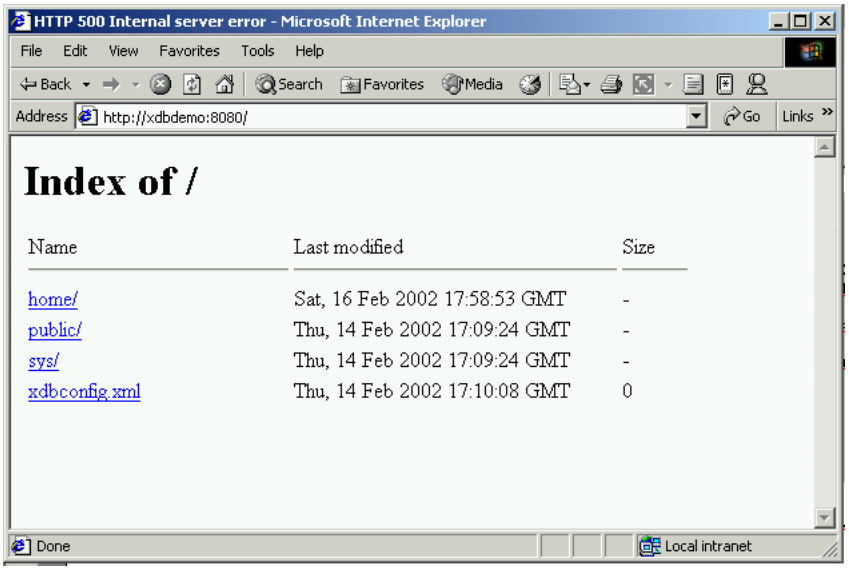


Figure 21–3 Accessing Repository Data using HTTP(S)/WebDAV and Navigational Access From IE Browser: Viewing Web Folders



Accessing Oracle XML DB Resources using Internet Protocols

Oracle Net Services provides one way of accessing database resources. Oracle XML DB support for Internet protocols provides another way of accessing database resources.

Where You Can Use Oracle XML DB Protocol Access

Oracle Net Services is optimized for record-oriented data. Internet protocols are designed for stream-oriented data, such as binary files or XML text documents. Oracle XML DB protocol access is a valuable alternative to Net Services in the following scenarios:

- Direct database access from file-oriented applications using the database like a file system
- Heterogeneous application server environments that require a uniform data access method (such as XML over HTTP, which is supported by most data servers, including MS SQL Server, Exchange, Notes, many XML databases, stock quote services and news feeds)
- Application server environments that require data in the form of XML text
- Web applications that use client-side XSL to format datagrams that do not need much application processing
- Web applications that use Java servlets that run inside the database
- Web access to XML-oriented stored procedures

Using Protocol Access

Accessing Oracle XML DB using a protocol proceeds as follows:

1. A connection object is established, and the protocol might read part of the request.
2. The protocol decides whether the user is already authenticated and wants to reuse an existing session or the connection must be re-authenticated (the latter is more common).
3. An existing session is pulled from the session pool, or else a new one is created.
4. If authentication has not been provided, and the request is HTTP `get` or `head`, then the session is run as the `ANONYMOUS` user. If the session has already been authenticated as the `ANONYMOUS` user, then there is no cost to reuse the existing session. If authentication has been provided, then the database re-authentication routines are used to authenticate the connection.
5. The request is parsed.
6. (HTTP only) If the requested path name maps to a servlet, then the servlet is invoked using Java Virtual Machine (VM). The servlet code writes the response to a response stream or asks `XMLType` instances to do so.

Retrieving Oracle XML DB Resources

When the protocol indicates that a resource is to be retrieved, the path name to the resource is resolved. Resources being fetched are always streamed out as XML, with the exception of resources containing the `XDBBinary` element, an element defined to be the XML binary data type, which have their contents streamed out in RAW form.

Storing Oracle XML DB Resources

When the protocol indicates that a resource must be stored, Oracle XML DB checks the document file name extension for `.xml`, `.xsl`, `.xsd`, and so on. If the document is XML, then a pre-parse step is done, whereby enough of the resource is read to determine the XML `schemaLocation` and `namespace` of the root element in the document. If a registered schema is located at the `schemaLocation` URL, and it has a

definition for the root element of the current document, then the default table specified for that root element is used to store the contents of the resource.

Using Internet Protocols and XMLType: XMLType Direct Stream Write

Oracle XML DB supports Internet protocols at the `XMLType` level by using Java `XMLType` method `writeToStream()`. This method is implemented natively and writes `XMLType` data directly to the protocol request stream. This avoids Java VM execution costs and the overhead of converting database data through Java data types and creating Java objects, resulting in significantly higher performance. Performance is further enhanced if the Java code deals only with XML element trees that are close to the root, and does not traverse too many of the leaf elements, so that relatively few Java objects are created.

See Also: Chapter 28, "Accessing the Repository using Protocols"

Accessing Oracle ASM Files using Protocols and Resource APIs – For DBAs

Oracle Automatic Storage Management (Oracle ASM) organizes database files into *disk groups* for simplified management and added benefits such as database mirroring and I/O balancing.

Repository access using protocols and resource APIs (such as `DBMS_XDB`) extends to Oracle ASM files. These files are accessed in the *virtual* repository folder `/sys/asm`. However, this access is reserved for database administrators (DBAs). It is *not* intended for developers.

A typical use of such access is to copy Oracle ASM files from one database instance to another. For example, a DBA can view folder `/sys/asm` in a graphical user interface using the WebDAV protocol, and then drag-and-drop a copy of a data-pump dump set from an Oracle ASM disk group to an operating-system file system.

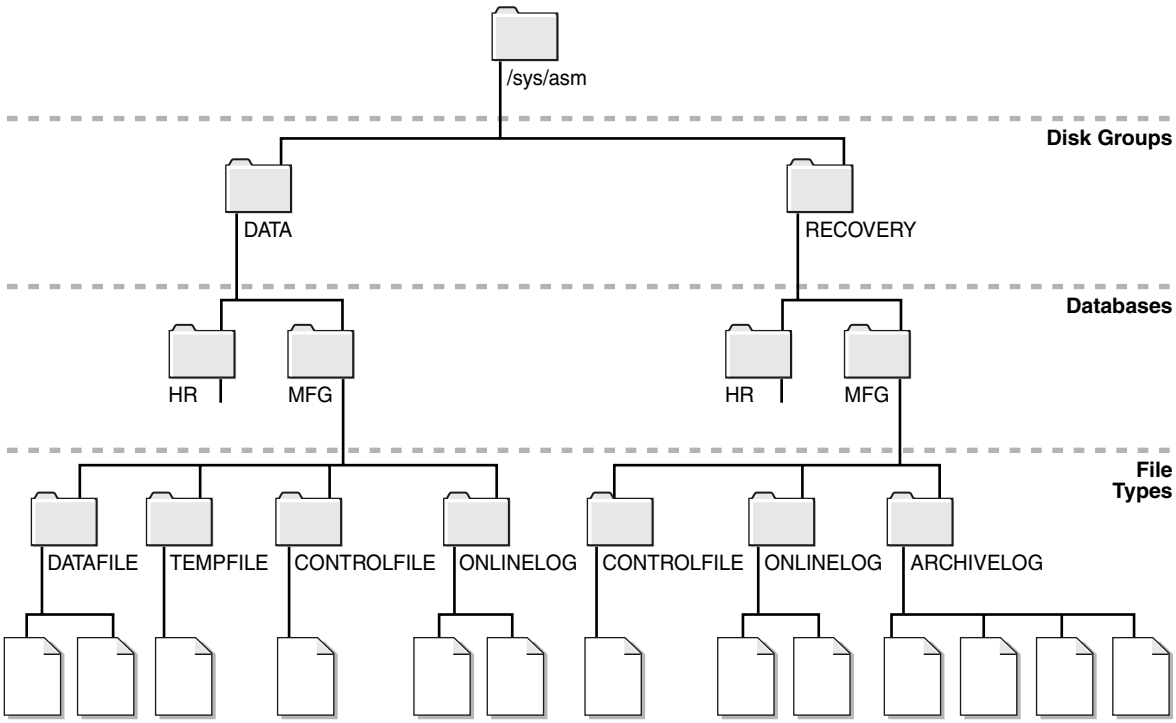
Virtual folder `/sys/asm` is created by default during Oracle XML DB installation. If the database is not configured to use Oracle ASM, the folder is empty and no operations are permitted on it.

Folder `/sys/asm` contains folders and subfolders that follow the hierarchy defined by the structure of an Oracle ASM *fully qualified filename*:

- It contains a subfolder for each mounted *disk group*.
- A disk-group folder contains a subfolder for each *database* that uses that disk group. In addition, a disk-group folder may contain files and folders corresponding to Oracle ASM *aliases* created by the administrator.
- A database folder contains file-type folders.
- A file-type folder contains Oracle ASM files, which are binary.

This hierarchy is shown in Figure 21–4, which omits directories created for aliases, for simplicity.

Figure 21–4 Oracle ASM Virtual Folder Hierarchy



The following usage restrictions apply to virtual folder `/sys/asm`. You *cannot*:

- *query* `/sys/asm` using SQL
- put regular files under `/sys/asm` (you can put only Oracle ASM files there)
- *move* (rename) an Oracle ASM file to a different Oracle ASM disk group or to a folder outside Oracle ASM
- create *hard links* to existing Oracle ASM files or directories

In addition:

- You must have the privileges of role `DBA` to view folder `/sys/asm`.
- To access `/sys/asm` using Oracle XML DB protocols, you must log in as a user other than `SYS`.

Again, Oracle ASM virtual-folder operations are intended only for *database administrators*, not developers.

See Also:

- "Using FTP with Oracle ASM Files" on page 28-11 for an example of using protocol FTP with `/sys/asm`
- *Oracle Automatic Storage Management Administrator's Guide* for information about the syntax of a fully qualified Oracle ASM filename and details on the virtual folder structure

Query-Based Access to Repository Resources

There are two views that enable SQL access to Oracle XML DB Repository data:

- `PATH_VIEW`

- RESOURCE_VIEW

Table 21–2 summarizes the differences between PATH_VIEW and RESOURCE_VIEW.

Table 21–2 Differences Between PATH_VIEW and RESOURCE_VIEW

PATH_VIEW	RESOURCE_VIEW
Contains link properties	No link properties
Has one row for each unique *path* in repository	Has one row for each *resource* in repository

Rows in these two repository views are of XMLType. In the RESOURCE_VIEW, the single path associated with a resource is arbitrarily chosen from among the possible paths that refer to the resource. Oracle XML DB provides SQL functions, such as under_path, that let applications search for the resources contained within a particular folder (recursively), obtain the resource depth, and so on.

DML can be used on the repository views to insert, rename, delete, and update resource properties and contents. Programmatic APIs must be used for other operations, such as creating links to existing resources.

See Also:

- Chapter 25, "Accessing the Repository using RESOURCE_VIEW and PATH_VIEW" for details on SQL access to Oracle XML DB Repository
- Chapter 27, "Repository Access Control"

Servlet Access to Repository Resources

Oracle XML DB implements Java Servlet API, version 2.2, with the following exceptions:

- All servlets must be distributable. They must expect to run in different virtual machines.
- WAR and web.xml files are not supported. Oracle XML DB supports a subset of the XML configurations in this file. An XSL style sheet can be applied to the web.xml to generate servlet definitions. An external tool must be used to create database roles for those defined in the web.xml file.
- JSP (Java Server Pages) support can be installed as a servlet and configured manually.
- HTTPSession and related classes are not supported.
- Only one servlet context (that is, one Web application) is supported.

See Also: Chapter 32, "Writing Oracle XML DB Applications in Java"

Operations on Repository Resources

You can operate on data stored in Oracle XML DB Repository resources using any of the following:

- Oracle XML DB resource APIs for Java
- A combination of Oracle XML DB resource views API and Oracle XML DB resource API for PL/SQL

- Internet protocols (HTTP(S)/WebDAV and FTP) and Oracle XML DB protocol server
- Oracle XML DB Content Connector and, through it, the standard Content Repository API for Java (JCR).

These access methods can be used equivalently. It does not matter how you add content to the repository or retrieve it from there. For example, you can add content to the repository using SQL or PL/SQL and then retrieve it using an Internet protocol,or the other way around.

Table 21–3 lists common Oracle XML DB Repository operations, and describes how these operations can be accomplished using each of several access methods. The table shows functionality common to the different methods, but not all of the methods are equally suited to any particular task. Unless mentioned otherwise, "resource" in this table can be either a file resource or a folder resource.

Table 21–3 also shows the resource privileges that are required for each operation. In addition to the privileges listed in the table, privilege `xdb:read-properties` is required on each resource affected by an operation. Operations that affect the parent folder of a resource, in addition to the resource targeted by the operation, also require privilege `xdb:read-properties` on that parent folder. For example, deleting a resource affects both the resource to delete and its parent folder, so you need privilege `xdb:read-properties` on both the resource and its parent folder.

Table 21–3 Accessing Oracle XML DB Repository: API Options

Data Access	SQL and PL/SQL	Protocols	Resource Privileges Required	JCR Support

Create resource	`DBMS_XDB.createResource()`
	`INSERT INTO PATH_VIEW`
	`VALUES(path, RES,`
	`schemaURL, elem);`
		HTTP: `PUT;`
		FTP: `PUT`
			`DAV::bind` on parent folder	Yes

Update resource contents	`UPDATE RESOURCE_VIEW`
	`SET RES =`
	`updateXML(`
	`RES,`
	`'/Resource/Contents',`
	`lob)`
	`WHERE equals_path(RES, path) > 0`
		HTTP: `PUT;`
		FTP: `PUT`
			`xdb:write-content` on resource	Yes

Update resource properties	`UPDATE RESOURCE_VIEW`
	`SET RES =`
	`updateXML(`
	`RES,`
	`'/Resource/propname1',`
	`newval,`
	`'/Resource/propname2' ...)`
	`WHERE equals_path(RES, path) > 0`
		WebDAV: `PROPPATCH;`	`DAV::write-properties` on resource	Yes

Update resource ACL	`UPDATE RESOURCE_VIEW`
	`SET RES =`
	`updateXML(`
	`RES,`
	`'/ Resource/ACL',`
	`XMLType)`
	`WHERE equals_path(RES, path) > 0`
		not applicable	`DAV::write-acl` on resource	No

Unlink resource (delete if last link)	`DBMS_XDB.deleteResource()`
	`DELETE FROM RESOURCE_VIEW`
	`WHERE equals_path(RES, path) > 0`
		HTTP: `DELETE;`
		FTP: `delete`
			`DAV::unbind` on parent folder
			`xdb:unlink-from` on resource
				Yes

Table 21–3 (Cont.) Accessing Oracle XML DB Repository: API Options

Data Access	SQL and PL/SQL	Protocols	Resource Privileges Required	JCR Support

Forcibly remove all links to resource	`DBMS_XDB.deleteResource()`
	`DELETE FROM PATH_VIEW`
	`WHERE`
	`XMLCast(`
	`XMLQuery(`
	`'declare namespace n1=`
	`"http://xmlns.oracle.com/xdb/XDBRes`
	`ource.xsd"; (: :)`
	`//n1:DisplayName'`
	`PASSING RES RETURNING CONTENT)`
	`AS VARCHAR2(256))`
	`= 'My resource'`
		FTP: `quote rm_rf` *`resource`*	`DAV::unbind` on all parent folders
			`xdb:unlink-from` on resource
				Yes

Move resource	`UPDATE PATH_VIEW SET path = newpath` `WHERE equals_path(RES, path) > 0`	WebDAV: `MOVE;` FTP: `rename`	`DAV::unbind` on source parent folder `DAV::bind` on target parent folder `xdb:unlink-from` and `xdb:link-to` on resource	Yes

Copy resource	`INSERT INTO PATH_VIEW`
	`SELECT newpath, RES, link`
	`FROM PATH_VIEW`
	`WHERE equals_path(RES, oldpath)`
	`> 0`
		WebDAV: `COPY;`	▪ Copy to new:
			`DAV::bind` on target parent folder
			`DAV::read` on resource
			▪ Copy to existing (replacement):
			`DAV::read` on resource
			`DAV::write-properties` and `DAV::write-content` on existing target resource
				Yes

Create hard link to existing resource	`DBMS_XDB.link(`
	`srcpath IN VARCHAR2,`
	`linkfolder IN VARCHAR2,`
	`linkname IN VARCHAR2);`
		N/A	`DAV::bind` on parent folder
			`xdb:link-to` on resource
				No

Create weak link to existing resource	`DBMS_XDB.link(`
	`srcpath IN VARCHAR2,`
	`linkfolder IN VARCHAR2,`
	`linkname IN VARCHAR2,`
	`DBMS_XDB.LINK_TYPE_WEAK);`
		N/A	`DAV::bind` on parent folder
			`xdb:link-to` on resource
				No

Change owner of resource	`UPDATE RESOURCE_VIEW`
	`SET RES =`
	`updateXML(`
	`RES,`
	`'/Resource/Owner/text()',`
	`'new_owner')`
	`WHERE equals_path(RES, path) > 0`
		N/A	`DAV::take-ownership` on resource	Yes

Get binary or text representation of resource contents	`SELECT XDBURIType(path).getBLOB()`
	`FROM DUAL;`
	
	`SELECT`
	`XMLQuery(`
	`'declare default element namespace`
	`"http://xmlns.oracle.com/xdb/XDBRes`
	`ource.xsd"; (: :)`
	`/Resource/Contents'`
	`PASSING RES RETURNING CONTENT)`
	`FROM RESOURCE_VIEW p`
	`WHERE equals_path(RES, path) > 0`
		HTTP: `GET;`
		FTP: `get`
			`xdb:read-contents` on resource	Yes

Table 21–3 (Cont.) Accessing Oracle XML DB Repository: API Options

Data Access	SQL and PL/SQL	Protocols	Resource Privileges Required	JCR Support
Get XMLType representation of resource contents	`SELECT XDBURIType(path).getBLOB().getXML FROM DUAL;` `SELECT XMLQuery( 'declare default element namespace "http://xmlns.oracle.com/xdb/XDBResource.xsd"; (: :) /Resource/Contents/*' PASSING RES RETURNING CONTENT) FROM RESOURCE_VIEW p WHERE equals_path(RES, path) > 0`	not applicable	`xdb:read-contents` on resource	No
Get resource properties	`SELECT XMLCast( XMLQuery( 'declare default element namespace "http://xmlns.oracle.com/xdb/XDBResource.xsd"; (: :) /Resource/XXX' PASSING RES RETURNING CONTENT) AS VARCHAR2(256)) FROM RESOURCE_VIEW WHERE equals_path(RES, path) > 0`	WebDAV: `PROPFIND (depth = 0);`	`xdb:read-properties` on resource	Yes
List resources in folder	`SELECT PATH FROM PATH_VIEW WHERE under_path(res, path, 1) > 0`	WebDAV: `PROPFIND (depth = 0);`	`xdb:read-contents` on folder	Yes
Create folder	`Call DBMS_XDB.createFolder(VARCHAR2)`	WebDAV: `MKCOL;` FTP: `mkdir`	`DAV::bind` on parent folder	Yes
Unlink folder	`DBMS_XDB.deleteResource()` `DELETE FROM PATH_VIEW WHERE equals_path(RES, path) > 0;`	HTTP: `DELETE;` FTP: `rmdir`	`DAV::unbind` on parent folder `xdb:unlink-from` on resource	Yes
Forcibly delete folder and all links to it	`DBMS_XDB.deleteResource(VARCHAR2);`	not applicable	`DAV::unbind` on all parent folders `xdb:unlink-from` on folder resource	Yes
Get resource with a row lock	`SELECT ... FROM RESOURCE_VIEW FOR UPDATE ...;`	not applicable	`xdb:read-properties` and `xdb:read-contents` on resource	No
Add WebDAV lock on resource	`DBMS_XDB.LockResource(path, true, true);`	WebDAV: `LOCK;` FTP: `quote lock`	`DAV::write-properties` on resource	No
Remove WebDAV lock	`BEGIN DBMS_XDB.GetLockToken(path, deltoken); DBMS_XDB.UnlockToken(path, deltoken); END;`	WebDAV: `UNLOCK;` FTP: `quote unlock`	`DAV::write-properties` and `DAV::unlock` on resource	No
Check out file resource	`DBMS_XDB_VERSION.checkOut`	not applicable	`DAV::write-properties` on resource	No
Check in file resource	`DBMS_XDB_VERSION.checkIn`	not applicable	`DAV::write-properties` on resource	No
Uncheck out file resource	`DBMS_XDB_VERSION.unCheckOut`	not applicable	`DAV::write-properties` on resource	No

Table 21–3 (Cont.) Accessing Oracle XML DB Repository: API Options

Data Access	SQL and PL/SQL	Protocols	Resource Privileges Required	JCR Support
Make file resource versioned	`DBMS_XDB_VERSION.makeVersioned`	not applicable	`DAV::write-properties` on resource	No
Remove an event handler	`DBMS_XEVENT.remove`	not applicable	`xdb:write-config` on resource or parent folder (depending on the context)	No
Commit changes	`COMMIT;`	Automatic commit after each request	not applicable	Yes
Rollback changes	`ROLLBACK;`	not applicable	not applicable	Yes

See Also:

- Chapter 25, "Accessing the Repository using RESOURCE_VIEW and PATH_VIEW"
- Chapter 26, "Accessing the Repository using PL/SQL"
- Chapter 28, "Accessing the Repository using Protocols"
- Chapter 31, "Using Oracle XML DB Content Connector"
- *Oracle Database PL/SQL Packages and Types Reference* for information about PL/SQL package `DBMS_XDB`
- *Oracle Database PL/SQL Packages and Types Reference* for information about PL/SQL package `DBMS_XDB_VERSION`
- *Oracle Database PL/SQL Packages and Types Reference* for information about PL/SQL package `DBMS_XEVENT`

22

Configuring Oracle XML DB Repository

This chapter describes how to configure Oracle XML DB Repository. It contains these topics:

- Resource Configuration Files Configure a Resource
- Configuring a Resource
- Common Configuration Parameters

This chapter describes general configuration that applies to all repository resources. It does not describe configuration parameters for specific uses of resources. In particular, it does not describe configuration parameters for handling events or managing XLink and XInclude processing.

See Also:

- "Configuring Repository Events" on page 30-8
- Configuring Resources for XLink and XInclude on page 23-9
- XDBResource.xsd: XML Schema for Oracle XML DB Resources on page A-1
- XDBResConfig.xsd: XML Schema for Resource Configuration on page A-9

Resource Configuration Files Configure a Resource

Resource configuration is a general mechanism that you can use for events, mime-type mappings, servlet parameters, XLink and XInclude processing, default ACL specifications, and more.

You configure a Oracle XML DB Repository resource for any purpose by associating it with a resource configuration file, which defines configurable parameters for the resource. A **resource configuration file** is an XML file that conforms to the XML schema `XDBResConfig.xsd`, which is accessible in Oracle XML DB Repository at path `/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/XDBResConfig.xsd`. This XML schema is defined by Oracle XML DB, and you cannot alter it.

A resource configuration file is itself a resource in Oracle XML DB Repository. You use PL/SQL procedure `DBMS_RESCONFIG.addResConfig` to map a resource to the file that configures it. A single resource configuration file can alternatively apply to all resources in the repository. In that case, you use PL/SQL procedure `DBMS_RESCONFIG.addRepositoryResConfig` to map it to the repository as a whole.

The same resource configuration file can be used to configure more than one resource, if appropriate. Oracle recommends that you have resources share a configuration file

this way whenever the same configuration makes sense. This can improve run-time performance. It also simplifies repository management by letting you update a configuration in a single place and have the change affect multiple resources.

Avoid creating multiple, equivalent resource configuration files, because that can impact performance negatively. If Oracle XML DB detects duplicate resource configuration files, it raises an error.

Typically, you configure a resource for use with a particular application. In order for a resource to be shared by multiple applications, it must be possible for different applications to configure it differently. You do this by creating multiple resource configuration files and mapping them to the same resource. Each resource is thus associated with a list of configurations, a **resource configuration list**. Configurations in a configuration list are processed in the list order.

The repository itself has a list of resource configuration files, for repository-wide configuration, which really means configuration of all resources in the repository. The same configuration file must not be used for both the repository itself and a specific resource. Otherwise, an error is raised. An error is also raised if the same resource configuration file appears more than once in any given resource configuration list.

Note: An error is raised if you try to create more than 125 resource configuration files for repository-wide configuration.

The resource configuration list of a new resource is based on the information in the `configuration` elements of all resource configuration files for the parent folder of the new resource. If there is no such information (no configuration file or no `defaultChildConfig` elements in the files), then the `configuration` elements of the repository resource configuration list are used. If that information is also missing, then the new resource has an empty resource configuration list.

You can view the configuration list for a particular resource by extracting element `/Resource/RCList` from column `RES` of view `RESOURCE_VIEW`, or by using PL/SQL procedure `DBMS_RESCONFIG.getResConfigPath`. You can view the configuration list for the repository as a whole by using PL/SQL procedure `DBMS_RESCONFIG.getRepositoryResConfigPath`. To modify the repository-wide configuration list, you must be granted role `XDBADMIN`.

See Also: "Configuration Element defaultChildConfig" on page 22-3

Configuring a Resource

Follow these steps to configure an Oracle XML DB Repository resource or the repository as a whole (all resources):

1. Create a resource configuration file that defines the configuration. This XML file must conform to XML schema `XDBResConfig.xsd`.
2. Add the resource configuration file to the repository as a resource in its own right: a configuration resource. You can use PL/SQL function `DBMS_XDB.createResource` to do this.
3. Map this configuration resource to the resources that it configures, or to the repository if it applies to all resources. Use PL/SQL procedure `DBMS_RESCONFIG.addResConfig` or `DBMS_RESCONFIG.appendResConfig` to map an individual resource. Use `DBMS_RESCONFIG.addRepositoryResConfig` to map the repository as a whole.

4. Commit.

> **Note:** Before performing any operation that uses a resource configuration file, you must perform a `COMMIT` operation. Until you do that, an ORA-22881 "dangling REF" error is raised whenever you use the configuration file.

PL/SQL package `DBMS_RESCONFIG` provides additional procedures to delete a configuration from a configuration list, obtain a list of paths to configurations in a configuration list, and more.

> **Note:** If you delete a resource configuration file that is referenced by another resource, a dangling `REF` error is raised whenever an attempt is made to access the configured resource.

See Also:

- Example 22–1 for an example of a simple resource configuration file
- "Configuring Repository Events" on page 30-8 for complete examples of configuring resources
- "XDBResConfig.xsd: XML Schema for Resource Configuration" on page A-9
- *Oracle Database PL/SQL Packages and Types Reference* for information about package `DBMS_RESCONFIG`

Common Configuration Parameters

This section describes commonly used configuration parameters, that is, elements in a configuration file. Parameters specific to particular types of configuration are described elsewhere.

Configuration Element ResConfig

The top-level element of a resource configuration file is `ResConfig`. Besides `namespace` and `schemaLocation` attributes, it can contain an optional `enable` attribute. Set the value of attribute `enable` to `false` to disable the resource configuration file, so that it has no effect on the resources mapped to it. This can be useful for debugging or disabling an application. The default value of `enable`, used if the attribute is not present, is `true`.

Configuration Element defaultChildConfig

This configuration element applies to folders only. It holds configuration information that you want to be applied to all child resources in the folder. Element `defaultChildConfig` has one or more `configuration` child elements, each of which defines a possible configuration for resources in the folder.

A `configuration` element has the following child elements:

- `pre-condition` (optional) – This element specifies a condition that must be met before the resource configuration identified by the `path` element (see next) can be

used as the default configuration. If element `pre-condition` is absent, then the resource configuration file targeted by `path` applies to all resources in the folder. That is, the precondition is treated as true.

A `pre-condition` element has an optional `existsNode` child element. An `existsNode` element has a required `XPath` child element and an optional `namespace` child element, both strings. These define an XPath 1.0 expression and a namespace, respectively, that are used to check the existence of a resource. If that resource exists, then the precondition is satisfied, so the resource configuration file identified by `path` is used as a default resource configuration file for all child resources in the folder. The first component of the `XPath` element must be `Resource`.

> **Note:** A complex XPath expression for element `XPath` can impact performance negatively.

If multiple `configuration` elements have true preconditions, then each of the resource configuration files identified by their associated `path` elements applies to all of the resources in the folder.

- `path` (required) – This element specifies an absolute repository path to a resource configuration file that is to be used as the default configuration for a new resource whenever the precondition specified by element `pre-condition` is satisfied.

Typically, the value of the `path` element is a path to the current resource configuration file, that is, the file that contains the `path` element. Example 22–1 illustrates this, assuming that the resource configuration file is located at path `/cm/app_rc.xml` in the repository. In this example, the precondition is that there be a `Resource` node whose content is of type `xml`. When that precondition is met, the resource configuration file in Example 22–1 applies to all resources in same folder as the configuration file (`/cm/app_rc.xml`).

Example 22–1 Resource Configuration File

```
<ResConfig xmlns="http://xmlns.oracle.com/xdb/XDBResConfig.xsd"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="http://xmlns.oracle.com/xdb/XDBResConfig.xsd
                               http://xmlns.oracle.com/xdb/XDBResConfig.xsd">
  <defaultChildConfig>
    <configuration>
      <pre-condition>
        <existsNode>
          <XPath>/Resource[ContentType="xml"]</XPath>
        </existsNode>
      </pre-condition>
      <path>/cm/app_rc.xml<path>
    </configuration>
  </defaultChildConfig>
</ResConfig>
```

Configuration Element applicationData

You use element `applicationData` to store application-specific data. An application typically passes this data to an event handler when the handler is run. You can use any XML content that you want inside element `applicationData`. An event handler uses PL/SQL function `DBMS_XEVENT.getApplicationData` or Java function

`oracle.xdb.XMLType.getApplicationData` to access the data in the `applicationData` of the resource configuration file for the event listener.

Example 22–2 shows an `applicationData` element for use with an Oracle Spatial application.

Example 22–2 applicationData Element

```
<applicationData>
  <spatial:data xmlns:spatial="http://oracle/cartridge/spatial.xsd">
    <spatial:xpos>5</spatial:xpos>
    <spatial:ypos>10</spatial:ypos>
  </spatial:data>
</applicationData>
```

See Also:

- *Oracle Database PL/SQL Packages and Types Reference* for information about PL/SQL function `DBMS_XEVENT.getApplicationData`
- *Oracle Database XML Java API Reference*, class `XDBRepositoryEvent` for information about Java function `oracle.xdb.XMLType.getApplicationData`
- Example 30–1, "Resource Configuration File for Java Event Listeners with Preconditions" on page 30-9 for an example of a resource configuration file for event listeners

23

Using XLink and XInclude with Oracle XML DB

This chapter describes how to use XLink and XInclude with resources in Oracle XML DB Repository. It contains these topics:

- Overview of XLink and XInclude
- XLink and XInclude Link Types
- XInclude: Compound Documents
- Using XLink with Oracle XML DB
- Using XInclude with Oracle XML DB
- Examining XLink and XInclude Links using DOCUMENT_LINKS View
- Configuring Resources for XLink and XInclude
- Managing XLink and XInclude Links using DBMS_XDB.processLinks

Overview of XLink and XInclude

A document-oriented, or **content-management**, application often tracks relationships, between documents, and those relationships are often represented and manipulated as links of various kinds. Such links can affect application behavior in various ways, including affecting the document content and the response to user operations such as mouse clicks.

W3C has two recommendations that are pertinent in this context, for documents that are managed in XML repositories:

- **XLink –** Defines various types of links between resources. These links can model arbitrary relationships between documents. Those documents can reside inside or outside the repository.
- **XInclude –** Defines ways to include the content of multiple XML documents or fragments in a single infoset. This provides for compound documents, which model inclusion relationships. **Compound documents** are documents that contain other documents. More precisely, they are file resources that include documents or document fragments. The included objects can be file resources in the same repository or documents or fragments outside the repository.

Each of these standards is very general, and it is not limited to modeling relationships between XML documents. There is no requirement that the documents linked using XLink or included in an XML document using XInclude be XML documents.

Using XLink and XInclude to represent document relationships provides flexibility for applications, facilitates reuse of component documents, and enables their fine-grained manipulation (access control, versioning, metadata, and so on). Whereas using XML data structure (an ancestor–descendents hierarchy) to model relationships requires those relationships to be relatively fixed, using XLink and XInclude to model relationships can easily allow for change in those relationships.

Note: For XML schema-based documents to be able to use XLink and XInclude attributes, the XML schema must either explicitly declare those attributes or allow any attributes.

See Also:

- `http://www.w3.org/TR/xlink` for information about the XLink standard
- `http://www.w3.org/TR/xinclude` for information about the XInclude standard

XLink and XInclude Link Types

This section describes XLink and XInclude link types and the relation between these and Oracle XML DB Repository links. XLink links are more general than repository links. XLink links can be simple or extended. Oracle XML DB supports only simple XLink links, not extended links.

XLink and XInclude Links Model Document Relationships

XLink and XInclude links model arbitrary relationships among documents. The meaning and behavior of a relationship are determined by the applications that use the link. They are not inherent in the link itself. XLink and XInclude links can be mapped to Oracle XML DB **document links**. When document links target Oracle XML DB Repository resources, they can (according to a configuration option) be hard or weak links. In this, they are similar to repository links in that context. Repository links can be navigated using file system-related protocols such as FTP and HTTP. Document links cannot, but they can be navigated using the XPath 2.0 function `fn:doc`.

See Also: "Hard Links and Weak Links" on page 21-7

XLink and XInclude Link Types

XLink and XInclude can provide links to other documents. In the case of XInclude, attributes `href` and `xpointer` are used to specify the target document.

Xlink links can be simple or extended. **Simple** links are unidirectional, from a source to a target. **Extended** links (sometimes called **complex**) can model relationships between multiple documents, with different directionalities. Both simple and extended links can include link metadata. XLink links are represented in XML data using various attributes of the namespace `http://www.w3.org/1999/xlink`, which has the predefined prefix `xlink`. Simple links are represented in XML data using attribute `type` with value `simple`, that is, `xlink:type = "simple"`. Extended Xlink links are represented using `xlink:type = "extended"`.

Third-party extended Xlink links are not contained in any of the documents whose relationships they model. Third-party links can thus be used to relate documents, such as binary files, that, themselves, have no way of representing a link.

The source end of a simple Xlink link (that is, the document containing the link) must be an XML document. The target end of a simple link can be any document. There are no such restrictions for extended links. Example 23–3 shows examples of simple links. The link targets are represented using attribute `xlink:href`.

XInclude: Compound Documents

XInclude is the W3C recommendation for the syntax and processing model for merging the infosets of multiple XML documents into a single infoset. Element `xi:include` is used to include another document, specifying its URI as the value of an `href` attribute. Element `xi:include` can be nested, so that an included document can itself include other documents.

(However, an inclusion cycle raises an error in Oracle XML DB. The resources are created, but an error is raised when the inclusions are expanded.)

XInclude thus provides for *compound* documents: repository file resources that include other XML documents or fragments. The included objects can be file resources in the same repository or documents or fragments outside the repository.

A book might be an example of a typical compound document, as managed by a content-management system. Each book includes chapter documents, which can each be managed as separate objects, with their own URLs. A chapter document can have its own metadata and access control, and it can be versioned. A book can include (reference) a specific version of a chapter document. The same chapter document can be included in multiple book documents, for reuse. Because inclusion is modeled using XInclude, content management is simplified. It is easy, for example, to replace one chapter in a book by another.

Example 23–1 illustrates an XML `Book` element that includes four documents. One of those documents, `part1.xml`, is also shown. Document `part1.xml` includes other documents, representing chapters.

Example 23–1 XInclude Used in a Book Document to Include Parts and Chapters

The top-level document representing a book contains element `Book`.

```
<Book xmlns:xi="http://www.w3.org/2001/XInclude">
  <xi:include href=toc.xml"/>
  <xi:include href=part1.xml"/>
  <xi:include href=part2.xml"/>
  <xi:include href=index.xml"/>
</Book>
```

A major book part, file (resource) `part2.xml`, contains a `Part` element, which includes multiple chapter documents.

```
<?xml version="1.0"?>
<Part xmlns:xi="http://www.w3.org/2001/XInclude">
  <xi:include href="chapter5.xml"/>
  <xi:include href="chapter6.xml"/>
  <xi:include href="chapter8.xml"/>
  <xi:include href="chapter9.xml"/>
</Part>
```

These are some additional features of XInclude:

- Inclusion of plain text – You can include unparsed, non-XML text using attribute `parse` with a value of `text`: `parse = "text"`.

- Inclusion of XML fragments – You can use an `xpointer` attribute in an `xi:include` element to specify an XML fragment to include, instead of an entire document.
- Fallback processing – In case of error, such as inability to access the URI of an included document, an `xi:include` syntax error, or an `xpointer` reference that returns null, XInclude performs the treatment specified by element `xi:fallback`. This generally specifies an alternative element to be included. The alternative element can itself use `xi:include` to include other documents.

Using XLink with Oracle XML DB

Oracle XML DB supports only simple XLink links, not extended XLink links.

When an XML document containing XLink attributes is added to Oracle XML DB Repository, either as resource content or as user-defined resource metadata, special processing can occur, depending on how the repository or individual repository resources are configured. Element `XLinkConfig` of the resource configuration document, `XDBResConfig.xsd`, determines this behavior. In particular, you can configure resources so that XLink links are ignored, or so that they are mapped to Oracle XML DB document links. In the latter case, configuration can specify that the document links are to be hard or weak. Hard and weak document links have the same properties as hard and weak repository links.

The privileges needed to create or update document links are the same as those needed to create or update repository links. Even partially updating a document requires the same privileges needed to delete the entire document and reinsert it. In particular, even if you update just one document link you must have delete and insert privileges for *each* of the documents linked by the document containing the link.

If configuration maps XLink links to document links, then, whenever a document containing XLink links is added to the repository, the XLink information is extracted and stored in a system link table. Link target (destination) locations are replaced by direct paths that are based on the resource OIDs. Configuration can also specify whether OID paths are to be replaced by named paths (URLs) upon document retrieval. Using OID paths instead of named paths generally offers a performance advantage when links are processed, including when resource contents are retrieved.

You can use XLink within resource content, but not within resource metadata.

See Also:

- "Examining XLink and XInclude Links using DOCUMENT_LINKS View" on page 23-7
- Chapter 29, "User-Defined Repository Metadata"
- "Hard Links and Weak Links" on page 21-7
- "Configuring Resources for XLink and XInclude" on page 23-9
- "XDBResConfig.xsd: XML Schema for Resource Configuration" on page A-9

Using XInclude with Oracle XML DB

Oracle XML DB supports XInclude 1.0 as the standard mechanism for managing compound documents. It does not support attribute `xpointer` and the inclusion of document fragments, however. Only complete documents can be included (using attribute `href`).

You can use XInclude to create XML documents that include existing content. You can also configure the implicit decomposition of non-schema-based XML documents, creating a set of repository resources that contain XInclude inclusion references.

The content of included documents must be XML data or plain text (with attribute `parse = "text"`). You cannot include binary content directly using XInclude, but you can use XLink to link to binary content.

You can use XInclude within resource content, but not within resource metadata.

> **See Also:** "Examining XLink and XInclude Links using DOCUMENT_LINKS View" on page 23-7

Expanding Compound-Document Inclusions

When you retrieve a compound document from Oracle XML DB Repository, you have a choice:

- Retrieve it as is, with the `xi:include` elements remaining as such. This is the default behavior.
- Retrieve it after replacing the `xi:include` elements with their targets, recursively, that is, after expansion of all inclusions. An error is raised if any `xi:include` element cannot be resolved.

To retrieve the document in expanded form, use PL/SQL constructor `XDBURIType`, passing a value of `'1'` or `'3'` as the second argument (flags). Example 23–2 illustrates this. These are the possible values for the `XDBURIType` constructor second argument:

- `1` – Expand all XInclude inclusions before returning the result. If any such inclusion cannot be resolved according to the XInclude standard fallback semantics, then raise an error.
- `2` – Suppress all errors that might occur during document retrieval. This includes dangling `href` pointers.
- `3` – Same as `1` and `2` together.

Example 23–2 retrieves all documents that are under repository folder `public/bookdir`, expanding each inclusion:

Example 23–2 Expanding Document Inclusions using XDBURIType

```
SELECT XDBURIType(ANY_PATH, '1').getXML() FROM RESOURCE_VIEW
  WHERE under_path(RES, '/public/bookdir') = 1;

XDBURITYPE(ANY_PATH,'1').GETXML()
---------------------------------
<Book>
  <Title>A book</Title>
  <Chapter id="1">
    <Title>Introduction</Title>
    <Body>
      <Para>blah blah</Para>
      <Para>foo bar</Para>
    </Body>
  </Chapter>
  <Chapter id="2">
    <Title>Conclusion</Title>
    <Body>
      <Para>xyz xyz</Para>
      <Para>abc abc</Para>
```

```
    </Body>
  </Chapter>
</Book>

<Chapter id="1">
  <Title>Introduction</Title>
  <Body>
    <Para>blah blah</Para>
    <Para>foo bar</Para>
  </Body>
</Chapter>

<Chapter id="2">
  <Title>Conclusion</Title>
  <Body>
    <Para>xyz xyz</Para>
    <Para>abc abc</Para>
  </Body>
</Chapter>

3 rows selected.
```

(The result shown here corresponds to the resource `bookfile.xml` shown in Example 23–8, together with its included resources, `chap1.xml` and `chap2.xml`.)

See Also:

- "Versioning, Locking, and Controlling Access to Compound Documents" on page 23-6 for information about access control during expansion
- *Oracle Database PL/SQL Packages and Types Reference* for more information about `XDBURIType`

Validating Compound Documents

You validate a compound document the way you would any XML document. However, you can choose to validate it in either form: with `xi:include` elements as is or after replacing them with their targets.

You can also choose to use one XML schema to validate the unexpanded form, and another to validate the expanded form. For example, you might use one XML schema to validate without first expanding, in order to set up storage structures, and then use another XML schema to validate the expanded document after it is stored.

Updating Compound Documents

You can update a compound document just as you would update any resource. This replaces the resource with a new value. It thus corresponds to a resource deletion followed by a resource insertion. This means, in particular, that any `xi:include` elements in the original resource are deleted. Any `xi:include` elements in the replacement (inserted) document are processed as usual, according to the configuration defined at the time of insertion.

Versioning, Locking, and Controlling Access to Compound Documents

The components of a compound document are separate resources. They are versioned and locked independently, and their access is controlled independently.

- Document links to version-controlled resources (VCRs) always resolve to the latest version of the target resource, or the selected version within the current workspace. You can, however, explicitly refer to any specific version, by identifying the target resource by its OID-based path.
- Locking a document that contains `xi:include` elements does not also lock the included documents. Locking an included document does not also lock documents that include it.
- The access control list (ACL) on each referenced document is checked whenever you retrieve a compound document with expansion. This is done using the privileges of the current user (invoker's rights). If privileges are insufficient for any of the included documents, the expansion is canceled and an error is raised.

See Also:

- "Expanding Compound-Document Inclusions" on page 23-5
- Chapter 24, "Managing Resource Versions" for information about VCRs
- Chapter 27, "Repository Access Control" for information about resource ACLs

Examining XLink and XInclude Links using DOCUMENT_LINKS View

You can query the read-only public view `DOCUMENT_LINKS` to obtain system information about document links derived from both XLink and XInclude links. The information in this view includes the following columns, for each link:

- `SOURCE_ID` – The source resource OID. `RAW(16)`.
- `TARGET_ID` – The target resource OID. `RAW(16)`.
- `TARGET_PATH` – Always `NULL`. Reserved for future use. `VARCHAR2(4000)`.
- `LINK_TYPE` – The document link type: `Hard` or `Weak`. `VARCHAR2(8)`.
- `LINK_FORM` – Whether the original link was of form `XLink` or `XInclude`. `VARCHAR2(8)`.
- `SOURCE_TYPE` – Always `Resource Content`. `VARCHAR2(17)`.

You can obtain information about a resource from this view only if one of the following conditions holds:

- The resource is a link source, and you have the privilege `read-contents` or `read-properties` on it.
- The resource is a link target, and you have the privilege `read-properties` on it.

See Also: *Oracle Database Reference* for more information on public view `DOCUMENT_LINKS`

Querying DOCUMENT_LINKS for XLink Information

Example 23–3 shows how XLink links are treated when resources are created, and how to obtain system information about document links from view `DOCUMENT_LINKS`. It assumes that the folder containing the resource has been configured to map XLink links to document hard links.

Example 23–3 Querying Document Links Mapped From XLink Links

```
DECLARE
  b BOOLEAN;
BEGIN
  b := DBMS_XDB.createResource(
         '/public/hardlinkdir/po101.xml',
         '<PurchaseOrder id="101" xmlns:xlink="http://www.w3.org/1999/xlink">
            <Company xlink:type="simple"
                     xlink:href="/public/hardlinkdir/oracle.xml">Oracle Corporation</Company>
            <Approver xlink:type="simple"
                      xlink:href="/public/hardlinkdir/quine.xml">Willard Quine</Approver>
          </PurchaseOrder>');

  b := DBMS_XDB.createResource(
         '/public/hardlinkdir/po102.xml',
         '<PurchaseOrder id="102" xmlns:xlink="http://www.w3.org/1999/xlink">
            <Company xlink:type="simple"
                     xlink:href="/public/hardlinkdir/oracle.xml">Oracle Corporation</Company>
            <Approver xlink:type="simple"
                      xlink:href="/public/hardlinkdir/curry.xml">Haskell Curry</Approver>
            <ReferencePO xlink:type="simple"
                         xlink:href="/public/hardlinkdir/po101.xml"/>
          </PurchaseOrder>');
END;
/

SELECT r1.ANY_PATH source, r2.ANY_PATH target, dl.LINK_TYPE, dl.LINK_FORM
  FROM DOCUMENT_LINKS dl, RESOURCE_VIEW r1, RESOURCE_VIEW r2
  WHERE dl.SOURCE_ID = r1.RESID and dl.TARGET_ID = r2.RESID;

SOURCE                        TARGET                        LINK_TYPE LINK_FORM
----------------------------- ----------------------------- --------- ---------
/public/hardlinkdir/po101.xml /public/hardlinkdir/oracle.xml Hard      XLink
/public/hardlinkdir/po101.xml /public/hardlinkdir/quine.xml  Hard      XLink
/public/hardlinkdir/po102.xml /public/hardlinkdir/oracle.xml Hard      XLink
/public/hardlinkdir/po102.xml /public/hardlinkdir/curry.xml  Hard      XLink
/public/hardlinkdir/po102.xml /public/hardlinkdir/po101.xml  Hard      XLink
```

See Also: "Mapping XInclude Links to Hard Document Links, with OID Retrieval" on page 23-12 for an example of configuring a folder to map XLink links to hard links

Querying DOCUMENT_LINKS for XInclude Information

Example 23–4 queries view `DOCUMENT_LINKS` to show all document links.

Example 23–4 Querying Document Links Mapped From XInclude Links

```
DECLARE
  ret BOOLEAN;
BEGIN
  ret := DBMS_XDB.createResource(
           '/public/hardlinkdir/book.xml',
           '<Book xmlns:xi="http://www.w3.org/2001/XInclude">
              <xi:include href="/public/hardlinkdir/toc.xml"/>
              <xi:include href="/public/hardlinkdir/part1.xml"/>
              <xi:include href="/public/hardlinkdir/part2.xml"/>
              <xi:include href="/public/hardlinkdir/index.xml"/>
            </Book>');
END;

SELECT r1.ANY_PATH source, r2.ANY_PATH target, dl.LINK_TYPE, dl.LINK_FORM
```

```
  FROM DOCUMENT_LINKS dl, RESOURCE_VIEW r1, RESOURCE_VIEW r2
  WHERE dl.SOURCE_ID = r1.RESID and dl.TARGET_ID = r2.RESID;

SOURCE                       TARGET                       LINK_TYPE LINK_FORM
------                       ------                       --------- ---------
/public/hardlinkdir/book.xml /public/hardlinkdir/toc.xml   Hard      XInclude
/public/hardlinkdir/book.xml /public/hardlinkdir/part1.xml Hard      XInclude
/public/hardlinkdir/book.xml /public/hardlinkdir/part2.xml Hard      XInclude
/public/hardlinkdir/book.xml /public/hardlinkdir/index.xml Hard      XInclude
```

Configuring Resources for XLink and XInclude

You configure XLink and XInclude treatment for Oracle XML DB Repository resources as you would configure any other treatment of repository resources—see "Configuring a Resource" on page 22-2. The rest of this section describes the resource configuration file that you use as a resource to configure XLink and XInclude processing for other resources.

A resource configuration file is an XML file that conforms to the XML schema `XDBResConfig.xsd`, which is accessible in Oracle XML DB Repository at path `/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/XDBResConfig.xsd`. You use elements `XLinkConfig` and `XIncludeConfig`, children of element `ResConfig`, to configure XLink and XInclude treatment, respectively. If one of these elements is absent, then there is no treatment of the corresponding type of links.

Both `XLinkConfig` and `XIncludeConfig` can have attribute `UnresolvedLink` and child elements `LinkType` and `PathFormat`. Element `XIncludeConfig` can also have child element `ConflictRule`. If the `LinkType` element content is `None`, however, then there must be no `PathFormat` or `ConflictRule` element.

You cannot define any preconditions for `XLinkConfig` or `XIncludeConfig`. During repository resource creation, the `ResConfig` element of the parent folder determines the treatment of XLink and XInclude links for the new resource. If the parent folder has no `ResConfig` element, then the repository-wide configuration applies.

Any change to the resource configuration file applies only to documents that are created or updated after the configuration-file change. To process links in existing documents, use PL/SQL procedure `DBMS_XDB.processLinks`, after specifying the appropriate resource configuration parameters.

See Also:

- "Managing XLink and XInclude Links using DBMS_XDB.processLinks" on page 23-13
- Chapter 22, "Configuring Oracle XML DB Repository"

Configuring Treatment of Unresolved Links: UnresolvedLink Attribute

A `LinkConfig` element can have an `UnresolvedLink` attribute with a value of `Error` (default value) or `Skip`. This determines what happens if an XLink or XInclude link cannot be resolved at the time of document insertion into the repository (resource creation). `Error` means raise an error and roll back the current operation. `Skip` means skip any treatment of the XLink or XInclude link. Skipping treatment creates the resource with no corresponding document links, and sets the resource's `HasUnresolvedLinks` attribute to `true`, to indicate that the resource has unresolved links.

Using `Skip` as the value of attribute `UnresolvedLink` can be especially useful when you create a resource that contains a cycle of weak links, which would otherwise lead to unresolved-link errors during resource creation. After the resource and all of its linked resources have been created, you can use PL/SQL procedure `DBMS_XDB.processLinks` to process the skipped links. If all XLink and XInclude links have been resolved by this procedure, then attribute `HasUnresolvedLinks` is set to `false`.

Resource attribute `HasUnresolvedLinks` is also set to `true` for a resource that has a weak link to a resource that has been deleted. Deleting a resource thus effectively also deletes any weak links pointing to that resource. In particular, whenever the last hard link to a resource is deleted, the resource is itself deleted, and all resources that point to the deleted resource with a weak link have attribute `HasUnresolvedLinks` set to `true`.

See Also:

- "Hard Links and Weak Links" on page 21-7
- "Managing XLink and XInclude Links using DBMS_XDB.processLinks" on page 23-13

Configuring the Document Links to Create: LinkType Element

You use the `LinkType` element of a resource configuration file to specify the type of document link to be created whenever an XLink or XInclude link is encountered when a document is stored in Oracle XML DB Repository. The `LinkType` element has these possible values (element content):

- `None` (default) – Ignore XLink or XInclude links: create no corresponding document links.
- `Hard` – Map XLink or XInclude links to hard document links in repository documents.
- `Weak` – Map XLink or XInclude links to weak document links in repository documents.

See Also:

- Example 23–5
- Example 23–6

Configuring the Path Format for Retrieval: PathFormat Element

You use the `PathFormat` element of a resource configuration file to specify the path format to be used when retrieving documents with `xlink:href` or `xi:include:href` attributes. The `PathFormat` element has these possible values (element content) for hard and weak document links:

- `OID` (default) – Map XLink or XInclude `href` paths to OID-based paths in repository documents—that is, use OIDs directly.
- `Named` – Map XLink or XInclude `href` paths to named paths (URLs) in repository documents. The path is computed from the internal OID when the document is retrieved, so retrieval can be slower than in the case of using OID paths directly.

See Also:

- Example 23–5
- Example 23–6

Configuring Conflict-Resolution for XInclude: ConflictRule Element

You use the `ConflictRule` element of a resource configuration file to specify the conflict-resolution rules to use if the path computed for a component document is already present in Oracle XML DB Repository. The `ConflictRule` element has these possible values (element content):

- `Error` (default) – Raise an error.
- `Overwrite` – Update the document targeted by the existing repository path, replacing it with the document to be included. If the existing document is a version-controlled resource, then it must already be checked out, unless it is autoversioned. Otherwise, an error is raised.
- `Syspath` – Change the path to the included document to a new, system-defined path.

See Also: Chapter 24, "Managing Resource Versions" for information about version-controlled resources

Configuring Decomposition of Documents using XInclude: SectionConfig Element

You use the `SectionConfig` element of a resource configuration file to specify how non-schema-based XML documents are to be decomposed when added to Oracle XML DB Repository, to create a set of resources that contain XInclude inclusion references. You use simple XPath expressions in the resource configuration file to identify which parts of a document to map to separate resources, and which resources to map them to.

Element `SectionConfig` contains one or more `Section` elements, each of which contains the following child elements:

- `sectionPath` – Simple XPath 1.0 expression that identifies a section root. This must use only child and descendant axes, and it must not use wildcards.
- `documentPath` (optional) – Simple XPath 1.0 expression that is evaluated to identify the resources to be created from decomposing the document according to `sectionPath`. The XPath expression must use only child, descendant, and attribute axes.
- `namespace` (optional) – Namespace in effect for `sectionPath` and `documentPath`.

Element `Section` also has a `type` attribute that specifies the type of section to be created. Value `Document` means create a document. The default value, `None`, means do not create anything. Using `None` is equivalent to removing the `SectionConfig` element. You can thus set the `type` attribute to `None` to disable a `SectionConfig` element temporarily, without removing it, and then set it back to `Document` to enable it again.

If an element in the document being added to the repository matches more than one `sectionPath` value, only the first such expression (in document order) is used.

If no `documentPath` element is present, then the resource created has a system-defined name, and is put into the folder specified for the original document.

See Also:

- Example 23–7, "Configuring XInclude Document Decomposition" on page 23-12
- Example 23–8, "Repository Document, Showing Generated xi:include Elements" on page 23-13

XLink and XInclude Configuration Examples

Example 23–5 shows a configuration-file section that configures XInclude treatment, mapping XInclude attributes to Oracle XML DB Repository hard document links. Repository paths in retrieved resources are configured to be based on resource OIDs.

Example 23–5 Mapping XInclude Links to Hard Document Links, with OID Retrieval

```
<ResConfig>
  . . .
  <XIncludeConfig UnresolvedLink="Skip">
    <LinkType>Hard</LinkType>
    <PathFormat>OID</PathFormat>
  </XIncludeConfig>
  . . .
</ResConfig>
```

Example 23–6 shows an `XLinkConfig` section that maps XLink links to weak document links in the repository. In this case, retrieval of a document uses named paths (URLs).

Example 23–6 Mapping XLInk Links to Weak Links, with Named-Path Retrieval

```
<ResConfig>
  . . .
  <XLinkConfig UnresolvedLink="Skip">
    <LinkType>Weak</LinkType>
    <PathFormat>Named</PathFormat>
  </XLinkConfig>
  . . .
</ResConfig>
```

Example 23–7 shows a `SectionConfig` section that specifies that each `Chapter` element in an input document is to become a separate repository file, when the input document is added to Oracle XML DB Repository. The repository path for the resulting file is specified using configuration element `documentPath`, and this path is relative to the location of the resource configuration file of Example 23–6.

Example 23–7 Configuring XInclude Document Decomposition

```
<ResConfig>
  . . .
  <SectionConfig>
    <Section type = "Document">
      <sectionPath>//Chapter</sectionPath>
      <documentPath>concat("chap", @id, ".xml")</documentPath>
    </Section>
  </SectionConfig>
  . . .
</ResConfig>
```

The XPath expression here uses XPath function `concat` to concatenate the following strings to produce the resulting repository path to use:

- `chap` – (prefix) `chap`.
- The value of attribute `id` of element `Chapter` in the input document.
- `.xml` as a file extension.

For example, a repository path of `chap27.xml` would result from an input document with a `Chapter` element that has an `id` attribute with value `27`:

```
<Chapter id="27"> ... </Chapter>
```

If the configuration document of Example 23–6 and the book document that contains the `XInclude` elements are in repository folder `/public/bookdir`, then the individual chapter files generated from `XInclude` decomposition are in files `/public/bookdir/chap`*N*`.xml`, where the values of *N* are the values of the `id` attributes of `Chapter` elements.

The book document that is added to the repository is derived from the input book document. The embedded `Chapter` elements in the input book document are replaced by `xi:include` elements that reference the generated chapter documents—Example 23–8 illustrates this.

Example 23–8 Repository Document, Showing Generated xi:include Elements

```
SELECT XDBURIType('/public/bookdir/bookfile.xml').getclob() FROM DUAL;

XDBURITYPE('/PUBLIC/BOOKDIR/BOOKFILE.XML').GETCLOB()
--------------------------------------------------------------------------------
<Book>
  <Title>A book</Title>
  <xi:include xmlns:xi="http://www.w3.org/2001/XInclude" href="/public/bookdir/chap1.xml"/>
  <xi:include xmlns:xi="http://www.w3.org/2001/XInclude" href="/public/bookdir/chap2.xml"/>
</Book>
```

See Also:

- Chapter 22, "Configuring Oracle XML DB Repository"
- XDBResConfig.xsd: XML Schema for Resource Configuration
- "Configuring Decomposition of Documents using XInclude: SectionConfig Element" on page 23-11

Managing XLink and XInclude Links using DBMS_XDB.processLinks

You can use PL/SQL procedure `DBMS_XDB.processLinks` to manually process all XLink and XInclude links in a single document or in all documents of a folder. Pass `RECURSIVE` as the mode argument to this procedure, if you want to process all hard-linked subfolders recursively. All XLink and XInclude links are processed according to the corresponding configuration parameters. If any of the links within a resource cannot be resolved, the resource's `HasUnresolvedLinks` attribute is set to `true`, to indicate that the resource has unresolved links. The default value of attribute `HasUnresolvedLinks` is `false`.

See Also: "Configuring Treatment of Unresolved Links: UnresolvedLink Attribute" on page 23-9

24

Managing Resource Versions

This chapter describes how to create and manage versions of Oracle XML DB resources.

This chapter contains these topics:

- Overview of Oracle XML DB Versioning
- Versioning and Resource IDs
- Versioning and ACLs
- Resource Versioning Examples

Overview of Oracle XML DB Versioning

Versioning lets you create and manage different versions of a resource in Oracle XML DB Repository. A record, or history, is kept of all changes to an Oracle XML DB resource that is under version control. When you update a version-controlled resource, Oracle XML DB stores the pre-update contents as a separate resource version – a snapshot for the historical record.

Versioning features include the following:

- Version control for a resource.

  You can turn version control on or off for an Oracle XML DB Repository resource.

- Updating a version-controlled resource.

  When Oracle XML DB updates a version-controlled resource, it creates a new version of the resource. This new version is not deleted from the database when you delete the version-controlled resource.

- Accessing a version-controlled resource.

  You can access a version-controlled resource the same way you access any other resource.

- Accessing a resource version.

  To access a particular version of a resource, you use the resource ID of that version. The resource ID can be obtained from the resource version history or from the version-controlled resource itself. See "Versioning and Resource IDs".

Table 24–1 lists some terms used in this chapter.

Table 24–1 Oracle XML DB Versioning Terms

Term	Description
Versionable resource	A resource that can be put under version control. All Oracle XML DB resources except folders and ACLs are versionable.
Version-controlled resource	A resource that is under version control.
Version resource	A particular version of a version-controlled resource. A version resource is itself a resource. It is system-generated, and it has no associated path name. It is read-only (it cannot be updated or deleted).
`checkOut`, `checkIn`, `unCheckOut`	Operations for managing version-controlled resources. You must use `checkOut` before you can modify a version-controlled resource. Use `checkIn` to make your changes permanent. Use `unCheckOut` to cancel your changes. (Use `COMMIT` after each of these operations.)

Note: Oracle XML DB supports version control only for Oracle XML DB resources. It does *not* support version control for user-defined tables or data in Oracle Database.

Oracle does *not* guarantee preservation of the resource ID of a version across check-in and check-out. Everything except the resource ID of the latest version is preserved.

Oracle XML DB supports versioning of XML resources that are not XML schema-based. It also supports versioning of XML schema-based resources and resources that contain XML schema-based metadata, but only if the underlying tables have *no* associated triggers or constraints.

If hierarchy is enabled for a table, then the table has a trigger. This includes tables that are created as part of XML schema registration, for which the default behavior is to enable hierarchy.

Be aware also that if you query one of the tables underlying a resource, the query can return data from multiple versions of the resource. This is because the data for the different resource versions is stored in the same underlying table, using different rows.

Versioning and Resource IDs

A resource object ID, or **resource ID**, is a unique, constant, system-generated identifier for an Oracle XML DB resource. Each resource has a resource ID. This includes *version resources*, which are system-generated resources that do not have any path names. A resource ID is sometimes called a **RESID**.

You use PL/SQL package `DBMS_XDB_VERSION` to put a resource under version-control and manage different versions of it. Some of the `DBMS_XDB_VERSION` routines accept the path name of a version-controlled resource as argument and return the resource ID of the relevant version resource.

For example, you use function `DBMS_XDB_VERSION.makeVersioned` to put a resource under version control, that is, to turn it into a version-controlled resource. It accepts as argument a repository path to the resource.

You need not use the same path name for a given version-controlled resource when you perform various versioning operations on it, but the path names you use must all refer to the same resource.

Whenever a path name is passed as an argument representing a version-controlled resource, it is the latest (that is, the current) version of the resource that is used. A *path name always stands for the latest version.* The only way you can refer to a version other than the current version is to use its resource ID.

The resource ID of a given version is constant. Remember that a version is itself a resource, and the resource ID of a resource never changes.

Each time you check in a version-controlled resource, Oracle XML DB creates a new version resource. A **version resource** is a snapshot of a resource (its content and metadata) together with a resource ID. The collection of version resources for a given version-controlled resource constitutes a historical sequence of previous versions, the **version series** or history of the resource.

When you check in a version-controlled resource that has resource ID ***R***, Oracle XML DB creates a new resource ID, ***P***, which refers to a snapshot of the resource (both content and metadata), as it was before it was last checked out. The snapshot was made before check-out, but the associated version resource (and its resource ID ***P***) are created at check-in time. Together, the new resource ID ***P*** and the snapshot it refers to thus represent the *previous*, not the current, version of the resource. Resource ID ***R*** continues to refer to the current version.

Put another way, when you check in a version-controlled resource, a version resource is created that represents the previous state of the version-controlled resource. Like any new resource, this new version resource is allocated a new resource ID (***P***).

You can think about making a version resource (check-in) the way you think about making a backup copy of a file: Just as you give a new name to the backup file, so the previous-version snapshot of a resource is given a new resource ID. The current resource retains the original resource ID, just as your working file keeps its original name.

What this means is that when you check in a resource, in order to "create a new version", what's really new is the version resource (resource ID ***P*** and the snapshot it references) that represents the *old* (previous) version. The newest, or latest, version of the resource (***R***) is really just the current version. Remember: new version resource = old (previous) version of the resource content and metadata.

Resource ID ***R*** refers to the *current* version of the version-controlled resource throughout its lifetime, from the moment it was put under version control until it is deleted. You can always access the latest version of a resource using its original resource ID.

When you need to refer to a previous version of a resource, you must use its resource ID to reference it. You cannot use a path name. You can use function `DBMS_XDB_VERSION.getPredsByRESID` to obtain the resource ID of the previous version of a given resource.

Note: If you *delete* a resource, then any subsequent reference to it, whether by resource ID or path name, raises an error (typically `ORA-31001: Invalid resource handle or path name`). You *cannot* access any version of a version-controlled resource that has been deleted.

Versioning and ACLs

A version resource is immutable. It is a snapshot of resource content and metadata, plus a resource ID, and both snapshot and ID are static. Likewise, the ACL of a version resource cannot be changed.

You can modify the ACL of a version-controlled resource that you have checked out. When you check it in, the modified ACL continues to be associated with the current (latest) version of the resource, and the previous version, that is, the newly created version resource, is associated with the ACL before it was modified. That is, the previous version is associated with the previous ACL, and the current version is associated with the updated ACL.

What is important to keep in mind is this:

- Different versions of a resource can have different ACLs associated with them.
- You can modify the ACL associated with the current version after you check out the resource.
- Check-in associates the ACL as it was before check-out with the newly created version resource, that is, with the previous version of the resource.
- The ACL associated with a given version remains the same.

Resource Versioning Examples

This section presents examples that do the following:

- Put a resource under version control, that is, create a version-controlled resource – Example 24–2
- Retrieve the content of the resource by referring to the resource ID – Example 24–3
- Check out the version-controlled resource (for all users) – Example 24–4
- Update the resource content – Example 24–5
- Check in the resource – Example 24–6
- Retrieve the content and metadata of both the new and old versions of the resource – Example 24–7, Example 24–8, Example 24–9
- Cancel a resource check-out – Example 24–10

Example 24–3 creates an Oracle XML DB Repository resource at repository path `/public/t1.txt`. The resource has as content the text `Mary had a little lamb`. The example uses SQL*Plus command `VARIABLE` to declare bind variables `targetPath`, `current_RESID`, and `previous_RESID`, which are used in other examples in this section.

Example 24–1 Creating a Repository Resource

```
VARIABLE targetPath      VARCHAR2(700)
VARIABLE current_RESID   VARCHAR2(32)
VARIABLE previous_RESID  VARCHAR2(32)

DECLARE
  res BOOLEAN;
BEGIN
  :targetPath  := '/public/t1.txt';
  IF (DBMS_XDB.existsResource(:targetPath))
     THEN DBMS_XDB.deleteResource(:targetPath);
```

```
  END IF;
  res := DBMS_XDB.createResource(:targetPath, 'Mary had a little lamb');
END;
/
```

The new resource is *not* version-controlled. Example 24–2 uses PL/SQL function `makeVersioned` to put it under version control. This function returns the resource ID of the first version resource for the version-controlled resource. The function does not auto-commit. You must explicitly use `COMMIT`.

Example 24–2 Creating a Version-Controlled Resource

```
DECLARE
  resid DBMS_XDB_VERSION.RESID_TYPE;
BEGIN
  resid := DBMS_XDB_VERSION.makeVersioned(:targetPath);
  :current_RESID := resid;
  COMMIT;
END;
/
```

Example 24–2 also copies the resource ID of the new version resource to bind variable `current_RESID`. Example 24–3 shows how to use type constructor `XDBUritype` together with PL/SQL function `createOIDPath` to retrieve the resource content by referencing the resource ID.

Example 24–3 Retrieving Resource Content by Referencing the Resource ID

```
SELECT XDBURIType(DBMS_XDB.createOIDPath(:current_RESID)).getClob() FROM DUAL;

XDBURITYPE(DBMS_XDB.CREATEOIDPATH(:CURRENT_RESID)).GETCLOB()
------------------------------------------------------------
Mary had a little lamb

1 row selected.
```

Example 24–4 checks out the version-controlled resource (and commits), so that it can be modified. Note that any user can modify a resource that has been checked out.

Example 24–4 Checking Out a Version-Controlled Resource

```
BEGIN
  DBMS_XDB_VERSION.checkOut(:targetPath);
  COMMIT;
END;
/
```

Example 24–5 updates the content of the checked-out resource. Before the (LOB) content can be updated, you must lock the resource. The example uses a dummy update of the resource display name (a scalar attribute) to do this.

Example 24–5 Updating Resource Content

```
DECLARE
  content        BLOB;
  newContentBlob BLOB;
  newContentClob CLOB;
  source_offset  INTEGER := 1;
  target_offset  INTEGER := 1;
  warning        INTEGER;
```

```
  lang_context   INTEGER := 0;
BEGIN
  -- Lock the resource using a dummy update.
  UPDATE RESOURCE_VIEW
    SET RES =
      updateXML(RES, '/Resource/DisplayName/text()',
                XMLCast(XMLQuery(
                          'declare namespace ns =
                           "http://xmlns.oracle.com/xdb/XDBResource.xsd"; (: :)
                           $r/ns:Resource/ns:DisplayName/text()'
                          PASSING RES AS "r" RETURNING CONTENT)
                        AS VARCHAR2(128)))
    WHERE equals_path(res, :targetPath) = 1;
  -- Get the LOB locator.
  SELECT XMLCast(XMLQuery('declare namespace ns =
                           "http://xmlns.oracle.com/xdb/XDBResource.xsd"; (: :)
                           $r/ns:Resource/ns:XMLLob'
                          PASSING RES AS "r" RETURNING CONTENT)
                 AS BLOB)
    INTO content FROM RESOURCE_VIEW
    WHERE equals_path(RES, :targetPath) = 1;
  -- Update the LOB.
  newContentClob := 'Hickory dickory dock, the mouse ran up the clock';
  DBMS_LOB.createTemporary(newContentBlob, false, DBMS_LOB.CALL);
  DBMS_LOB.convertToBlob(newContentBlob, newContentClob,
                         DBMS_LOB.getLength(newContentClob),
                         source_offset, target_offset,
                         nls_charset_id('AL32UTF8'), lang_context, warning);
  DBMS_LOB.open(content, DBMS_LOB.lob_readwrite);
  DBMS_LOB.trim(content, 0);
  DBMS_LOB.append(content, newContentBlob);
  DBMS_LOB.close(content);
  DBMS_LOB.freeTemporary(newContentBlob);
  DBMS_LOB.freeTemporary(newContentClob);
  COMMIT;
END;
/
```

Example 24–5 retrieves the LOB content using the LOB locator, which is element `/ns:Resource/ns:XMLLob`. It empties the existing content and adds new content using PL/SQL procedures `trim` and `append` in package `DBMS_LOB`. It commits the content change.

> **See Also:** *Oracle Database SecureFiles and Large Objects Developer's Guide* for information about updating a LOB

At this point, the content has been modified, but this change has not been recorded in the version series. Example 24–6 checks in the resource and commits the check-in.

Example 24–6 Checking In a Version-Controlled Resource

```
DECLARE
  resid DBMS_XDB_VERSION.RESID_TYPE;
BEGIN
  resid := DBMS_XDB_VERSION.checkIn(:targetPath);
  :previous_RESID := DBMS_XDB_VERSION.getPredsByRESID(resid)(1);
  COMMIT;
END;
/
```

PL/SQL function `checkIn` returns the resource ID of the current version, which is the same as `current_RESID`. Example 24–6 passes this value to PL/SQL function `getPredsByRESID`. This function returns the list of resource IDs for the (immediate) predecessors of its argument resource.[1] Example 24–6 assigns the first (and only) element of this list to bind variable `previous_RESID`.

At this point, the value of `current_RESID` is the resource ID of the current version, and the value of `previous_RESID` is the resource ID of the previous version.

You can retrieve the content or metadata of a resource using any of the following methods:

- `XDBURIType` together with PL/SQL function `DBMS_XDB.createOIDPath` – Retrieve content. See Example 24–3 and Example 24–7.
- PL/SQL function `DBMS_XDB_VERSION.getContentsCLOBByRESID` – Retrieve content. See Example 24–8.
- PL/SQL function `DBMS_XDB_VERSION.getResourceByRESID` – Retrieve metadata. See Example 24–9.

You can use `XDBURIType` with `createOIDPath` to access resource content using protocols. For example, you could have Oracle XML DB serve up various versions of a graphic image file resource for a Web page, setting the `HREF` for the HTML `IMAGE` tag to a value returned by `createOIDPath`.

Example 24–7 through Example 24–9 use these different methods to retrieve the two versions of the resource addressed by bind variables `current_RESID` and `previous_RESID` after check-in.

Example 24–7 Retrieving Resource Version Content using XDBURITYPE and CREATEOIDPATH

```
SELECT XDBURIType(DBMS_XDB.createOIDPath(:current_RESID)).getClob() FROM DUAL;

XDBURITYPE(DBMS_XDB.CREATEOIDPATH(:CURRENT_RESID)).GETCLOB()
------------------------------------------------------------
Mary had a little lamb

1 row selected.

SELECT XDBURIType(DBMS_XDB.createOIDPath(:previous_RESID)).getClob() FROM DUAL;

XDBURITYPE(DBMS_XDB.CREATEOIDPATH(:PREVIOUS_RESID)).GETCLOB()
-------------------------------------------------------------
Hickory dickory dock, the mouse ran up the clock

1 row selected.
```

Example 24–8 Retrieving Resource Version Content using GETCONTENTSCLOBBYRESID

```
SELECT DBMS_XDB_VERSION.getContentsCLOBByRESID(:current_RESID) FROM DUAL;

DBMS_XDB_VERSION.GETCONTENTSCLOBBYRESID(:CURRENT_RESID)
-------------------------------------------------------
Mary had a little lamb
```

[1] In Oracle XML DB, a version resource always has a *single* predecessor, that is, a single version that immediately precedes it. The WebDAV standard provides for the possibility of multiple predecessors.

```
1 row selected.

SELECT DBMS_XDB_VERSION.getContentsCLOBByRESID(:previous_RESID) FROM DUAL;

DBMS_XDB_VERSION.GETCONTENTSCLOBBYRESID(:PREVIOUS_RESID)
--------------------------------------------------------
Hickory dickory dock, the mouse ran up the clock

1 row selected.
```

Example 24–9 Retrieving Resource Version Metadata using GETRESOURCEBYRESID

```
SELECT XMLSerialize(DOCUMENT DBMS_XDB_VERSION.getResourceByRESID(:current_RESID)
                    AS CLOB INDENT SIZE = 2)
  FROM DUAL;
<Resource xmlns="http://xmlns.oracle.com/xdb/XDBResource.xsd" Hidden="false"
          Invalid="false" VersionID="2" ActivityID="0" Container="false"
          CustomRslv="false" VersionHistory="false" StickyRef="true">
  <CreationDate>2009-05-06T12:33:34.012133</CreationDate>
  <ModificationDate>2009-05-06T12:33:34.280199</ModificationDate>
  <DisplayName>t1.txt</DisplayName>
  <Language>en-US</Language>
  <CharacterSet>UTF-8</CharacterSet>
  <ContentType>text/plain</ContentType>
  <RefCount>1</RefCount>
  <ACL>
    <acl description="Public:All privileges to PUBLIC"
         xmlns="http://xmlns.oracle.com/xdb/acl.xsd"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.oracle.com/xdb/acl.xsd
                             http://xmlns.oracle.com/xdb/acl.xsd"
         shared="true">
      <ace>
        <grant>true</grant>
        <principal>PUBLIC</principal>
        <privilege>
          <all/>
        </privilege>
      </ace>
    </acl>
  </ACL>
  <Owner>HR</Owner>
  <Creator>HR</Creator>
  <LastModifier>HR</LastModifier>
  <SchemaElement>http://xmlns.oracle.com/xdb/XDBSchema.xsd#text</SchemaElement>
  <Contents>
    <text>Mary had a little lamb</text>
  </Contents>
  <VCRUID>69454F2EF12E3375E040578C8A1764B5</VCRUID>
  <Parents>69454F2EF12F3375E040578C8A1764B5</Parents>
</Resource>

1 row selected.

SELECT XMLSerialize(DOCUMENT DBMS_XDB_VERSION.getResourceByRESID(:previous_RESID)
                    AS CLOB INDENT SIZE = 2)
  FROM DUAL;
<Resource xmlns="http://xmlns.oracle.com/xdb/XDBResource.xsd" Hidden="false"
          Invalid="false" VersionID="1" Container="false" CustomRslv="false"
          VersionHistory="false" StickyRef="true">
```

```
  <CreationDate>2009-05-06T12:33:34.012133</CreationDate>
  <ModificationDate>2009-05-06T12:33:34.012133</ModificationDate>
  <DisplayName>t1.txt</DisplayName>
  <Language>en-US</Language>
  <CharacterSet>UTF-8</CharacterSet>
  <ContentType>text/plain</ContentType>
  <RefCount>0</RefCount>
  <ACL>
    <acl description="Public:All privileges to PUBLIC"
         xmlns="http://xmlns.oracle.com/xdb/acl.xsd"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.oracle.com/xdb/acl.xsd
                             http://xmlns.oracle.com/xdb/acl.xsd"
         shared="true">
      <ace>
        <grant>true</grant>
        <principal>PUBLIC</principal>
        <privilege>
          <all/>
        </privilege>
      </ace>
    </acl>
  </ACL>
  <Owner>HR</Owner>
  <Creator>HR</Creator>
  <LastModifier>HR</LastModifier>
  <SchemaElement>http://xmlns.oracle.com/xdb/XDBSchema.xsd#text</SchemaElement>
  <Contents>
    <text>Hickory dickory dock, the mouse ran up the clock</text>
  </Contents>
  <VCRUID>69454F2EF12E3375E040578C8A1764B5</VCRUID>
</Resource>

1 row selected.
```

You can cancel a check-out using PL/SQL function `DBMS_XDB_VERSION.unCheckOut`. Example 24–10 illustrates this.

Example 24–10 Canceling a Check-Out using UNCHECKOUT

```
DECLARE
  resid DBMS_XDB_VERSION.RESID_TYPE;
BEGIN
  resid := DBMS_XDB_VERSION.unCheckOut(:targetPath);
END;
/
```

Table 24–2 summarizes the subprograms in PL/SQL package `DBMS_XDB_VERSION` that are mentioned in this chapter.

Table 24–2 PL/SQL Functions and Procedures in Package DBMS_XDB_VERSION

Function or Procedure	Description
`makeVersioned(pathname VARCHAR2) RETURN DBMS_XDB_VERSION.RESID_TYPE;`	Turn a resource with the given path name into a version controlled resource. If two or more path names refer to the same resource, then the resource is copied, and argument path name is bound with the copy. The new resource is put under version control. All other path names continue to refer to the original resource. The argument is the path name of the resource to be put under version control. Returns the resource ID of the first version resource of the version-controlled resource. This is not an auto-commit SQL operation. An error is raised of you call `makeVersioned` for a folder, version resource, or ACL, or if the target resource does not exist. Note: No error or warning is raised if you call `makeVersioned` for a version-controlled resource.
`checkOut(pathname VARCHAR2);`	Check out a version-controlled resource. You cannot update or delete a version-controlled resource until you check it out. Check-out is for all users: any user can modify a resource that has been checked out. The argument is the path name of the version-controlled resource to be checked out. This is not an auto-commit SQL operation. If two users check out the same version-controlled resource at the same time, then one user must roll back. As a precaution, commit after checking out and before updating a resource. An error is raised if the target resource is not under version control, does not exist, or is already checked out.
`checkIn (pathname VARCHAR2) RETURN DBMS_XDB_VERSION.RESID_TYPE;`	Check in a version-controlled resource that has been checked out. `pathname` - Path name of the checked-out resource. Returns the resource id of the newly created version. This is not an auto-commit SQL operation. You need not use the same path name that was used for check-out. However, the check-in path name and the check-out path name must reference the same resource, or else results are unpredictable. If the resource has been renamed, then the new name must be used when checking it in. An error is raised if the path name refers to no resource.
`unCheckOut(pathname VARCHAR2) RETURN DBMS_XDB.RESID_TYPE;`	Check in a checked-out resource. The argument is the path name of the checked-out resource. Returns the resource id of the version before the resource was checked out. This is not an auto-commit SQL operation. You need not use the same path name that was used for check-out. However, the `unCheckOut` path name and the check-out path name must reference the same resource, or else results are unpredictable. If the resource has been renamed, then the new name must be used for `unCheckOut`. An error is raised if the path name refers to no resource.

Table 24–2 (Cont.) PL/SQL Functions and Procedures in Package DBMS_XDB_VERSION

Function or Procedure	Description
`getPredecessors(pathname VARCHAR2) RETURN RESID_LIST_TYPE;`	Given a path name that references a version resource or a version-controlled resource, return the predecessors of the resource. Retrieving predecessors by resource ID, using function getPredsByRESID is more efficient than by path name, using function `getPredecessors`.
`getPredsByRESID(resid DBMS_XDB.RESID_TYPE) RETURN RESID_LIST_TYPE;`	The list of predecessors returned has only one element (the parent): Oracle XML DB does not support version branching.
`getSuccessors(pathname VARCHAR2) RETURN RESID_LIST_TYPE;`	Given a version resource or a version-controlled resource, return the successors of the resource. Retrieving successors by resource ID, using function getSuccsByRESID is more efficient than by path name, using function `getSuccessors`.
`getSuccsByRESID(resid DBMS_XDB.RESID_TYPE) RETURN RESID_LIST_TYPE;`	The list of successors returned has only one element (the parent): Oracle XML DB does not support version branching.
`getResourceByRESID(resid DBMS_XDB.RESID_TYPE) RETURN XMLType;`	Given a resource ID, return the resource as an `XMLType` instance.

25

Accessing the Repository using RESOURCE_VIEW and PATH_VIEW

This chapter describes the predefined public views, RESOURCE_VIEW and PATH_VIEW, that provide access to Oracle XML DB repository data. It discusses Oracle SQL functions under_path and equals_path that query resources based on their path names and path and depth that return resource path names and depths, respectively.

This chapter contains these topics:

- Overview of Oracle XML DB RESOURCE_VIEW and PATH_VIEW
- RESOURCE_VIEW and PATH_VIEW SQL Functions
- Using RESOURCE_VIEW and PATH_VIEW SQL Functions
- Working with Multiple Oracle XML DB Resources
- Performance Tuning of Oracle XML DB Repository Operations
- Searching for Resources using Oracle Text

See Also:

- *Oracle Database Reference* for more information about view PATH_VIEW
- *Oracle Database Reference* for more information about view RESOURCE_VIEW

Overview of Oracle XML DB RESOURCE_VIEW and PATH_VIEW

Figure 25–1 shows how Oracle XML DB RESOURCE_VIEW and PATH_VIEW provide a mechanism for using SQL to access data stored in Oracle XML DB Repository. Data stored in the repository using protocols such as FTP and WebDAV, or using application program interfaces (APIs), can be accessed in SQL using RESOURCE_VIEW values and PATH_VIEW values.

RESOURCE_VIEW consists of a resource, itself an XMLType, that contains the name of the resource that can be queried, its ACLs, and its properties, static or extensible.

- If the content comprising the resource is XML data stored somewhere in an XMLType table or view, then the RESOURCE_VIEW points to the XMLType row that stores the content.
- If the content is not XML data, then the RESOURCE_VIEW stores it as a LOB.

> **Note:** As of Oracle Database Release 11.2.0.1.0, repository content stored in line as a LOB uses SecureFile LOB storage. Prior to that, it used BasicFile LOB storage.

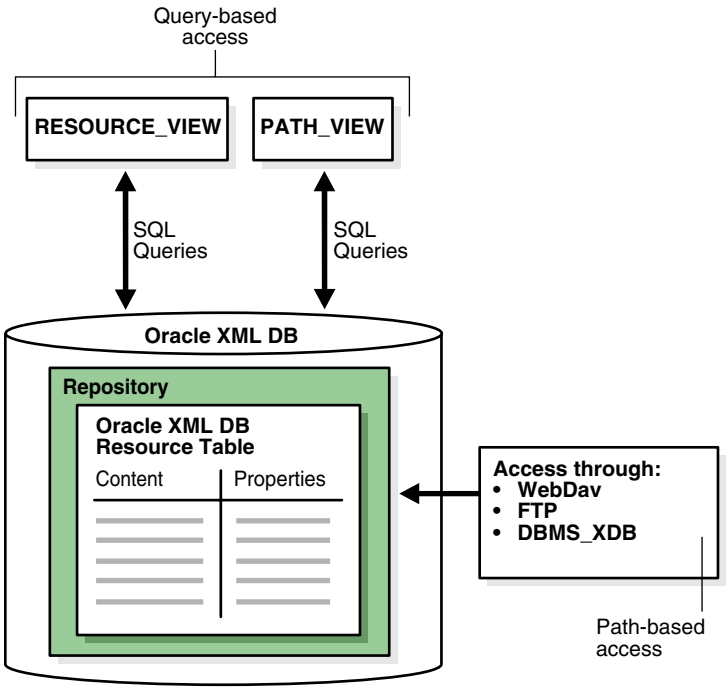
Parent-child relationships between folders (necessary to construct the hierarchy) are maintained and traversed efficiently using the hierarchical repository index. Text indexes are available to search the properties of a resource, and internal B-tree indexes over Names and ACLs speed up access to these attributes of the Resource `XMLType`.

`RESOURCE_VIEW` and `PATH_VIEW`, along with PL/SQL package `DBMS_XDB`, provide all query-based access to Oracle XML DB and DML functionality that is available through the API.

The base table for `RESOURCE_VIEW` is `XDB.XDB$RESOURCE`. Access this table only through `RESOURCE_VIEW` or the `DBMS_XDB` API.

> **See Also:** Chapter 3, "Using Oracle XML DB"

Figure 25–1 Accessing Repository Resources using RESOURCE_VIEW and PATH_VIEW



> **Note:** Neither `RESOURCE_VIEW` nor `PATH_VIEW` contains the *root* folder (`/`) resource. All other repository resources are included.

RESOURCE_VIEW Definition and Structure

The `RESOURCE_VIEW` contains one row for each resource in Oracle XML DB Repository (except for the root folder resource). Table 25–1 describes its structure.

Table 25–1 Structure of RESOURCE_VIEW

Column	Data Type	Description
RES	XMLType	A resource in the repository (except for the root folder resource)
ANY_PATH	VARCHAR2	An (absolute) path to the resource
RESID	RAW	Resource OID, which is a unique handle to the resource

PATH_VIEW Definition and Structure

The PATH_VIEW contains one row for each unique path to access a resource in Oracle XML DB Repository (except for the root folder resource). Each resource may have multiple paths, also called **links**. Table 25–2 describes its structure.

Table 25–2 Structure of PATH_VIEW

Column	Data Type	Description
PATH	VARCHAR2	An (absolute) path to repository resource RES
RES	XMLType	The resource referred to by column PATH
LINK	XMLType	Link property
RESID	RAW	Resource OID

Figure 25–2 illustrates the structure of RESOURCE_VIEW and PATH_VIEW.

The path in the RESOURCE_VIEW is an arbitrary one and one of the accessible paths that can be used to access that resource. Oracle SQL function under_path lets applications search for resources that are contained (recursively) within a particular folder, get the resource depth, and so on. Each row in the PATH_VIEW and RESOURCE_VIEW columns is of XMLType. DML on repository views can be used to insert, rename, delete, and update resource properties and contents. Programmatic APIs must be used for some operations, such as creating links to existing resources.

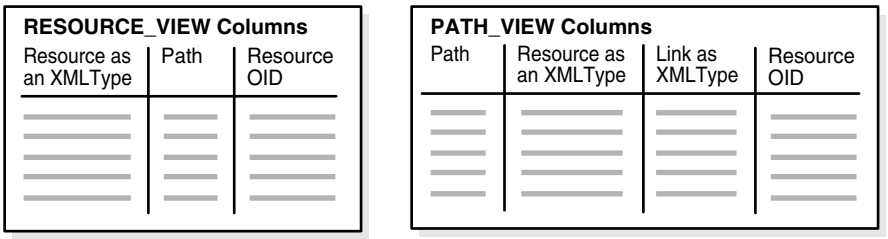
Paths in the ANY_PATH column of the RESOURCE_VIEW and the PATH column in the PATH_VIEW are *absolute* paths: they start at the root.

> **Note:** Test resource paths for equality using Oracle SQL function equals_path: equals_path('/my/path') = 1. Do *not* test ANY_PATH for equality against an absolute path: ANY_PATH = '/my/path'.

Paths returned by the path function are *relative* paths under the path name specified by function under_path. For example, if there are two resources referenced by path names /a/b/c and /a/d, respectively, then a path expression that retrieves paths under folder /a returns relative paths b/c and d.

When there are multiple hard links to the same resource, only paths under the path name specified by function under_path are returned. If /a/b/c, /a/b/d, and /a/e are all links to the same resource, then a query on PATH_VIEW that retrieves all of the paths under /a/b returns only /a/b/c and /a/b/d, not /a/e.

Figure 25–2 RESOURCE_VIEW and PATH_VIEW Structure



Understanding the Difference Between RESOURCE_VIEW and PATH_VIEW

Views `RESOURCE_VIEW` and `PATH_VIEW` differ as follows:

- `PATH_VIEW` displays *all* the path names to a particular resource. `RESOURCE_VIEW` displays *one* of the possible path names to the resource
- `PATH_VIEW` also displays the properties of the link

Figure 25–3 illustrates this difference between `RESOURCE_VIEW` and `PATH_VIEW`.

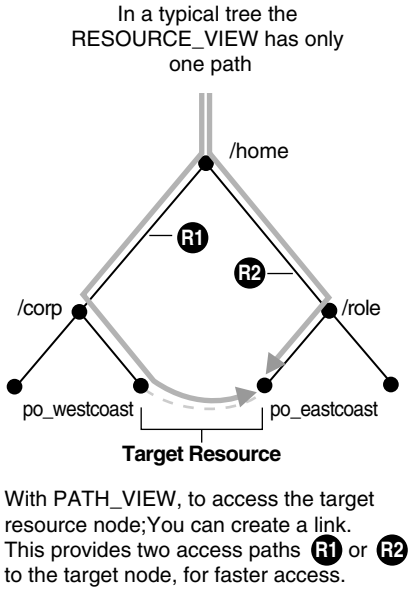
Because many Internet applications only need one URL to access a resource, `RESOURCE_VIEW` is widely applicable.

`PATH_VIEW` contains the *link* properties and resource properties, whereas the `RESOURCE_VIEW` only contains resource properties.

Whenever possible, use `RESOURCE_VIEW`, not `PATH_VIEW`, for better performance. Because it handles the information for multiple paths, `PATH_VIEW` access can be slower. If you use `RESOURCE_VIEW`, then the database can determine that only one path is needed, and the index can do less work to determine all the possible paths.

Note: When using the `RESOURCE_VIEW`, if you specify a path with functions `under_path` or `equals_path`, then they find the resource regardless of whether or not that path is the arbitrary one chosen to normally appear with that resource using `RESOURCE_VIEW`.

Figure 25–3 RESOURCE_VIEW and PATH_VIEW Explained



RESOURCE_VIEW Example:
select path(1) from RESOURCE_VIEW where under_path(res, '/sys',1);
displays one path to the resource:
/home/corp/po_westcoast

PATH_VIEW Example:
select path from PATH_VIEW;
displays all pathnames to the resource:
/home/corp/po_westcoast
/home/role/po_eastcoast

Operations You Can Perform using UNDER_PATH and EQUALS_PATH

You can perform the following operations using Oracle SQL functions `under_path` and `equals_path`:

- Given a path name:
  - Get a resource or its OID
  - List the directory given by the path name
  - Create a resource
  - Delete a resource
  - Update a resource
- Given a condition that uses `under_path` or other SQL functions:
  - Update resources
  - Delete resources
  - Get resources or their OID

See the "Using RESOURCE_VIEW and PATH_VIEW SQL Functions" and `equals_path`.

RESOURCE_VIEW and PATH_VIEW SQL Functions

This section describes Oracle SQL functions that are applicable to `RESOURCE_VIEW` and `PATH_VIEW`.

UNDER_PATH SQL Function

Oracle SQL function `under_path` uses the hierarchical index of Oracle XML DB Repository to return the paths to all hard links under a particular path. This index is designed to speed access when traversing a path (the most common usage).

If the other parts of the query predicate are very selective, however, then a functional implementation of `under_path` can be chosen that walks back up the repository. This can be more efficient, because fewer links must be traversed. Figure 25–4 shows the `under_path` syntax.

Figure 25–4 UNDER_PATH Syntax

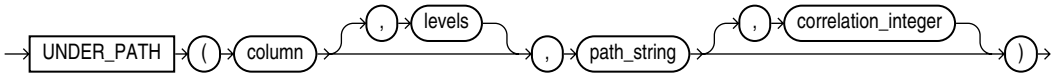


Table 25–3 details the signature of Oracle SQL function `under_path`.

Table 25–3 UNDER_PATH SQL Function Signature

Syntax	Description
`under_path(`*`resource_column`*`, `*`pathname`*`);`	Determines whether a resource is under a specified path. Parameters: ▪ *`resource_column`* –The column name or column alias of the `RESOURCE` column in the `PATH_VIEW` or `RESOURCE_VIEW`. ▪ *`pathname`* – The path name to resolve.
`under_path(`*`resource_column`*`, `*`depth`*`, `*`pathname`*`);`	Determines whether a resource is under a specified path, with a depth argument to restrict the number of levels to search. Parameters: ▪ *`resource_column`* – The column name or column alias of the `RESOURCE` column in the `PATH_VIEW` or `RESOURCE_VIEW`. ▪ `depth` – The maximum depth to search. A nonnegative integer. ▪ *`pathname`* – The path name to resolve.
`under_path(`*`resource_column`*`, `*`pathname`*`, `*`correlation`*`);`	Determines if a resource is under a specified path, with a correlation argument for related SQL functions. Parameters: ▪ *`resource_column`* – The column name or column alias of the `RESOURCE` column in the `PATH_VIEW` or `RESOURCE_VIEW`. ▪ *`pathname`* – The path name to resolve. ▪ *`correlation`* – An integer that can be used to correlate `under_path` with related SQL functions (`path` and `depth`).
`under_path(`*`resource_column`*`, `*`depth`*`, `*`pathname`*`, `*`correlation`*`);`	Determines if a resource is under a specified path with a depth argument to restrict the number of levels to search, and with a correlation argument for related SQL functions. Parameters: ▪ *`resource_column`* – The column name or column alias of the `RESOURCE` column in the `PATH_VIEW` or `RESOURCE_VIEW`. ▪ *`depth`* – The maximum depth to search. A nonnegative integer. ▪ *`pathname`* – The path name to resolve. ▪ *`correlation`* – An integer that can be used to correlate `under_path` with related SQL functions (`path` and `depth`). Note that only one of the accessible paths to the resource must be under the path argument for a resource to be returned.

> **Note:** Function `under_path` does not follow weak links, because such traversal could lead to cycles. A weak-link argument to `under_path` is resolved correctly, but weak links are not followed when traversing resources under that path.

EQUALS_PATH SQL Function

Oracle SQL function `equals_path` is used to find the resource with the specified path name. It is functionally equivalent to `under_path` with a depth restriction of zero.

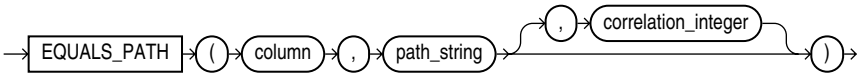
```
equals_path(resource_column, pathname);
```

where:

- `resource_column` is the column name or column alias of the `RESOURCE` column in `PATH_VIEW` or `RESOURCE_VIEW`.
- `pathname` is the (absolute) path name to resolve. This can contain components that are hard or weak resource links.

Figure 25–5 illustrates the complete `equals_path` syntax.

Figure 25–5 EQUALS_PATH Syntax



> **Note:**
>
> - Test resource paths for equality using Oracle SQL function `equals_path`: `equals_path('/my/path') = 1`. Do *not* test `ANY_PATH` for equality against an absolute path: `ANY_PATH = '/my/path'`.
> - Use bind variables, instead of hard-coded strings, with `equals_path`.

PATH SQL Function

Oracle SQL function `path` returns the relative path name of the resource under the specified `pathname` argument. Note that the path column in the `RESOURCE_VIEW` always contains the absolute path of the resource. The `path` syntax is:

```
path(correlation);
```

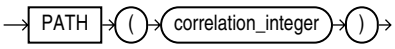
where:

- `correlation` is an integer that can be used to correlate `under_path` with related SQL functions (`path` and `depth`).

> **Note:** If a path is not under the specified `pathname` argument, a `NULL` value is returned as the output of the current path.

Figure 25–6 illustrates the `path` syntax.

Figure 25–6 PATH Syntax



DEPTH SQL Function

Oracle SQL function `depth` returns the folder depth of the resource under the specified starting path.

```
depth(correlation);
```

where:

`correlation` is an integer that can be used to correlate `under_path` with related SQL functions (`path` and `depth`).

Using RESOURCE_VIEW and PATH_VIEW SQL Functions

The following `RESOURCE_VIEW` and `PATH_VIEW` examples use Oracle SQL functions `under_path`, `equals_path`, `path`, and `depth`.

Accessing Repository Data Paths, Resources and Links: Examples

The following examples illustrate how you can access paths, resources, and link properties in Oracle XML DB Repository. The first few examples use resources specified by the following paths:

```
/a/b/c
/a/b/c/d
/a/e/c
/a/e/c/d
```

Example 25–1 uses Oracle SQL function `path` to retrieve the *relative* paths under path `/a/b`.

Example 25–1 Determining Paths Under a Path: Relative

```
SELECT path(1) FROM RESOURCE_VIEW WHERE under_path(RES, '/a/b', 1) = 1;

PATH(1)
-------
c
c/d

2 rows selected.
```

Example 25–2 uses `ANY_PATH` to retrieve the *absolute* paths under path `/a/b`.

Example 25–2 Determining Paths Under a Path: Absolute

```
SELECT ANY_PATH FROM RESOURCE_VIEW WHERE under_path(RES, '/a/b') = 1;

ANY_PATH
--------
/a/b/c
/a/b/c/d

2 rows selected.
```

Example 25–3 is the same as Example 25–2, except that the test is *not*-equals (`!=`) instead of equals (=). The query in Example 25–3 finds *all paths in the repository* that are *not* under path `/a/b`.

Example 25–3 Determining Paths Not Under a Path

```
SELECT ANY_PATH FROM RESOURCE_VIEW WHERE under_path(RES, '/a/b') != 1

ANY_PATH
--------
/a
/a/b
/a/e
/a/e/c
/a/e/c/d
/home
/home/OE
/home/OE/PurchaseOrders
/home/OE/PurchaseOrders/2002
/home/OE/PurchaseOrders/2002/Apr
/home/OE/PurchaseOrders/2002/Apr/AMCEWEN-20021009123336171PDT.xml
/home/OE/PurchaseOrders/2002/Apr/AMCEWEN-20021009123336271PDT.xml
/home/OE/PurchaseOrders/2002/Apr/EABEL-20021009123336251PDT.xml
. . .
/public
/sys
/sys/acls
/sys/acls/all_all_acl.xml
/sys/acls/all_owner_acl.xml
/sys/acls/bootstrap_acl.xml
/sys/acls/ro_all_acl.xml
/sys/apps
/sys/apps/plsql
/sys/apps/plsql/xs
/sys/apps/plsql/xs/netaclrc.xml
/sys/apps/plsql/xs/netaclsc.xml
/sys/databaseSummary.xml
/sys/log
/sys/schemas
/sys/schemas/OE
/sys/schemas/OE/localhost:8080
. . .

326 rows selected.
```

Example 25–4 Determining Paths using Multiple Correlations

```
SELECT ANY_PATH, path(1), path(2)
  FROM RESOURCE_VIEW
  WHERE under_path(RES, '/a/b', 1) = 1 OR under_path(RES, '/a/e', 2) = 1;

ANY_PATH   PATH(1)  PATH(2)
---------- -------- --------
/a/b/c     c
/a/b/c/d   c/d
/a/e/c              c
/a/e/c/d            c/d

4 rows selected.
```

Example 25–5 Relative Path Names for Three Levels of Resources

```
SELECT path(1) FROM RESOURCE_VIEW WHERE under_path(RES, 3, '/sys', 1) = 1;
```

This produces a result similar to the following.

```
PATH(1)
-------
acls
acls/all_all_acl.xml
acls/all_owner_acl.xml
acls/bootstrap_acl.xml
acls/ro_all_acl.xml
apps
apps/plsql
apps/plsql/xs
databaseSummary.xml
log
schemas
schemas/OE
schemas/OE/localhost:8080
schemas/PUBLIC
schemas/PUBLIC/www.w3.org
schemas/PUBLIC/xmlns.oracle.com

93 rows selected.
```

Example 25–6 Extracting Resource Metadata using UNDER_PATH

```
SELECT ANY_PATH,
       XMLQuery('declare namespace ns = "http://xmlns.oracle.com/xdb/XDBResource.xsd"; (: :)
                 $r/ns:Resource' PASSING RES AS "r" RETURNING CONTENT)
  FROM RESOURCE_VIEW WHERE under_path(RES, '/sys') = 1;
```

This produces a result similar to the following:

```
ANY_PATH
--------
XMLQUERY('DECLARENAMESPACENS="HTTP://XMLNS.ORACLE.COM/XDB/XDBRESOURCE.XSD";(::)$
--------------------------------------------------------------------------------
/sys/acls
<Resource xmlns="http://xmlns.oracle.com/xdb/XDBResource.xsd">
  <CreationDate>2008-06-25T13:17:45.164662</CreationDate>
  <ModificationDate>2008-06-25T13:17:47.865163</ModificationDate>
  <DisplayName>acls</DisplayName>
  <Language>en-US</Language>
  <CharacterSet>UTF-8</CharacterSet>
  <ContentType>application/octet-stream</ContentType>
  <RefCount>1</RefCount>
</Resource>

/sys/acls/all_all_acl.xml
<Resource xmlns="http://xmlns.oracle.com/xdb/XDBResource.xsd">
  <CreationDate>2008-06-25T13:17:47.759806</CreationDate>
  <ModificationDate>2008-06-25T13:17:47.759806</ModificationDate>
  <DisplayName>all_all_acl.xml</DisplayName>
  <Language>en-US</Language>
  <CharacterSet>UTF-8</CharacterSet>
  <ContentType>text/xml</ContentType>
  <RefCount>1</RefCount>
</Resource>
. . .
```

```
41 rows selected.
```

Example 25–7 Using Functions PATH and DEPTH with PATH_VIEW

```
SELECT path(1) path, depth(1) depth FROM PATH_VIEW
  WHERE under_path(RES, 3, '/sys', 1) = 1;
```

This produces a result similar to the following:

```
PATH                                     DEPTH
----                                     -----
acls                                     1
acls/all_all_acl.xml                     2
acls/all_owner_acl.xml                   2
acls/bootstrap_acl.xml                   2
acls/ro_all_acl.xml                      2
apps                                     1
apps/plsql                               2
apps/plsql/xs                            3
databaseSummary.xml                      1
log                                      1
schemas                                  1
schemas/OE                               2
schemas/OE/localhost:8080                3
schemas/PUBLIC                           2
schemas/PUBLIC/www.w3.org                3
schemas/PUBLIC/xmlns.oracle.com          3
. . .
```

Example 25–8 Extracting Link and Resource Information from PATH_VIEW

```
SELECT PATH,
       XMLCast(XMLQuery(
                 'declare namespace ns =
                          "http://xmlns.oracle.com/xdb/XDBResource.xsd"; (: :)
                  $l/ns:LINK/ns:Name' PASSING LINK AS "l" RETURNING CONTENT)
               AS VARCHAR2(256)),
       XMLCast(XMLQuery(
                 'declare namespace ns =
                          "http://xmlns.oracle.com/xdb/XDBResource.xsd"; (: :)
                  $l/ns:LINK/ns:ParentName' PASSING LINK AS "l" RETURNING CONTENT)
               AS VARCHAR2(256)),
       XMLCast(XMLQuery(
                 'declare namespace ns =
                          "http://xmlns.oracle.com/xdb/XDBResource.xsd"; (: :)
                  $l/ns:LINK/ns:ChildName' PASSING LINK AS "l" RETURNING CONTENT)
               AS VARCHAR2(256)),
       XMLCast(XMLQuery(
                 'declare namespace ns =
                          "http://xmlns.oracle.com/xdb/XDBResource.xsd"; (: :)
                  $r/ns:Resource/ns:DisplayName'
                    PASSING RES AS "r" RETURNING CONTENT)
               AS VARCHAR2(128))
  FROM PATH_VIEW WHERE PATH LIKE '/sys%';
```

This produces a result similar to the following:

```
/sys/schemas/PUBLIC/www.w3.org/1999/xlink.xsd
xlink.xsd

/sys/schemas/PUBLIC/www.w3.org/1999/xlink
```

```
xlink

/sys/schemas/PUBLIC/www.w3.org/1999/csx.xlink.xsd
csx.xlink.xsd

. . .

118 rows selected.
```

Example 25–9 All Repository Paths to a Certain Depth Under a Path

```
SELECT path(1) FROM PATH_VIEW WHERE under_path(RES, 3, '/sys', 1) > 0;
```

This produces a result similar to the following:

```
PATH(1)
-------
acls
acls/all_all_acl.xml
acls/all_owner_acl.xml
acls/bootstrap_acl.xml
acls/ro_all_acl.xml
apps
apps/plsql
apps/plsql/xs
databaseSummary.xml
log
principals
principals/groups
principals/users
schemas
schemas/PUBLIC
schemas/PUBLIC/www.opengis.net
schemas/PUBLIC/www.w3.org
schemas/PUBLIC/xmlns.oracle.com
workspaces
. . .

43 rows selected.
```

Example 25–10 Locating a Repository Path using EQUALS_PATH

```
SELECT ANY_PATH FROM RESOURCE_VIEW WHERE equals_path(RES, '/sys') > 0;

ANY_PATH
--------
/sys

1 row selected.
```

Example 25–11 Retrieve RESID of a Given Resource

```
SELECT RESID FROM RESOURCE_VIEW
  WHERE XMLCast(XMLQuery(
                  'declare namespace ns =
                     "http://xmlns.oracle.com/xdb/XDBResource.xsd"; (: :)
                   $r/ns:Resource/ns:DisplayName'
                  PASSING RES AS "r" RETURNING CONTENT)
                AS VARCHAR2(128))
        = 'example';
```

This produces a result similar to the following:

```
RESID
--------------------------------
F301A10152470252E030578CB00B432B

1 row selected.
```

Example 25–12 Obtaining the Path Name of a Resource from its RESID

```
DECLARE
  resid_example RAW(16);
  path          VARCHAR2(4000);
BEGIN
  SELECT RESID INTO resid_example FROM RESOURCE_VIEW
    WHERE XMLCast(XMLQuery(
                     'declare namespace ns =
                        "http://xmlns.oracle.com/xdb/XDBResource.xsd"; (: :)
                      $r/ns:Resource/ns:DisplayName'
                     PASSING RES AS "r" RETURNING CONTENT)
                   AS VARCHAR2(128))

          = 'example';
  SELECT ANY_PATH INTO path FROM RESOURCE_VIEW WHERE RESID = resid_example;
  DBMS_OUTPUT.put_line('The path is: ' || path);
END;
/
The path is: /public/example

PL/SQL procedure successfully completed.
```

Example 25–13 Folders Under a Given Path

```
SELECT ANY_PATH FROM RESOURCE_VIEW
  WHERE under_path(RES, 1, '/sys') = 1
    AND XMLExists('declare namespace ns =
                      "http://xmlns.oracle.com/xdb/XDBResource.xsd"; (: :)
                    $r/ns:Resource[@Container = xs:boolean("true")]'
                   PASSING RES AS "r");
```

This produces a result like the following:

```
ANY_PATH
--------
/sys/acls
/sys/apps
/sys/log
/sys/schemas

4 rows selected.
```

Example 25–14 Joining RESOURCE_VIEW with an XMLType Table

```
SELECT ANY_PATH, XMLQuery('$p/PurchaseOrder/LineItems'
                          PASSING po.OBJECT_VALUE AS "p" RETURNING CONTENT)
  FROM purchaseorder po, RESOURCE_VIEW rv
  WHERE ref(po)
        = XMLCast(XMLQuery('declare default element namespace
                             "http://xmlns.oracle.com/xdb/XDBResource.xsd"; (: :)
                             fn:data(/Resource/XMLRef)'
                            PASSING rv.RES RETURNING CONTENT)
                   AS REF XMLType)
```

```
    AND ROWNUM < 2;

ANY_PATH
--------------------------------------------------------------------------------
XMLQUERY('$P/PURCHASEORDER/LINEITEMS'PASSINGPO.OBJECT_VALUEAS"P"RETURNINGCONTENT
--------------------------------------------------------------------------------
/home/OE/PurchaseOrders/2002/Apr/AMCEWEN-20021009123336171PDT.xml
<LineItems>
  <LineItem ItemNumber="1">
    <Description>Salesman</Description>
    <Part Id="37429158920" UnitPrice="39.95" Quantity="2"/>
  </LineItem>
  <LineItem ItemNumber="2">
    <Description>Big Deal on Madonna Street</Description>
    <Part Id="37429155424" UnitPrice="29.95" Quantity="1"/>
  </LineItem>
  <LineItem ItemNumber="3">
    <Description>Hearts and Minds</Description>
    <Part Id="37429166321" UnitPrice="39.95" Quantity="1"/>
  </LineItem>

. . .

  <LineItem ItemNumber="23">
    <Description>Great Expectations</Description>
    <Part Id="37429128022" UnitPrice="39.95" Quantity="4"/>
  </LineItem>
</LineItems>

1 row selected.
```

Deleting Repository Resources: Examples

The examples in this section illustrate how to delete resources and paths.

If you delete only *leaf* resources, then you can use `DELETE FROM RESOURCE_VIEW`, as in Example 25–15.

Example 25–15 Deleting Resources

```
DELETE FROM RESOURCE_VIEW WHERE equals_path(RES, '/public/myfile') = 1';
```

For multiple links to the same resource, deleting from `RESOURCE_VIEW` deletes the resource together with *all* of its links. Deleting from `PATH_VIEW` deletes only the link with the specified path.

Example 25–16 illustrates this.

Example 25–16 Deleting Links to Resources

Suppose that `'/home/myfile1'` is a link to `'/public/myfile'`:

```
CALL DBMS_XDB.link('/public/myfile', '/home', 'myfile1');
```

The following SQL DML statement deletes everything in Oracle XML DB Repository that is found at path `/home/myfile1` – both the link and the resource:

```
DELETE FROM RESOURCE_VIEW WHERE equals_path(RES, '/home/myfile1') = 1;
```

The following DML statement deletes *only the link* with path `/home/file1`:

```
DELETE FROM PATH_VIEW WHERE equals_path(RES, '/home/file1') = 1;
```

Deleting Nonempty Folder Resources

The DELETE DML operator is not allowed on a nonempty folder. If you try to delete a nonempty folder, you must first delete its contents and then delete the resulting empty folder. This rule must be applied recursively to any folders contained in the target folder.

However, the order of the paths returned from a WHERE clause is not guaranteed, and the DELETE operator does not allow an ORDER BY clause in its table-expression subclause. You *cannot* do the following:

```
DELETE FROM (SELECT 1 FROM RESOURCE_VIEW
               WHERE under_path(RES, '/public', 1) = 1
               ORDER BY depth(1) DESCENDING);
```

Example 25–17 illustrates how to delete a nonempty folder: folder example is deleted, along with its subfolder example1.

Example 25–17 Deleting a Nonempty Folder

```
SELECT PATH FROM PATH_VIEW WHERE under_path(RES, '/home/US1') = 1;

PATH
--------------------------
/home/US1/example
/home/US1/example/example1

2 rows selected.

DECLARE
  CURSOR c1 IS
    SELECT ANY_PATH p FROM RESOURCE_VIEW
      WHERE under_path(RES, '/home/US1', 1) = 1
        AND XMLExists('declare namespace ns =
                          "http://xmlns.oracle.com/xdb/XDBResource.xsd"; (: :)
                        $r/ns:Resource[ns:Owner="US1"]' PASSING RES AS "r")
      ORDER BY depth(1) DESC;
  del_stmt VARCHAR2(500) :=
    'DELETE FROM RESOURCE_VIEW WHERE equals_path(RES, :1)=1';
BEGIN
  FOR r1 IN c1 LOOP
    EXECUTE IMMEDIATE del_stmt USING r1.p;
  END LOOP;
END;
/

PL/SQL procedure successfully completed.

SELECT PATH FROM PATH_VIEW WHERE under_path(RES, '/home/US1') = 1;

no rows selected
```

> **Note:** As always, take care to avoid deadlocks with concurrent transactions when operating on multiple rows.

Updating Repository Resources: Examples

This section illustrates how to update resources and paths.

Example 25–18 changes the resource at path /test/HR/example/paper.

Example 25–18 Updating a Resource

This is the complete resource before the update operation:

```
SELECT XMLSerialize(DOCUMENT r.RES AS CLOB)
  FROM RESOURCE_VIEW r WHERE equals_path(r.RES, '/test/HR/example/paper') = 1;

XMLSERIALIZE(DOCUMENTR.RESASCLOB)
--------------------------------------------------------------------------------
<Resource xmlns="http://xmlns.oracle.com/xdb/XDBResource.xsd" Hidden="false" Inv
alid="false" Container="false" CustomRslv="false" VersionHistory="false" StickyR
ef="true">
  <CreationDate>2005-04-29T16:30:01.588835</CreationDate>
  <ModificationDate>2005-04-29T16:30:01.588835</ModificationDate>
  <DisplayName>paper</DisplayName>
  <Language>en-US</Language>
  <CharacterSet>UTF-8</CharacterSet>
  <ContentType>application/octet-stream</ContentType>
  <RefCount>1</RefCount>
  <ACL>
    <acl description="Public:All privileges to PUBLIC" xmlns="http://xmlns.oracl
e.com/xdb/acl.xsd" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:sch
emaLocation="http://xmlns.oracle.com/xdb/acl.xsd                          http:
//xmlns.oracle.com/xdb/acl.xsd">
      <ace>
        <principal>PUBLIC</principal>
        <grant>true</grant>
        <privilege>
          <all/>
        </privilege>
      </ace>
    </acl>
  </ACL>
  <Owner>TESTUSER1</Owner>
  <Creator>TESTUSER1</Creator>
  <LastModifier>TESTUSER1</LastModifier>
  <SchemaElement>http://xmlns.oracle.com/xdb/XDBSchema.xsd#binary</SchemaElement
>
  <Contents>
    <binary>4F7261636C65206F7220554E4958</binary>
  </Contents>
</Resource>

1 row selected.
```

All of the XML elements shown here are resource *metadata* elements, with the exception of `Contents`, which contains the resource *content*.

This `UPDATE` statement updates (only) the `DisplayName` metadata element.

```
UPDATE RESOURCE_VIEW r
  SET r.RES = updateXML(r.RES, '/Resource/DisplayName/text()', 'My New Paper')
    WHERE equals_path(r.RES, '/test/HR/example/paper') = 1;

1 row updated.

SELECT XMLSerialize(DOCUMENT r.RES AS CLOB)
  FROM RESOURCE_VIEW r WHERE equals_path(r.RES, '/test/HR/example/paper') = 1;

XMLSERIALIZE(DOCUMENTR.RESASCLOB)
--------------------------------------------------------------------------------
<Resource xmlns="http://xmlns.oracle.com/xdb/XDBResource.xsd" Hidden="false" Inv
```

```
alid="false" Container="false" CustomRslv="false" VersionHistory="false" StickyR
ef="true">
  <CreationDate>2005-04-29T16:30:01.588835</CreationDate>
  <ModificationDate>2005-04-29T16:30:01.883838</ModificationDate>
  <DisplayName>My New Paper</DisplayName>
  <Language>en-US</Language>

  . . .

  <Contents>
    <binary>4F7261636C65206F7220554E4958</binary>
  </Contents>
</Resource>

1 row selected.
```

> **See Also:** Chapter 29, "User-Defined Repository Metadata" for additional examples of updating resource metadata

Note that, by default, the `DisplayName` element content, `paper`, was the same text as the last location step of the resource path, `/test/HR/example/paper`. This is only the default value, however. The `DisplayName` is independent of the resource path, so updating it does not change the path.

Element `DisplayName` is defined by the WebDAV standard, and it is recognized by WebDAV applications. Applications, such as an FTP client, that are not WebDAV-based do not recognize the `DisplayName` of a resource. An FTP client lists the resource as `paper` (using FTP command `ls`, for example) even after the `UPDATE` operation.

Example 25–19 changes the path for the resource from `/test/HR/example/paper` to `/test/myexample`. It is analogous to using the UNIX or Linux command `mv /test/HR/example/paper /test/myexample`.

Example 25–19 Updating a Path in the PATH_VIEW

```
SELECT ANY_PATH FROM RESOURCE_VIEW WHERE under_path(RES, '/test') = 1;

ANY_PATH
--------
/test/HR
/test/HR/example
/test/HR/example/paper

3 rows selected.

UPDATE PATH_VIEW
  SET PATH = '/test/myexample' WHERE PATH = '/test/HR/example/paper';

ANY_PATH
--------
/test/HR
/test/HR/example
/test/myexample

3 rows selected.
```

> **See Also:** Table 21–3, " Accessing Oracle XML DB Repository: API Options" on page 21-15 for additional examples that use SQL functions that apply to `RESOURCE_VIEW` and `PATH_VIEW`

Working with Multiple Oracle XML DB Resources

The repository operations listed in Table 21–3 on page 21-15 typically apply to a single resource at a time. To perform the same operation on multiple Oracle XML DB resources, or to find one or more Oracle XML DB resources that meet a certain set of criteria, use SQL with `RESOURCE_VIEW` and `PATH_VIEW`.

For example, you can perform the following operations:

- Updating based on attributes – see Example 25–20
- Finding resources inside a folder – see Example 25–21
- Copying a set of Oracle XML DB resources – see Example 25–22

Example 25–20 Updating Resources Based on Attributes

```
UPDATE RESOURCE_VIEW
  SET RES = updateXML(RES, '/Resource/DisplayName/text()', 'My New Paper')
    WHERE XMLCast(XMLQuery(
                     'declare namespace ns =
                        "http://xmlns.oracle.com/xdb/XDBResource.xsd"; (: :)
                      $r/ns:Resource/ns:DisplayName'
                     PASSING RES AS "r" RETURNING CONTENT)
                   AS VARCHAR2(128))
          = 'My Paper';

1 row updated.

SELECT ANY_PATH FROM RESOURCE_VIEW
  WHERE XMLCast(XMLQuery('declare namespace ns =
                            "http://xmlns.oracle.com/xdb/XDBResource.xsd"; (: :)
                          $r/ns:Resource/ns:DisplayName'
                         PASSING RES AS "r" RETURNING CONTENT)
                 AS VARCHAR2(128))
          = 'My New Paper';

ANY_PATH
---------------
/test/myexample

1 row selected.
```

Example 25–21 Finding Resources Inside a Folder

```
SELECT ANY_PATH FROM RESOURCE_VIEW
  WHERE under_path(resource, '/sys/schemas/PUBLIC/xmlns.oracle.com/xdb') = 1;

ANY_PATH
----------------------------------------------------------------
/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/XDBFolderListing.xsd
/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/XDBResource.xsd
/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/XDBSchema.xsd
/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/XDBStandard.xsd
/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/acl.xsd
/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/dav.xsd
/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/log
/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/log/ftplog.xsd
/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/log/httplog.xsd
/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/log/xdblog.xsd
/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/stats.xsd
/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/xdbconfig.xsd
```

```
12 rows selected.
```

The SQL DML statement in Example 25–22 copies all of the resources in folder `public` to folder `newlocation`. It is analogous to the UNIX or Linux command `cp /public/* /newlocation`. Target folder `newlocation` must exist before the copy.

Example 25–22 Copying Resources

```
SELECT PATH FROM PATH_VIEW WHERE under_path(RES, '/test') = 1;

PATH
-----------------
/test/HR
/test/HR/example
/test/myexample

3 rows selected.

INSERT INTO PATH_VIEW
  SELECT '/newlocation/' || path(1), RES, LINK, NULL FROM PATH_VIEW
    WHERE under_path(RES, '/test', 1) = 1
    ORDER BY depth(1);

3 rows created.

SELECT PATH FROM PATH_VIEW WHERE under_path(RES, '/newlocation') = 1;

PATH
------------------------
/newlocation/HR
/newlocation/HR/example
/newlocation/myexample

3 rows selected.
```

Performance Tuning of Oracle XML DB Repository Operations

This section includes some guidelines for improving the performance of repository operations such as resource creation and querying.

For optimal performance of queries on repository resources, gather statistics for the optimizer using procedure `DBMS_XDB_ADMIN.gatherRepositoryStats` after resource creation. To use `gatherRepositoryStats`, you need role `DBA` or role `XDBADMIN` with privilege `ANALYZE ANY`.

Folders that contain a large number of resources can negatively affect concurrency, particularly when many resources are created or deleted. As a rule of thumb, do not have folders that contain more than 10,000 resources. This empirical limit is based on the database block size and the average filename length.

If you create resources in bulk, perform a `COMMIT` operation at least every 1,000 resources. Performance can be negatively impacted if you commit very often or you commit less often than every 1,000 resource creations.

When creating a file resource that is an XML Schema-based document for which the XML schema is known, specify the XML schema URL as a parameter to PL/SQL function `DBMS_XDB.createResource`. This saves preparsing the document to determine the XML schema.

Oracle XML DB uses configuration file `xdbconfig.xml` for configuring the system and protocol environment. This file includes an element parameter, `resource-view-cache-size`, that defines the size in dynamic memory of the `RESOURCE_VIEW` cache. The default value is `1048576`.

The performance of some queries on `RESOURCE_VIEW` and `PATH_VIEW` can be improved by tuning `resource-view-cache-size`. In general, the bigger the cache size, the faster the query. The default `resource-view-cache-size` is appropriate for most cases, but you may want to enlarge your `resource-view-cache-size` element when querying a sizable `RESOURCE_VIEW`.

The default limits for the following elements are soft limits. The system automatically adapts when these limits are exceeded.

- `xdbcore-loadableunit-size` – This element indicates the maximum size to which a loadable unit (partition) can grow in Kilobytes. When a partition is read into memory or a partition is built while consuming a new document, the partition is built until it reaches the maximum size. The default value is 16 KB.
- `xdbcore-xobmem-bound` – This element indicates the maximum memory in kilobytes that a document is allowed to occupy. The default value is 1024 KB. Once the document exceeds this number, some loadable units (partitions) are swapped out.

See Also:

- Chapter 34, "Administering Oracle XML DB"
- *Oracle Database PL/SQL Packages and Types Reference* for information about PL/SQL function `DBMS_XDB.createResource`
- *Oracle Database 2 Day + Security Guide* for information about database schema `XDB`

Searching for Resources using Oracle Text

Table `XDB$RESOURCE` in database schema `XDB` stores the metadata and content of repository resources. You can search for resources that contain a specific keyword by using Oracle SQL function `contains` with `RESOURCE_VIEW` or `PATH_VIEW`.

Example 25–23 Find All Resources Containing "Paper"

```
SELECT PATH FROM PATH_VIEW WHERE contains(RES, 'Paper') > 0;

PATH
-----------------------
/newlocation/myexample
/test/myexample

2 rows selected.
```

Example 25–24 Find All Resources Containing "Paper" that are Under a Specified Path

```
SELECT ANY_PATH FROM RESOURCE_VIEW
   WHERE contains(RES, 'Paper') > 0 AND under_path(RES, '/test') > 0;

ANY_PATH
----------------
/test/myexample
```

```
1 row selected.
```

To evaluate such queries, you must first create a context index on the XDB$RESOURCE table. Depending on the type of documents stored in Oracle XML DB, choose one of the following options for creating your context index:

- *If Oracle XML DB contains only XML documents*, that is, no binary data, then a regular Context Index can be created on the XDB$RESOURCE table. This is the case for Example 25–24.

  ```
  CREATE INDEX xdb$resource_ctx_i ON XDB.XDB$RESOURCE(OBJECT_VALUE)
    INDEXTYPE IS CTXSYS.CONTEXT;
  ```

  See Also: Chapter 4, "XMLType Operations" and Chapter 12, "Full-Text Search Over XML Data"

- *If Oracle XML DB contains binary data* such as Microsoft Word documents, then a user filter is required to filter such documents prior to indexing. Use package DBMS_XDBT (dbmsxdbt.sql) to create and configure the Context Index.

  ```
  -- Install the package - connected as SYS
  @dbmsxdbt
  -- Create the preferences
  EXEC DBMS_XDBT.createPreferences;
  -- Create the index
  EXEC DBMS_XDBT.createIndex;
  ```

  See Also:

  - *Oracle Database PL/SQL Packages and Types Reference*, for information about installing and using DBMS_XDBT.
  - "APIs for XML" on page 1-4

Package DBMS_XDBT also includes procedures to synchronize and optimize the index. You can use procedure configureAutoSync() to automatically sync the index by using job queues.

26

Accessing the Repository using PL/SQL

This chapter describes the Oracle XML DB resource application program interface (API) for PL/SQL (PL/SQL package `DBMS_XDB`). It contains these topics:

- Overview of PL/SQL Package DBMS_XDB
- DBMS_XDB: Resource Management
- DBMS_XDB: ACL-Based Security Management
- DBMS_XDB: Configuration Management

Overview of PL/SQL Package DBMS_XDB

PL/SQL package `DBMS_XDB` is the Oracle XML DB resource application program interface (API) for PL/SQL. It is also known as the PL/SQL *foldering* API. This API provides functions and procedures to access and manage Oracle XML DB Repository resources using PL/SQL. It includes methods for managing resource security and Oracle XML DB configuration.

Oracle XML DB Repository is modeled on XML, and provides a database file system for any data. The repository maps path names (or URLs) onto database objects of `XMLType` and provides management facilities for these objects.

PL/SQL package `DBMS_XDB` is an API that you can use to manage all of the following:

- Oracle XML DB resources
- Oracle XML DB security based on access control lists (ACLs). An ACL is a list of access control entries (ACEs) that determines which principals (users and roles) have access to which resources
- Oracle XML DB configuration

See Also:

- *Oracle Database PL/SQL Packages and Types Reference*
- "APIs for XML" on page 1-4

DBMS_XDB: Resource Management

Table 26–1 describes the `DBMS_XDB` Oracle XML DB resource management functions and procedures.

Table 26–1 DBMS_XDB Resource Management Functions and Procedures

Function/Procedure	Description
`appendResourceMetadata`	Add user-defined metadata to a resource.
`createFolder`	Create a new folder resource.
`createOIDPath`	Create a virtual path to a resource, based on its object identifier (OID).
`createResource`	Create a new file resource.
`deleteResource`	Delete a resource from the repository.
`deleteResourceMetadata`	Delete specific user-defined metadata from a resource.
`existsResource`	Indicate whether or not a resource exists, given its absolute path.
`getLockToken`	Return a resource lock token for the current user, given a path to the resource.
`getResOID`	Return the object identifier (OID) of a resource, given its absolute path.
`getXDB_tablespace`	Return the current tablespace of database schema (user account) `XDB`.
`link`	Create a link to an existing resource.
`lockResource`	Obtain a WebDAV-style lock on a resource, given a path to the resource.
`moveXDB_tablespace`	Move user `XDB` to the specified tablespace.
`purgeResourceMetadata`	Delete all user-defined metadata from a resource.
`rebuildHierarchicalIndex`	Rebuild the hierarchical repository index, after import or export operations.
`renameResource`	Rename a resource.
`unlockResource`	Unlock a resource, given its lock token and path.
`updateResourceMetadata`	Modify user-defined resource metadata.

Tip: For optimal performance of queries on repository resources, gather statistics for the optimizer using procedure `DBMS_XDB_ADMIN.gatherRepositoryStats` after resource creation. You need the `XDBADMIN` role with privilege `ANALYZE ANY` or the `DBA` role to use `gatherRepositoryStats`.

See Also: *Oracle Database PL/SQL Packages and Types Reference*

The examples in this section illustrate the use of these functions and procedures.

Example 26–1 uses package `DBMS_XDB` to manage repository resources. It creates the following:

- a folder, `mydocs`, under folder `/public`
- two file resources, `emp_selby.xml` and `emp_david.xml`
- two links to the file resources, `person_selby.xml` and `person_david.xml`

It then deletes each of the newly created resources and links. The folder contents are deleted before the folder itself.

Example 26–1 Managing Resources using DBMS_XDB

```
DECLARE
  retb BOOLEAN;
BEGIN
  retb := DBMS_XDB.createfolder('/public/mydocs');
```

```
  retb := DBMS_XDB.createresource('/public/mydocs/emp_selby.xml',
                                  '<emp_name>selby</emp_name>');
  retb := DBMS_XDB.createresource('/public/mydocs/emp_david.xml',
                                  '<emp_name>david</emp_name>');
END;
/
PL/SQL procedure successfully completed.

CALL DBMS_XDB.link('/public/mydocs/emp_selby.xml',
                   '/public/mydocs',
                   'person_selby.xml');
Call completed.

CALL DBMS_XDB.link('/public/mydocs/emp_david.xml',
                   '/public/mydocs',
                   'person_david.xml');
Call completed.

CALL DBMS_XDB.deleteresource('/public/mydocs/emp_selby.xml');
Call completed.

CALL DBMS_XDB.deleteresource('/public/mydocs/person_selby.xml');
Call completed.

CALL DBMS_XDB.deleteresource('/public/mydocs/emp_david.xml');
Call completed.

CALL DBMS_XDB.deleteresource('/public/mydocs/person_david.xml');
Call completed.

CALL DBMS_XDB.deleteresource('/public/mydocs');
Call completed.
```

See Also: Chapter 29, "User-Defined Repository Metadata" for examples using `appendResourceMetadata` and `deleteResourceMetadata`

DBMS_XDB: ACL-Based Security Management

Table 26–2 lists the `DBMS_XDB` Oracle XML DB ACL- based security management functions and procedures.

Table 26–2 DBMS_XDB: Security Management Procedures and Functions

Function/Procedure	Description
`ACLCheckPrivileges`	Checks the access privileges granted to the current user by an ACL.
`changePrivileges`	Adds an ACE to a resource ACL.
`checkPrivileges`	Checks the access privileges granted to the current user for a resource.
`getACLDocument`	Retrieves the ACL document that protects a resource, given the path name of the resource.
`getPrivileges`	Returns all privileges granted to the current user for a resource.
`setACL`	Sets the ACL on a resource.

See Also:

- *Oracle Database PL/SQL Packages and Types Reference*
- *Oracle XML Developer's Kit Programmer's Guide*

The examples in this section illustrate the use of these functions and procedures.

In Example 26–2, database user HR creates two resources: a folder, /public/mydocs, with a file in it, emp_selby.xml. Procedure getACLDocument is called on the file resource, showing that the <principal> user for the document is PUBLIC.

Example 26–2 Using DBMS_XDB.GETACLDOCUMENT

```
CONNECT hr
Enter password: password

Connected.

DECLARE
  retb BOOLEAN;
BEGIN
  retb := DBMS_XDB.createFolder('/public/mydocs');
  retb := DBMS_XDB.createResource('/public/mydocs/emp_selby.xml',
                                  '<emp_name>selby</emp_name>');
END;
/
PL/SQL procedure successfully completed.

SELECT XMLSerialize(DOCUMENT
                    DBMS_XDB.getACLDocument('/public/mydocs/emp_selby.xml')
                    AS CLOB)
  FROM DUAL;

XMLSERIALIZE(DOCUMENTDBMS_XDB.GETACLDOCUMENT('/PUBLIC/MYDOCS/EMP_SELBY.XML')ASCL
--------------------------------------------------------------------------------
<acl description="Public:All privileges to PUBLIC" xmlns="http://xmlns.oracle.co
m/xdb/acl.xsd" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaL
ocation="http://xmlns.oracle.com/xdb/acl.xsd                           http://xm
lns.oracle.com/xdb/acl.xsd" shared="true">
  <ace>
    <grant>true</grant>
    <principal>PUBLIC</principal>
    <privilege>
      <all/>
    </privilege>
  </ace>
</acl>

1 row selected.
```

In Example 26–3, the system manager connects and uses procedure setACL to give the owner (database schema HR) all privileges on the file resource created in Example 26–2. Procedure getACLDocument then shows that the <principal> user is dav:owner, the owner (HR).

Example 26–3 Using DBMS_XDB.SETACL

```
CONNECT SYSTEM
Enter password: password

Connected.
```

```
-- Give all privileges to owner, HR.
CALL DBMS_XDB.setACL('/public/mydocs/emp_selby.xml',
                     '/sys/acls/all_owner_acl.xml');
Call completed.
COMMIT;
Commit complete.

SELECT XMLSerialize(DOCUMENT
                    DBMS_XDB.getACLDocument('/public/mydocs/emp_selby.xml')
                    AS CLOB)
  FROM DUAL;

XMLSERIALIZE(DOCUMENTDBMS_XDB.GETACLDOCUMENT('/PUBLIC/MYDOCS/EMP_SELBY.XML')ASCL
--------------------------------------------------------------------------------
<acl description="Private:All privileges to OWNER only and not accessible to oth
ers" xmlns="http://xmlns.oracle.com/xdb/acl.xsd" xmlns:dav="DAV:" xmlns:xsi="htt
p://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://xmlns.oracle.
com/xdb/acl.xsd http://xmlns.oracle.com/xdb/acl.xsd" shared="true">
  <ace>
    <grant>true</grant>
    <principal>dav:owner</principal>
    <privilege>
      <all/>
    </privilege>
  </ace>
</acl>

1 row selected.
```

In Example 26–4, user HR connects and uses function `changePrivileges` to add a new access control entry (ACE) to the ACL, which gives all privileges on resource `emp_selby.xml` to user `oe`. Procedure `getACLDocument` shows that the new ACE was added to the ACL.

Example 26–4 Using DBMS_XDB.CHANGEPRIVILEGES

```
CONNECT hr
Enter password: password

Connected.

SET SERVEROUTPUT ON

-- Add an ACE giving privileges to user OE
DECLARE
  r        PLS_INTEGER;
  ace      XMLType;
  ace_data VARCHAR2(2000);
BEGIN
  ace_data := '<ace xmlns="http://xmlns.oracle.com/xdb/acl.xsd"
                    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                    xsi:schemaLocation="http://xmlns.oracle.com/xdb/acl.xsd
                                        http://xmlns.oracle.com/xdb/acl.xsd
                                        DAV:http://xmlns.oracle.com/xdb/dav.xsd">
                 <principal>OE</principal>
                 <grant>true</grant>
                 <privilege><all/></privilege>
               </ace>';
  ace := XMLType.createXML(ace_data);
```

```
  r := DBMS_XDB.changePrivileges('/public/mydocs/emp_selby.xml', ace);
END;
/

PL/SQL procedure successfully completed.

COMMIT;

SELECT XMLSerialize(DOCUMENT
                    DBMS_XDB.getACLDocument('/public/mydocs/emp_selby.xml')
                    AS CLOB)
  FROM DUAL;

XMLSERIALIZE(DOCUMENTDBMS_XDB.GETACLDOCUMENT('/PUBLIC/MYDOCS/EMP_SELBY.XML')ASCL
--------------------------------------------------------------------------------
<acl description="Private:All privileges to OWNER only and not accessible to oth
ers" xmlns="http://xmlns.oracle.com/xdb/acl.xsd" xmlns:dav="DAV:" xmlns:xsi="htt
p://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://xmlns.oracle.
com/xdb/acl.xsd                          http://xmlns.oracle.com/xdb/acl.xsd" s
hared="false">
  <ace>
    <grant>true</grant>
    <principal>dav:owner</principal>
    <privilege>
      <all/>
    </privilege>
  </ace>
  <ace>
    <grant>true</grant>
    <principal>OE</principal>
    <privilege>
      <all/>
    </privilege>
  </ace>
</acl>

1 row selected.
```

In Example 26–5, user `oe` connects and calls `DBMS_XDB.getPrivileges`, which shows all of the privileges granted to user `oe` on resource `emp_selby.xml`.

Example 26–5 Using DBMS_XDB.GETPRIVILEGES

```
CONNECT oe
Enter password: password

Connected.

SELECT XMLSerialize(DOCUMENT
                    DBMS_XDB.getPrivileges('/public/mydocs/emp_selby.xml')
                    AS CLOB)
  FROM DUAL;

XMLSERIALIZE(DOCUMENTDBMS_XDB.GETPRIVILEGES('/PUBLIC/MYDOCS/EMP_SELBY.XML')ASCLO
--------------------------------------------------------------------------------
<privilege xmlns="http://xmlns.oracle.com/xdb/acl.xsd" xmlns:xsi="http://www.w3.
org/2001/XMLSchema-instance" xsi:schemaLocation="http://xmlns.oracle.com/xdb/acl
.xsd http://xmlns.oracle.com/xdb/acl.xsd DAV: http://xmlns.oracle.com/xdb/dav.xs
d" xmlns:xdbacl="http://xmlns.oracle.com/xdb/acl.xsd" xmlns:dav="DAV:">
  <read-properties/>
```

```
  <read-contents/>
  <write-config/>
  <link/>
  <unlink/>
  <read-acl/>
  <write-acl-ref/>
  <update-acl/>
  <resolve/>
  <link-to/>
  <unlink-from/>
  <dav:lock/>
  <dav:unlock/>
  <dav:write-properties/>
  <dav:write-content/>
  <dav:execute/>
  <dav:take-ownership/>
  <dav:read-current-user-privilege-set/>
</privilege>

1 row selected.
```

DBMS_XDB: Configuration Management

Table 26–3 lists the DBMS_XDB Oracle XML DB configuration management functions and procedures.

Table 26–3 DBMS_XDB: Configuration Management Functions and Procedures

Function/Procedure	Description
cfg_get	Returns the configuration information for the current session.
cfg_refresh	Refreshes the session configuration information using the current Oracle XML DB configuration file, xdbconfig.xml.
cfg_update	Updates the Oracle XML DB configuration information. This writes the configuration file, xdbconfig.xml.
getFTPPort	Returns the current FTP port number.
getHTTPPort	Returns the current HTTP port number.
setFTPPort	Sets the Oracle XML DB FTP port to the specified port number.
setHTTPPort	Sets the Oracle XML DB HTTP port to the specified port number.

See Also: *Oracle Database PL/SQL Packages and Types Reference*

The examples in this section illustrate the use of these functions and procedures.

Example 26–6 uses function cfg_get to retrieve the Oracle XML DB configuration file, xdbconfig.xml.

Example 26–6 Using DBMS_XDB.CFG_GET

```
CONNECT SYSTEM
Enter password: password

Connected.

SELECT DBMS_XDB.cfg_get() FROM DUAL;
```

```
DBMS_XDB.CFG_GET()
--------------------------------------------------------------------------------
<xdbconfig xmlns="http://xmlns.oracle.com/xdb/xdbconfig.xsd" xmlns:xsi="http://w
ww.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://xmlns.oracle.com/x
db/xdbconfig.xsd                                      http://xmlns.oracle.com/xdb
/xdbconfig.xsd">
  <sysconfig>
    <acl-max-age>19</acl-max-age>
    <acl-cache-size>32</acl-cache-size>
    <invalid-pathname-chars/>
    <case-sensitive>true</case-sensitive>
    <call-timeout>6000</call-timeout>
    <max-link-queue>65536</max-link-queue>
    <max-session-use>100</max-session-use>
    <persistent-sessions>false</persistent-sessions>
    <default-lock-timeout>3600</default-lock-timeout>
    <xdbcore-logfile-path>/sys/log/xdblog.xml</xdbcore-logfile-path>
    <xdbcore-log-level>0</xdbcore-log-level>
    <resource-view-cache-size>1048576</resource-view-cache-size>
    <protocolconfig>
      <common>
        . . .
      </common>
      <ftpconfig>
        . . .
      </ftpconfig>
      <httpconfig>
        <http-port>0</http-port>
        <http-listener>local_listener</http-listener>
        <http-protocol>tcp</http-protocol>
        <max-http-headers>64</max-http-headers>
        <max-header-size>16384</max-header-size>
        <max-request-body>2000000000</max-request-body>
        <session-timeout>6000</session-timeout>
        <server-name>XDB HTTP Server</server-name>
        <logfile-path>/sys/log/httplog.xml</logfile-path>
        <log-level>0</log-level>
        <servlet-realm>Basic realm=&quot;XDB&quot;</servlet-realm>
        <webappconfig>
        . . .
        </webappconfig>
        <authentication>
        . . .
        </authentication>
    </protocolconfig>
    <xdbcore-xobmem-bound>1024</xdbcore-xobmem-bound>
    <xdbcore-loadableunit-size>16</xdbcore-loadableunit-size>
    <acl-evaluation-method>ace-order</acl-evaluation-method>
  </sysconfig>
</xdbconfig>

1 row selected.
```

Example 26–7 illustrates the use of procedure `cfg_update`. The current configuration is retrieved as an `XMLType` instance and modified. It is then rewritten using `cfg_update`.

Example 26–7 Using DBMS_XDB.CFG_UPDATE

```
DECLARE
```

```
  configxml    SYS.XMLType;
  configxml2   SYS.XMLType;
BEGIN
  -- Get the current configuration
  configxml := DBMS_XDB.cfg_get();

  -- Modify the configuration
  SELECT updateXML(
           configxml,
           '/xdbconfig/sysconfig/protocolconfig/httpconfig/http-port/text()',
           '8000',
           'xmlns="http://xmlns.oracle.com/xdb/xdbconfig.xsd"')
    INTO configxml2 FROM DUAL;

  -- Update the configuration to use the modified version
  DBMS_XDB.cfg_update(configxml2);
END;
/

PL/SQL procedure successfully completed.

SELECT DBMS_XDB.cfg_get() FROM DUAL;

DBMS_XDB.CFG_GET()
--------------------------------------------------------------------------------
<xdbconfig xmlns="http://xmlns.oracle.com/xdb/xdbconfig.xsd" xmlns:xsi="http://w
ww.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://xmlns.oracle.com/x
db/xdbconfig.xsd http://xmlns.oracle.com/xdb/xdbconfig.xsd">
  <sysconfig>
    <acl-max-age>15</acl-max-age>
    <acl-cache-size>32</acl-cache-size>
    <invalid-pathname-chars/>
    <case-sensitive>true</case-sensitive>
    <call-timeout>6000</call-timeout>
    <max-link-queue>65536</max-link-queue>
    <max-session-use>100</max-session-use>
    <persistent-sessions>false</persistent-sessions>
    <default-lock-timeout>3600</default-lock-timeout>
    <xdbcore-logfile-path>/sys/log/xdblog.xml</xdbcore-logfile-path>
    <resource-view-cache-size>1048576</resource-view-cache-size>
    <protocolconfig>
      <common>
      . . .
      </common>
      <ftpconfig>
      . . .
      </ftpconfig>
      <httpconfig>
        <http-port>8000</http-port>
        . . .
      </httpconfig>
    </protocolconfig>
    <xdbcore-xobmem-bound>1024</xdbcore-xobmem-bound>
    <xdbcore-loadableunit-size>16</xdbcore-loadableunit-size>
    <acl-evaluation-method>ace-order</acl-evaluation-method>
</xdbconfig>

1 row selected.
```

27

Repository Access Control

Oracle Database provides classic database security such as row-level and column-level secure access by database users. It also provides fine-grained access control for resources in Oracle XML DB Repository. This chapter describes the latter. It includes how to create, set, and modify access control lists (ACLs) and how ACL security interacts with other Oracle Database security mechanisms.

This chapter contains these topics:

- Access Control Concepts
- Database Privileges for Repository Operations
- Privileges
- ACLs and ACEs
- Working with Access Control Lists (ACLs)
- ACL Caching
- Repository Resources and Database Table Security
- Integrating Oracle XML DB with LDAP

See Also:

- Chapter 28, "Accessing the Repository using Protocols" for more information about WebDAV
- Chapter 34, "Administering Oracle XML DB" for information about configuring and administering resource security
- "APIs for XML" on page 1-4 for information about the PL/SQL APIs you can use to manage resource security

Access Control Concepts

This section describes several access control terms and concepts. Each of the entities described here, user, role, principal, privilege, access control list (ACL), and access control entry (ACE), is implemented declaratively as an XML document or fragment.

Secure authorization requires defining which users, applications, or functions can have access to which data, to perform which kinds of operations. There are thus three dimensions: (1) *which users* can (2) perform *which operations* (3) on *which data*. We speak of (1) principals, (2) privileges, and (3) objects, corresponding to these three dimensions, respectively. Principals are users or roles.

Principals and privileges (dimensions 1 and 2) are related in a declarative way by defining *access control lists*. These are then related to the third dimension, data, in various ways, either declaratively or procedurally. For example, you can protect an Oracle XML DB Repository resource or table data by using PL/SQL procedure `DBMS_XDB.setACL` to set its controlling ACL.

Principal: A User or Role

In the context of fine-grained database access control, a **principal** is a user or a role. A **user** can be any person or application that accesses information in the database. A **role** is composed of users and possibly other roles, but this recursion cannot be circular. Ultimately, each role, and thus each principal, corresponds to a *set of users*.

A user is represented for access control purposes by an XML fragment with element `user`. A role is represented by a fragment with element `role`.

Oracle Database supports the following as principals:

- Database users and **database roles**. A **database user** is also sometimes referred to as a database **schema** or a user **account**. When a person or application logs onto the database, it uses a database user (schema) and password. A **database role** corresponds to a set of database privileges that can be granted to database users, applications, or other database roles—see "Database Roles Map Database Privileges to Users" on page 27-2.
- LDAP users and groups of LDAP users. For details on using LDAP principals see "Integrating Oracle XML DB with LDAP" on page 27-20.

When a term such as "user" or "role" is used here without qualification, it applies to each type of user or role. When it is important to distinguish the type, the qualifier "database" or "LDAP" is used.

Database Roles Map Database Privileges to Users

A database role is *granted* privileges, just as a database user can be granted privileges. A database role serves as an intermediary for mapping database privileges to database users (and applications): a role is granted privileges, and the role is then granted to users (giving them the privileges). The line between a group of users and a group of privileges that are granted to those users is blurred a bit in the concept of database role: the role can serve to group the privileges that are mapped to the users and to group the users to which the privileges are mapped. The mapping is done by defining the role and granting it to users, and traditional database terminology considers the role to be the same thing as the set of privileges that are granted to it.

In the context of fine-grained access control, a different mechanism, an *access control list* (ACL), is used as the intermediary that maps privileges to users. A role is simply a set of users. In this context, the act of associating privileges with users and with roles is not a database grant. It is a declarative ACL entry, together with a run-time evaluation of ACLs and resolution of ACL conflicts.

Please keep this terminology difference in mind, to avoid confusion. As a means of mapping privileges to users, a database role combines some of the functionality that in an access-control context is divided into (1) principals, (2) privileges, and (3) ACLs. In access control terminology, roles are classified with users as principals. In traditional database terminology, roles are instead classified as sets of privileges.

Principal DAV::owner

You can use principal `DAV::owner` in connection with a given Oracle XML DB Repository resource to refer to the resource owner. The owner of a resource is one of

the properties of the resource. You can use principal `DAV::owner` to facilitate ACL sharing among principals, because the owner of a resource often has special rights.

Privilege: A Permission

A **privilege** is a particular right or permission that can be granted or denied to a principal. A privilege is aggregate or atomic:

- **Aggregate** privilege – A privilege that includes other privileges.
- **Atomic** privilege – A privilege that does not include other privileges. It cannot be subdivided.

Aggregate privileges simplify usability when the number of privileges becomes large, and they promote interoperability between ACL clients. See "Privileges" on page 27-5.

Aggregate privileges retain their identity: they are not decomposed into the corresponding atomic (leaf) privileges. In WebDAV terms, Oracle Database aggregate privileges are not abstract. This implies that an aggregate privilege acts as a set of pointers to its component privileges, rather than a copy of those components. Thus, an aggregate privilege is always up to date, even if the definition of a component changes.

The set of privileges granted to a principal controls whether that principal can perform a given operation on the data that it protects. For example, if the principal (database user) `HR` wants to perform the `read` operation on a given resource, then read privileges must be granted to principal `HR` prior to the read operation.

Access Control Entry (ACE)

An **access control entry** (**ACE**) is an XML element (`ace`) that is an entry in an access control list (ACL). An ACE either grants or denies access to some repository resource or other database data by a particular principal (user or role). The ACE does not, itself, specify which data to protect. That is done outside the ACE and the ACL, by associating the ACL with target data. One way to make that association is by using PL/SQL procedure `DBMS_XDB.setACL`.

See "ACL and ACE Evaluation" on page 27-8.

An Oracle XML DB ACE either grants or denies privileges for a principal. An **`ace`** element has the following:

- Operation `grant`: either `true` (to grant) or `false` (to deny) access.
- Either a valid principal (element `principal`) or a completed list of principals (element `invert`).
- Privileges: A set of privileges to be granted or denied for a particular principal (element `privilege`).
- Principal format (optional): The format of the principal. An LDAP distinguished name (DN), a short name (database user/role or LDAP nickname), or an LDAP GUID. The default value is `short name`. If the principal name matches both a database user and an LDAP nickname, it is assumed to refer to the LDAP nickname.
- Collection (optional): A `BOOLEAN` attribute that specifies whether the principal is a collection of users (LDAP group or database role) or a single user (LDAP or database user).
- Start and end date (optional): Attributes that define the time period over which an ACE is valid. See "ACE Validity Time Period" on page 27-10.

Example 27–1 shows a simple ACE that grants privilege `DAV::all` to principal `DAV::owner`. It thus grants all privileges to the owner of the resource to which its ACL applies.

Example 27–1 Simple Access Control Entry (ACE) that Grants a Privilege

```
<ace>
  <grant>true</grant>
  <principal>DAV::owner</principal>
  <privilege>
    <DAV::all/>
  </privilege>
</ace>
```

Access Control List (ACL)

An **access control list** (**ACL**) is a list of access control entries (ACEs). By default, order in the list is relevant (see "ACL and ACE Evaluation" on page 27-8). Example 27–2 shows a simple ACL that contains only the ACE of Example 27–1.

Example 27–2 Simple Access Control List (ACL) that Grants a Privilege

```
<acl description="myacl"
     xmlns="http://xmlns.oracle.com/xdb/acl.xsd"
     xmlns:dav="DAV:"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://xmlns.oracle.com/xdb/acl.xsd
                         http://xmlns.oracle.com/xdb/acl.xsd">
  <ace>
    <grant>true</grant>
    <principal>dav:owner</principal>
    <privilege>
      <dav:all/>
    </privilege>
  </ace>
</acl>
```

Database Privileges for Repository Operations

Table 27–1 shows the database privileges required for some common operations on resources in Oracle XML DB Repository. In addition to the privileges listed in column **Privileges Required** you must have the `resolve` privilege for the folder containing the resource and for all of its parent folders, up to the root folder.

Table 27–1 Database Privileges Needed for Operations on Oracle XML DB Resources

Operation	Description	Privileges Required
`CREATE`	Create a new resource in folder F	`update` and `link` on folder F
`DELETE`	Delete resource R from folder F	`update` and `unlink-from` on R, `update` and `unlink` on folder F
`UPDATE`	Update the contents or properties of resources R	`update` on R

Table 27–1 (Cont.) Database Privileges Needed for Operations on Oracle XML DB Resources

Operation	Description	Privileges Required
GET	An FTP or HTTP(S) retrieval of resource R	read-properties, read-contents on R
SET_ACL	Set the ACL of a resource R	DAV::write-acl on R
LIST	List the resources in folder F	read-properties on folder F, read-properties on resources in folder F. Only those resources on which the user has read-properties privilege are listed.

See Also: "Upgrading an Existing Oracle XML DB Installation" on page 34-3 for information about treatment of database access privileges when upgrading

Privileges

This section describes the privileges that are provided with Oracle Database. These include the standard WebDAV privileges, which use the WebDAV namespace **DAV:**[1], and Oracle-specific privileges, which use the Oracle XML DB ACL namespace, http://xmlns.oracle.com/xdb/acl.xsd, which has the predefined prefix **xdb**.

See Also: RFC 3744: "Web Distributed Authoring and Versioning (WebDAV) Access Control Protocol", IETF Network Working Group Request For Comments #3744, May 2004

Atomic Privileges

Table 27–2 lists the atomic privileges.

Table 27–2 Atomic Privileges

Atomic Privilege	Description	Database Counterpart
DAV::lock	Lock a resource using WebDAV locks.	UPDATE
DAV::read-current-user-privilege-set	Access the DAV::current-user-privilege-set property of a resource.	N/A
DAV::take-ownership	Take ownership of a resource.	N/A
DAV::unlock	Unlock a resource locked using a WebDAV lock.	UPDATE
DAV::write-content	Modify the content of a resource.	UPDATE
DAV::write-properties	Modify the properties of a resource. Lock or unlock a resource. Modifiable properties include Author, DisplayName, Language, CharacterSet, ContentType, SBResExtra, Owner, OwnerID, CreationDate, Modification Date, ACL, ACLOID, Lock, and Locktoken.	UPDATE
xdb:link	Allow creation of links from a resource.	INSERT
xdb:link-to	Allow creation of links to a resource.	N/A
xdb:read-acl	Read the ACL of a resource.	SELECT

[1] Note the colon (:) as part of the namespace name. DAV: is the namespace itself, *not* a prefix. A prefix commonly used for namespace DAV: is dav, but this is only conventional. dav is not a predefined prefix for Oracle XML DB.

Table 27–2 (Cont.) Atomic Privileges

Atomic Privilege	Description	Database Counterpart
`xdb:read-contents`	Read the contents of a resource.	`SELECT`
`xdb:read-properties`	Read the properties of a resource.	`SELECT`
`xdb:resolve`	Traverse a folder (for folders only).	`SELECT`
`xdb:unlink`	Allow deletion of links from a resource.	`DELETE`
`xdb:unlink-from`	Allow deletion of links to a resource.	N/A
`xdb:update-acl`	Change the contents of the resource ACL.	`UPDATE`
`xdb:write-acl-ref`	Change the ACLOID of a resource.	`UPDATE`

Aggregate Privileges

Table 27–3 lists the aggregate privileges and the atomic privileges of which each is composed.

Table 27–3 Aggregate Privileges

Aggregate Privilege	Component Atomic Privileges
`DAV::all`	All atomic DAV privileges.
`xdb:all`	All atomic DAV privileges plus `xdb:link-to`.
`DAV::bind`	`xdb:link`
`DAV::unbind`	`xdb:unlink`
`DAV::read`	`xdb:read-properties`, `xdb:read-contents`, `xdb:resolve`
`DAV::read-acl`	`xdb:read-acl`
`DAV::write`	`DAV::write-content`, `DAV::write-properties`, `xdb:link`, `xdb:unlink`, `xdb:unlink-from`
`DAV::write-acl`	`xdb:write-acl-ref`, `xdb:update-acl`
`DAV::update`	`DAV::write-content`, `DAV::write-properties`
`xdb:update`	`DAV::write-properties`, `DAV::write-content`

ACLs and ACEs

An access control list (ACL) is a standard security mechanism that is used in some languages, such as Java, and some operating systems, such as Microsoft Windows. ACLs are also a part of the WebDAV standard. ACLs are used to protect resources, which in the case of Oracle Database can be either resources (files and folders) in Oracle XML DB Repository.

Repository resources can be accessed using WebDAV, and their protecting ACLs act as WebDAV ACLs. Each repository resource is protected by some ACL. ACLs that protect a resource are enforced no matter how the resource is accessed, whether by WebDAV, SQL, or any other way.

When a new resource is created in Oracle XML DB Repository, by default the ACL on its parent folder is used to protect the resource. After the resource is created, a new ACL can be set on it.

ACLs in Oracle Database are XML documents that are validated against the Oracle Database ACL XML schema, which is located in Oracle XML DB Repository at `/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/acl.xsd`. ACLs are themselves stored and managed as resources in the repository.

Before a principal performs an operation on ACL-protected data, the user privileges for the protected data are checked. The set of privileges checked depends on the operation to be performed.

Aggregate privileges are composed of other privileges. When an ACL is stored, the aggregate privileges it refers to act as sets of pointers to their component privileges.

All ACLs are stored in table **`XDB$ACL`**, which is owned by database user `XDB`. This is an XML schema-based `XMLType` table. Each row in this table (and therefore each ACL) has a system-generated object identifier (**OID**) that can be accessed as a column named **`OBJECT_ID`**.

Each Oracle XML DB Repository resource has a property named **`ACLOID`**. The `ACLOID` stores the OID of the ACL that protects the resource. An ACL is itself a resource, and the `XMLRef` property of an ACL, for example, `/sys/acls/all_all_acl.xml`, is a `REF` to the row in table `XDB$ACL` that contains the content of the ACL. These two properties form the link between table `XDB$RESOURCE`, which stores Oracle XML DB resources, and table `XDB$ACL`.

See Also:

- Appendix A, "Oracle-Supplied XML Schemas and Examples" for the ACL XML schema
- *Oracle Database 2 Day + Security Guide* for information about database schema `XDB`

System ACLs

Some ACLs are predefined and supplied with Oracle Database. They are referred to as **system ACLs**.

There is only one ACL that is self-protected, that is, protected by its own contents. It is the **bootstrap ACL**, a system ACL that is located in Oracle XML DB Repository at `/sys/acls/bootstrap_acl.xml`. The bootstrap ACL grants `READ` privilege to all users. It also grants `FULL ACCESS` to database roles `XDBADMIN` (the Oracle XML DB administrator) and `DBA`. Database role `XDBADMIN` is particularly useful for users who must register global XML schemas.

Other system ACLs include the following. Each is protected by the bootstrap ACL.

- **`all_all_acl.xml`** – Grants all privileges to all users.
- **`all_owner_acl.xml`** – Grants all privileges to the owner of the resource.
- **`ro_all_acl.xml`** – Grants read privileges to all users.

System ACLs use the file-naming convention `<privilege>_<users>_acl.xml`, where `<privilege>` represents the privilege granted, and `<users>` represents the users that are granted access to the resource. When you define your own ACLs, you can use any names you like.

See Also: "Local and Global XML Schemas" on page 7-14

ACL and ACE Evaluation

Privileges are checked before a principal is allowed to access a repository resource that is protected by one or more ACLs. This check is done by evaluating the protecting ACLs for that principal, in order. For each such ACL, the ACEs in it that apply to the principal are examined, in order.

If one ACE grants a certain privilege to the current user and another ACE denies that privilege to the user, then a conflict arises. There are two possible ways to manage conflicts among ACEs for the same principal.

- The default behavior, termed **`ace-order`**, is to use only the *first* ACE that occurs for a given principal. Additional ACEs for that principal have no effect. In this case, *ACE order is relevant*.
- You can, however, configure the database to use an alternate behavior, **`deny-trumps-grant`**. In this case, any ACE with child `deny` for a given principal denies permission to that principal, whether or not there are other ACEs for that principal that have a `grant` child. In this case, `deny` always takes precedence over `grant`, and *ACE order is irrelevant*.

You can configure ACL evaluation behavior by setting configuration parameter **`acl-evaluation-method`**, in configuration file `xdbconfig.xml`, to either `ace-order` or `deny-trumps-grant`. The default configuration file specifies `ace-method`, but the default value for element `acl-evaluation-method`, used when no method is given, is `deny-trumps-grant`.

Note: In releases prior to Oracle Database 11*g* Release 1, only one ACL evaluation behavior was available: `deny-trumps-grant` (though it was not specified in the configuration file).

The change to use `ace-order` as the default behavior has important consequences for upgrading and downgrading between database versions. See "Upgrading an Existing Oracle XML DB Installation" on page 34-3.

ACL Validation

When an ACL is created, it is validated against the XML schema for ACLs, and some correctness tests are run, such as ensuring that start and end dates for ACEs are in chronological order. There is no complete check at ACL creation time of relations among ACLs. For example, no check is made for the existence and correctness of a parent ACL. (See "ACL Inheritance" on page 27-9.)

Such a complete check of ACL correctness is called **ACL validity** checking, but it is not to be confused with its XML *schema* validity. For an ACL to be valid (as an ACL), it must also be XML schema-valid, but the converse does not hold.

A full ACL validity check is made at run time, whenever an ACL is evaluated to check whether a principal has the proper privileges for some operation. If this check finds that the ACL is invalid, then all privileges that the ACL would grant are *denied* to the specified principals.

ACL validity can also be checked independently of its run-time use to check privileges, by invoking PL/SQL procedure `DBMS_XDBZ.validateACL`. You can do this ahead of time, to avoid run-time errors or privilege denial due to ACL invalidity.

ACL Inheritance

An ACL can inherit grants, that is, associations of principals with privileges, from another ACL. Inheritance provides flexibility of definition and promotes reuse of access control policies.

Grants are defined in ACEs, so inheritance of grants involves traversing ACL inheritance chains and the associated ACES. But ACL inheritance, and therefore grant inheritance, can be *recursive*. If ACL *A1* inherits from ACL *A2*, a grant defined by *A1* is not necessarily present in an ACE of *A2*. It might instead be in an ACE in ACL *A3*, where *A2* inherits (directly or indirectly) from *A3*.

ACL inheritance is simple, not multiple, inheritance: an ACL inherits from at most one other ACL. Cycles are not permitted in an inheritance chain: an ACL that inherits directly or indirectly from itself is invalid. An ACL that inherits from an ACL that does not exist is also invalid.

The grants **declared in** an ACL are those explicitly defined by its ACES. The grants **defined for** an ACL are those defined by its ACES plus those inherited by it.

There are two kinds of ACL inheritance, extending inheritance and constraining inheritance, specified using element `extends-from` or element `constrained-with`, respectively. Both elements reference the ACL being inherited from. An ACL can have at most one `extends-from` or `constrained-with` element. Example 27–3 shows an `extends-from` element. Example 27–4 shows a `constrained-with` element.

Example 27–3 Element extends-from

```
<extends-from type="simple" href="/sys/acls/extend_acl.xml"/>
```

Example 27–4 Element constrained-with

```
<constrained-with type="simple" href="/sys/acls/constrain_acl.xml"/>
```

Extending inheritance extends the grants that are declared in the inheriting ACL (the **child ACL**) by some grants that are defined for the ACL it is inheriting from (the **parent ACL**). For example, if ACL *A1* declares that it extends from ACL *A2*, then *A1* can include grants defined for *A2*.

Constraining inheritance restricts the grants that are declared in the inheriting ACL to grants that are also defined for the ACL it is inheriting from. For example, if ACL *A1* declares that it inherits from ACL *A2* by constraining, then all grants defined for *A1* must also be defined for *A2*.

Extending inheritance is a set *union* operation, and constraining inheritance is a set *intersection* operation. When ACL A1 extends from ACL A2, the grants in both can be *combined* to determine whether a given principal is granted a given privilege. When ACL *A1* is constrained with ACL *A2*, only the grants that are *common* to both *A1* and *A2* are used to determine a grant.

More precisely, when ACL *A1* inherits from ACL *A2*, and *A1* is checked to see if a given principal has been granted a given set of privileges, determination proceeds as follows:

- If *A1* extends from *A2* – The ACEs that are declared in *A1* are examined first. If they do not grant all or deny any of the privileges in question to the principal, then the ACEs defined in the `extending-from` parent of *A1* are examined. This in turn means that if the ACEs that are explicitly declared in *A2* do not grant all or deny any of the privileges in question to the principal, and if *A2* extends from *A3*, then *A3* is examined. And so on.

- If *A1* is constrained with *A2* – The ACEs that are explicitly declared in *A1* and those defined for *A2* are each examined separately to ensure that they *both* grant all of the privileges in question to the specified principal. The check for *A2* proceeds the same way if it is constrained by ACL *A3*, and so on.

Put another way, extending inheritance accumulates granted privileges and constraining inheritance accumulates denied privileges. In extending inheritance, if either the child or the parent ACL grants a privilege to the principal, then the privilege is granted. In constraining inheritance, if either the child or the parent ACL denies a privilege, then it is denied.

> **See Also:** Example 27–5 on page 27-10 for an ACL that uses extending inheritance

Complementing the Principals in an ACE: Element invert

It is sometimes more convenient to define a set of principals by complementing another set of principals—that is the purpose of ACE element `invert`. Instead of listing each of the principals that you want to include, wrap the list of principals that you want to exclude with element `invert`.

In Example 27–5, the first ACE denies privilege `privilege1` to all principals except `IntranetUsers`. Because (by default) ACEs are considered in the order they appear, all subsequent ACEs are overridden by the first ACE, so principal `NonIntraNetUser` is denied privilege `privilege1` in spite of the explicit grant.

Example 27–5 Complementing a Set of Principals with Element invert

```
<acl description="invert ACL"
     xmlns="http://xmlns.oracle.com/xdb/acl.xsd"
     xmlns:dav="DAV:"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://xmlns.oracle.com/xdb/acl.xsd
                         http://xmlns.oracle.com/xdb/acl.xsd">
  <extends-from type="simple" href="/sys/acls/parent_acl.xml"/>
  <ace>
    <grant>false</grant>
    <invert><principal>dav:owner</principal></invert>
    <privilege><read-contents/></privilege>
  </ace>
  <ace>
    <grant>true</grant>
    <principal>GERONIMO</principal>
    <privilege><read-contents/></privilege>
  </ace>
</acl>
```

ACE Validity Time Period

You can use optional attributes `start_date` and `end_date` (of XML Schema type `dateTime`) to define the time period over which an ACE is valid. If **`start_date`** is specified, then the ACE is valid on and after that date. If **`end_date`** is specified, then the ACS is invalid after that date. The `end_date` value must follow the `start_date` chronologically or else be the same value. Otherwise, the ACE and its ACL are invalid. If no time zone is specified in an XML Schema `dateTime` value, then GMT (UTC) is assumed. Example 27–6 shows an ACE with start and end dates.

Example 27–6 ACE with Start and End Dates

```
<ace start_date="2008-02-12T00:00:00Z" end_date="2008-12-31T00:00:00Z">
  <grant>true</grant>
  <principal>GERONIMO</principal>
  <privilege><read-contents/></privilege>
</ace>
```

Working with Access Control Lists (ACLs)

Oracle Database access control lists (ACLs) are themselves (file) resources in Oracle XML DB Repository, so all of the access methods that operate on repository resources also apply to ACLs. In addition, there are several APIs specific to ACLs in package `DBMS_XDB`. Those procedures and functions let you use PL/SQL to access Oracle XML DB security mechanisms, check user privileges based on a particular ACL, and list the set of privileges the current user has for a particular ACL and resource.

See Also: Chapter 34, "Administering Oracle XML DB"

Creating an ACL using DBMS_XDB.CREATERESOURCE

Example 27–7 creates an ACL as file resource `/TESTUSER/acl1.xml`. If applied to a resource, this ACL grants all privileges to the owner of the resource.

Example 27–7 Creating an ACL using CREATERESOURCE

```
DECLARE
  b BOOLEAN;
BEGIN
  b := DBMS_XDB.createFolder('/TESTUSER');
  b := DBMS_XDB.createResource(
         '/TESTUSER/acl1.xml',
         '<acl description="myacl"
               xmlns="http://xmlns.oracle.com/xdb/acl.xsd"
               xmlns:dav="DAV:"
               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xsi:schemaLocation="http://xmlns.oracle.com/xdb/acl.xsd
                                   http://xmlns.oracle.com/xdb/acl.xsd">
            <ace>
              <grant>true</grant>
              <principal>dav:owner</principal>
              <privilege>
                <dav:all/>
              </privilege>
            </ace>
          </acl>',
         'http://xmlns.oracle.com/xdb/acl.xsd',
         'acl');
END;
```

Note: Before performing any operation that uses an ACL file resource that was created during the current transaction, you must perform a `COMMIT` operation. Until you do that, an ORA-22881 "dangling REF" error is raised whenever you use the ACL file.

Retrieving an ACL Document, Given its Repository Path

Example 27–8 shows how to retrieve an ACL document, given its location in Oracle XML DB Repository.

Example 27–8 Retrieving an ACL Document, Given its Repository Path

```
SELECT a.OBJECT_VALUE FROM RESOURCE_VIEW rv, XDB.XDB$ACL a
  WHERE ref(a)
        = XMLCast(XMLQuery('declare default element namespace
                            "http://xmlns.oracle.com/xdb/XDBResource.xsd"; (: :)
                            fn:data(/Resource/XMLRef)'
                           PASSING rv.RES RETURNING CONTENT)
                  AS REF XMLType)
    AND equals_path(rv.RES, '/TESTUSER/acl1.xml') = 1;

OBJECT_VALUE
--------------------------------------------------------------------------------
<acl description="myacl" xmlns="http://xmlns.oracle.com/xdb/acl.xsd" xmlns:dav="
DAV:" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="
http://xmlns.oracle.com/xdb/acl.xsd                                    http://xm
lns.oracle.com/xdb/acl.xsd" shared="true">
  <ace>
    <grant>true</grant>
    <principal>dav:owner</principal>
    <privilege>
      <dav:all/>
    </privilege>
  </ace>
</acl>
```

Setting the ACL of a Resource

Example 27–9 creates resource `/TESTUSER/po1.xml` and sets its ACL to `/TESTUSER/acl1.xml` using PL/SQL procedure `DBMS_XDB.setACL`.

Example 27–9 Setting the ACL of a Resource

```
DECLARE
  b BOOLEAN;
BEGIN
  b := DBMS_XDB.createResource('/TESTUSER/po1.xml', 'Hello');
END;
/

CALL DBMS_XDB.setACL('/TESTUSER/po1.xml', '/TESTUSER/acl1.xml');
```

Deleting an ACL

Example 27–10 illustrates how to delete an ACL using procedure `DBMS_XDB.deleteResource`. It deletes the ACL created in Example 27–7.

Example 27–10 Deleting an ACL

```
CALL DBMS_XDB.deleteResource('/TESTUSER/acl1.xml');
```

If a resource is being protected by an ACL that you want to delete, change the ACL of that resource before deleting the ACL.

Updating an ACL

You can update an ACL using any of the standard ways of updating resources. In particular, since an ACL is an XML document, you can use Oracle SQL function `updateXML` and related XML-updating functions to manipulate ACLs. You must `COMMIT` after making any ACL changes.

Oracle XML DB ACLs are *cached*, for fast evaluation. When a transaction that updates an ACL is committed, the modified ACL is picked up by existing database sessions, after the timeout specified in the Oracle XML DB configuration file, `/xdbconfig.xml`. The XPath location for this timeout parameter is `/xdbconfig/sysconfig/acl-max-age`. The value is expressed in seconds. Sessions initiated after the ACL is modified use the new ACL without any delay.

If an ACL resource is updated with non-ACL content, the same rules apply as for deletion. Thus, if any resource is being protected by an ACL that is being updated, you must first change the ACL.

> **See Also:** "Updating XML Data" on page 4-10 for information about the Oracle SQL functions used here to update XML data

You can use FTP or WebDAV to update an ACL. For more details on how to use these protocols, see Chapter 28, "Accessing the Repository using Protocols". You can update an ACL or an access control entry (ACE) using `RESOURCE_VIEW`.

Example 27–11 uses Oracle SQL function `updateXML` to update the ACL `/TESTUSER/acl1.xml` by replacing it entirely. The effect is to replace the principal value `DAV::owner` by `TESTUSER`, because the rest of the replacement ACL is the same as it was before.

Example 27–11 Updating (Replacing) an Access Control List

```
UPDATE RESOURCE_VIEW r
  SET r.RES =
        updateXML(
          r.RES,
          '/r:Resource/r:Contents/a:acl',
          '<acl description="myacl"
                xmlns="http://xmlns.oracle.com/xdb/acl.xsd"
                xmlns:dav="DAV:"
                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                xsi:schemaLocation="http://xmlns.oracle.com/xdb/acl.xsd
                                    http://xmlns.oracle.com/xdb/acl.xsd">
             <ace>
               <grant>true</grant>
               <principal>TESTUSER</principal>
               <privilege>
                 <dav:all/>
               </privilege>
             </ace>
           </acl>',
          'xmlns:r="http://xmlns.oracle.com/xdb/XDBResource.xsd"
           xmlns:a="http://xmlns.oracle.com/xdb/acl.xsd"')
  WHERE equals_path(r.RES, '/TESTUSER/acl1.xml') = 1;
```

Example 27–12 uses Oracle SQL function `appendChildXML` to append an ACE to an existing ACL. The ACE gives privileges `read-properties` and `read-contents` to user `HR`.

Example 27–12 Appending ACEs to an Access Control List

```
UPDATE RESOURCE_VIEW r
  SET r.RES =
        appendChildXML(
          r.RES,
          '/r:Resource/r:Contents/a:acl',
          XMLType('<ace xmlns="http://xmlns.oracle.com/xdb/acl.xsd">
                     <grant>true</grant>
                     <principal>HR</principal>
                     <privilege>
                       <read-properties/>
                       <read-contents/>
                     </privilege>
                   </ace>'),
                  'xmlns:r="http://xmlns.oracle.com/xdb/XDBResource.xsd"
                   xmlns:a="http://xmlns.oracle.com/xdb/acl.xsd"')
  WHERE equals_path(r.RES, '/TESTUSER/acl1.xml') = 1;
```

Example 27–13 uses Oracle SQL function `deleteXML` to delete an ACE from an ACL. The first ACE is deleted.

Example 27–13 Deleting an ACE from an Access Control List

```
UPDATE RESOURCE_VIEW r
  SET r.RES =
        deleteXML(r.RES,
                  '/r:Resource/r:Contents/a:acl/a:ace[1]',
                  'xmlns:r="http://xmlns.oracle.com/xdb/XDBResource.xsd"
                   xmlns:a="http://xmlns.oracle.com/xdb/acl.xsd"')
  WHERE equals_path(r.RES, '/TESTUSER/acl1.xml') = 1;
```

Retrieving the ACL Document that Protects a Given Resource

Example 26–2 illustrates how to use function `DBMS_XDB.getACLDocument` to retrieve the ACL document that protects a given resource.

Example 27–14 Retrieving the ACL Document for a Resource

```
SELECT XMLSerialize(DOCUMENT DBMS_XDB.getACLDocument('/TESTUSER/po1.xml')
                    AS CLOB)
  FROM DUAL;

XMLSERIALIZE(DOCUMENTDBMS_XDB.GETACLDOCUMENT('/TESTUSER/PO1.XML')ASCLOB)
--------------------------------------------------------------------------------
<acl description="myacl" xmlns="http://xmlns.oracle.com/xdb/acl.xsd" xmlns:dav="
DAV:" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="
http://xmlns.oracle.com/xdb/acl.xsd                                     http://x
mlns.oracle.com/xdb/acl.xsd">
  <ace>
    <grant>true</grant>
    <principal>TESTUSER</principal>
    <privilege>
      <dav:all/>
    </privilege>
  </ace>
  <ace xmlns="http://xmlns.oracle.com/xdb/acl.xsd">
    <grant>true</grant>
    <principal>HR</principal>
    <privilege>
      <read-properties/>
```

```
      <read-contents/>
    </privilege>
  </ace>
</acl>

1 row selected.
```

See Also: Example 26–2, "Using DBMS_XDB.GETACLDOCUMENT" on page 26-4

Retrieving Privileges Granted to the Current User for a Particular Resource

Example 27–15 illustrates how to retrieve privileges granted to the current user using function `DBMS_XDB.getPrivileges`.

Example 27–15 Retrieving Privileges Granted to the Current User for a Particular Resource

```
SELECT XMLSerialize(DOCUMENT DBMS_XDB.getPrivileges('/TESTUSER/po1.xml')
                    AS CLOB)
  FROM DUAL;

XMLSERIALIZE(DOCUMENTDBMS_XDB.GETPRIVILEGES('/TESTUSER/PO1.XML')ASCLOB)
--------------------------------------------------------------------------------
<privilege xmlns="http://xmlns.oracle.com/xdb/acl.xsd" xmlns:xsi="http://www.w3.
org/2001/XMLSchema-instance" xsi:schemaLocation="http://xmlns.oracle.com/xdb/acl
.xsd http://xmlns.oracle.com/xdb/acl.xsd DAV: http://xmlns.oracle.com/xdb/dav.xs
d" xmlns:xdbacl="http://xmlns.oracle.com/xdb/acl.xsd" xmlns:dav="DAV:">
  <read-acl/>
  <dav:execute/>
  <read-contents/>
  <update-acl/>
  <dav:write-content/>
  <dav:read-current-user-privilege-set/>
  <link-to/>
  <resolve/>
  <dav:lock/>
  <unlink-from/>
  <write-config/>
  <dav:write-properties/>
  <dav:unlock/>
  <link/>
  <write-acl-ref/>
  <read-properties/>
  <dav:take-ownership/>
  <unlink/>
</privilege>

1 row selected.
```

Checking Whether the Current User Has Privileges on a Resource

Example 27–16 illustrates how to use function `DBMS_XDB.checkPrivileges` to check whether the current user has a given set of privileges on a resource. This function returns a nonzero value if the user has the privileges.

Example 27–16 Checking If a User Has a Certain Privileges on a Resource

```
SELECT DBMS_XDB.checkPrivileges(
         '/TESTUSER/po1.xml',
```

```
         XMLType('<privilege
                     xmlns="http://xmlns.oracle.com/xdb/acl.xsd"
                     xmlns:dav="DAV:"
                     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                     xsi:schemaLocation="http://xmlns.oracle.com/xdb/acl.xsd
                                         http://xmlns.oracle.com/xdb/acl.xsd">
                   <read-contents/>
                   <read-properties/>
                 </privilege>'))
  FROM DUAL;

DBMS_XDB.CHECKPRIVILEGES('/TESTUSER/PO1.XML',
---------------------------------------------
                                            1

1 row selected.
```

Example 27–16 checks to see if the access privileges `read-contents` and `read-properties` have been granted to the current user on resource `/TESTUSER/po1.xml`. The positive-integer return value shows that they have.

Checking Whether a User Has Privileges using the ACL and Resource Owner

Function `DBMS_XDB.ACLCheckPrivileges` is typically used by applications that must perform ACL evaluation on their own, before allowing a user to perform an operation.

Example 27–17 checks whether the ACL `/TESTUSER/acl1.xml` grants the privileges `read-contents` and `read-properties` to the current user, `sh`. The second argument, `TESTUSER`, is the user that is substituted for `DAV::owner` in the ACL when checking. Since user `sh` does *not* match any of the users granted the specified privileges, the return value is zero.

Example 27–17 Checking User Privileges using ACLCheckPrivileges

```
CONNECT sh
Enter password: <password>

Connected.

SELECT DBMS_XDB.ACLCheckPrivileges(
         '/TESTUSER/acl1.xml',
         'TESTUSER',
         XMLType('<privilege
                     xmlns="http://xmlns.oracle.com/xdb/acl.xsd"
                     xmlns:dav="DAV:"
                     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                     xsi:schemaLocation="http://xmlns.oracle.com/xdb/acl.xsd
                                         http://xmlns.oracle.com/xdb/acl.xsd">
                   <read-contents/>
                   <read-properties/>
                 </privilege>'))
  FROM DUAL;

DBMS_XDB.ACLCHECKPRIVILEGES('/TESTUSER/ACL1.XML','TESTUSER',
------------------------------------------------------------
                                                           0

1 row selected.
```

Retrieving the Path of the ACL that Protects a Given Resource

Example 27–18 retrieves the path of the ACL that protects a given resource, by using a RESOURCE_VIEW query. The query uses the fact that the XMLRef and ACLOID elements of the resource form the link between an ACL and a resource.

Example 27–18 Retrieving the Path of the ACL that Protects a Given Resource

```
SELECT rv1.ANY_PATH
  FROM RESOURCE_VIEW rv1
  WHERE
    XMLCast(XMLQuery('declare default element namespace
                      "http://xmlns.oracle.com/xdb/XDBResource.xsd"; (: :)
                      fn:data(/Resource/XMLRef)'
                     PASSING rv1.RES RETURNING CONTENT)
            AS REF XMLType)
    = make_ref(XDB.XDB$ACL,
               (SELECT XMLCast(XMLQuery('declare default element namespace
                                         "http://xmlns.oracle.com/xdb/XDBResource.xsd"; (: :)
                                         fn:data(/Resource/ACLOID)'
                                        PASSING rv2.RES RETURNING CONTENT)
                               AS REF XMLType)
                 FROM RESOURCE_VIEW rv2
                 WHERE equals_path(rv2.RES, '/TESTUSER/po1.xml') = 1));

ANY_PATH
------------------
/TESTUSER/acl1.xml
```

Example 27–18 retrieves the path to an ACL, given a resource protected by the ACL. The ACLOID of a protected resource (r) stores the OID of the ACL resource (a) that protects it. The REF of the ACL resource is the same as that of the object identified by the protected-resource ACLOID.

The REF of the resource ACLOID can be obtained using Oracle SQL function make_ref, which returns a REF to an object-table row with a given OID.

In this example, make_ref returns a REF to the row of table XDB$ACL whose OID is the /Resource/ACLOID for the resource /TESTUSER/po1.xml. The inner query returns the ACLOID of the resource. The outer query returns the path to the corresponding ACL.

Retrieving the Paths of All Resources Protected by a Given ACL

Example 27–19 retrieves the paths of all resources protected by a given ACL.

Example 27–19 Retrieving the Paths of All Resources Protected by a Given ACL

```
SELECT rv1.ANY_PATH
  FROM RESOURCE_VIEW rv1
  WHERE make_ref(XDB.XDB$ACL,
                 XMLCast(XMLQuery('declare default element namespace
                                   "http://xmlns.oracle.com/xdb/XDBResource.xsd"; (: :)
                                   fn:data(/Resource/ACLOID)'
                                  PASSING rv1.RES RETURNING CONTENT)
                         AS REF XMLType))
       = (SELECT XMLCast(XMLQuery('declare default element namespace
                                   "http://xmlns.oracle.com/xdb/XDBResource.xsd"; (: :)
                                   fn:data(/Resource/XMLRef)'
                                  PASSING rv2.RES RETURNING CONTENT)
```

```
                    AS REF XMLType)
            FROM RESOURCE_VIEW rv2
            WHERE equals_path(rv2.RES, '/TESTUSER/acl1.xml') = 1);

ANY_PATH
-----------------
/TESTUSER/po1.xml

1 row selected.
```

Example 27–19 retrieves the paths to the resources whose `ACLOID REF` matches the `REF` of the ACL resource whose path is `/TESTUSER/acl1.xml`. Function `make_ref` returns the resource `ACLOID REF`.

The inner query retrieves the `REF` of the specified ACL. The outer query selects the paths of the resources whose `ACLOID REF` matches the `REF` of the specified ACL.

ACL Caching

Since ACLs are checked for each access to the data they protect, the performance of the ACL check operation is critical to the performance of such data, including Oracle XML DB Repository resources. In Oracle XML DB, the required performance for this repository operation is achieved by employing several caches.

ACLs are saved in a cache that is shared by all sessions in the database instance. When an ACL is updated, its entry in the cache is invalidated, together with all objects dependent on it. The next time the ACL is used, a new copy of it is brought into the cache. Oracle recommends that you share ACLs among resources as much as possible.

There is a session-specific cache of privileges granted to a given user by a given ACL. The entries in this cache have a time out (in seconds) specified by the element `<acl-max-age>` in the Oracle XML DB configuration file (`/xdbconfig.xml`). For maximum performance, set this timeout as large as possible. But note that there is a trade-off here: the greater the timeout, the longer it takes for current sessions to pick up an updated ACL.

Oracle XML DB also maintains caches to improve performance when using ACLs that have LDAP principals (LDAP groups or users). The goal of these caches is to minimize network communication with the LDAP server. One is a shared cache that maps LDAP GUIDs to the corresponding LDAP nicknames and Distinguished Names (DNs). This is used when an ACL document is being displayed (or converted to `CLOB` or `VARCHAR2` values from an `XMLType` instance). To purge this cache, use procedure `DBMS_XDBZ.purgeLDAPCache`. The other cache is session-specific and maps LDAP groups to their members (nested membership). Note that whenever Oracle XML DB encounters an LDAP group for the first time (in a session) it gets the nested membership of that group from the LDAP server. Hence it is best to use groups with as few members and levels of nesting as possible.

Repository Resources and Database Table Security

Resources in Oracle XML DB Repository are of two types:

- LOB-based (content is stored in a LOB which is part of the resource). Access is determined only by the ACL that protects the resource.
- `REF`-based (content is XML and is stored in a database table). Users must have the appropriate privilege for the underlying table or view where the XML content is stored in addition to ACL permissions for the resource.

Since the content of a REF-based resource can be stored in a table, it is possible to access this data directly using SQL queries on the table. A **uniform** access control mechanism is one where the privileges needed for access are independent of the method of access (for example, FTP, HTTP, or SQL). To provide a uniform security mechanism using ACLs, the underlying table must first be **hierarchy-enabled**, before resources that reference the rows in the table are inserted into Oracle XML DB.

The default tables produced by XML schema registration are hierarchy-enabled. Enabling hierarchy is the default behavior when you register an XML schema with Oracle XML DB. You can also enable hierarchy after registration, using procedure DBMS_XDBZ.enable_hierarchy.

Enabling hierarchy on a resource table does the following:

- Adds two hidden columns to store the ACLOID and the OWNER of the resources that reference the rows in the table.
- Adds a row-level security (RLS) policy to the table, which checks the ACL whenever a SELECT, UPDATE, or DELETE operation is executed on the table.
- Creates a database trigger, called the **path-index trigger**, that ensures that the last-modified information for a resource is updated whenever the corresponding row is updated in the XMLType table where the content is stored.

See Also:

- *Oracle Database PL/SQL Packages and Types Reference* for information about procedure DBMS_XMLSCHEMA.registerSchema
- *Oracle Database PL/SQL Packages and Types Reference* for information about procedure DBMS_XDBZ.enable_hierarchy

In any given table, it is possible that only some of the objects are mapped to Oracle XML DB resources. Only those objects that are mapped undergo ACL checking, but *all* of the objects have table-level security.

Note: You cannot hide data in XMLType tables from other users if *out-of-line* storage of is used. Out-of-line data is *not* protected by ACL security.

Optimization: Do not enforce acl-based security if you do not need it

ACL-based security provides control of access to XML content document-by-document, rather than just table-by-table. When you call PL/SQL procedure DBMS_XMLSCHEMA.register_chema, the tables it creates have ACL-based security enabled, by default.

One effect of this is that when the XML content of such a table is accessed using a SQL statement, a call to sys_checkACL is automatically added to the query WHERE clause, to ensure that the ACL security that was defined is enforced at the SQL level.

Enforcing ACL-based security adds overhead to the SQL query, however. If ACL-based security is *not* required, then use procedure disable_hierarchy in package DBMS_XDBZ to turn *off* ACL checking.

When ACL-based security is enabled for an XMLType table, the execution plan output for a query of that table contains a filter similar to the following:

```
3 - filter(SYS_CHECKACL("ACLOID","OWNERID",xmltype(''<privilege
```

```
xmlns="http://xmlns.oracle.com/xdb/acl.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.oracle.com/xdb/acl.xsd
http://xmlns.oracle.com/xdb/acl.xsd
DAV:http://xmlns.oracle.com/xdb/dav.xsd">
          <read-properties/><read-contents/></privilege>''))=1)
```

In this example, the filter checks that the user performing the SQL query has `read-contents` privilege on each of the documents to be accessed.

After calling `DBMX_XDBZ.disable_hierarchy`, an execution plan of the same query does not show `SYS_CHECKACL` in the filter.

> **See Also:** *Oracle Database PL/SQL Packages and Types Reference* for information about procedure `DBMS_XDBZ.disable_hierarchy`

Integrating Oracle XML DB with LDAP

This section discusses allowing LDAP users to use the features of Oracle XML DB, including ACLs. The typical scenario is a single, shared database schema (user), to which multiple LDAP users are mapped. This mapping is maintained in the Oracle Internet Directory. End users can log into the database using their LDAP username and password. They are then automatically mapped to the corresponding shared database schema. (Users can log in using SQL or any of the supported Oracle XML DB protocols.) The implicit ACL resolution is based on the current LDAP user and the corresponding LDAP group membership information.

Before you can use LDAP users and groups as principals in Oracle XML DB ACLs, the following prerequisites must be satisfied:

- An Oracle Internet Directory must be set up, and the database must be registered with it.
- SSL authentication must be set up between the database and the Oracle Internet Directory.
- A database user must be created that corresponds to the shared database schema.
- The LDAP users must be created and mapped in the Oracle Internet Directory to the shared database schema.
- The LDAP groups must be created and their members must be specified.
- ACLs must be defined for the LDAP groups and users, and they must be used to protect the repository resources to be accessed by the LDAP users.

> **See Also:**
>
> - *Oracle Internet Directory Administrator's Guide*
> - *Oracle Database Advanced Security Administrator's Guide* for information about setting up SSL authentication
> - *Oracle Database Enterprise User Security Administrator's Guide* for information about using shared database schemas for enterprise (LDAP) users

Example 27–20 shows an ACL for an LDAP user. Element `<principal>` contains the full *distinguished name* of the LDAP user – in this case, `cn=user1,ou=Americas,o=oracle,l=redwoodshores,st=CA,c=US`.

Example 27–20 ACL Referencing an LDAP User

```
<acl description="/public/txmlacl1/acl1.xml"
     xmlns="http://xmlns.oracle.com/xdb/acl.xsd" xmlns:dav="DAV:"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://xmlns.oracle.com/xdb/acl.xsd
                         http://xmlns.oracle.com/xdb/acl.xsd">
  <ace principalFormat="DistinguishedName">
    <grant>true</grant>
    <principal>cn=user1,ou=Americas,o=oracle,l=redwoodshores,st=CA,c=US
    </principal>
    <privilege>
      <dav:all/>
    </privilege>
  </ace>
</acl>
```

> **See Also:** *Oracle Internet Directory Administrator's Guide* for the format of an LDAP user distinguished name

Example 27–21 shows an ACL for an LDAP group. Element `<principal>` contains the full distinguished name of the LDAP group.

Example 27–21 ACL Referencing an LDAP Group

```
<acl xmlns="http://xmlns.oracle.com/xdb/acl.xsd" xmlns:dav="DAV:"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://xmlns.oracle.com/xdb/acl.xsd
                         http://xmlns.oracle.com/xdb/acl.xsd">
  <ace principalFormat="DistinguishedName">
    <grant>true</grant>
    <principal>cn=grp1,ou=Americas,o=oracle,l=redwoodshores,st=CA,c=US</principal>
    <privilege>
      <dav:read/>
    </privilege>
  </ace>
</acl>
```

> **See Also:** *Oracle Internet Directory Administrator's Guide* for the format of an LDAP group distinguished name

28

Accessing the Repository using Protocols

This chapter describes how to access Oracle XML DB Repository data using FTP, HTTP(S)/WebDAV protocols.

This chapter contains these topics:

- Overview of Oracle XML DB Protocol Server
- Oracle XML DB Protocol Server Configuration Management
- Using FTP and Oracle XML DB Protocol Server
- Using HTTP(S) and Oracle XML DB Protocol Server
- Using WebDAV and Oracle XML DB

Overview of Oracle XML DB Protocol Server

As described in Chapter 2, "Getting Started with Oracle XML DB" and Chapter 21, "Accessing Oracle XML DB Repository Data", Oracle XML DB Repository provides a hierarchical data repository in the database, designed for XML. Oracle XML DB Repository maps path names (or URLs) onto database objects of `XMLType` and provides management facilities for these objects.

Oracle XML DB also provides the Oracle XML DB *protocol server*. This supports standard Internet protocols, FTP, WebDAV, and HTTP(S), for accessing its hierarchical repository or file system. Note that HTTPS provides *secure* access to Oracle XML DB Repository.

These protocols can provide direct access to Oracle XML DB for many users without having to install additional software. The user names and passwords to be used with the protocols are the same as those for SQL*Plus. Enterprise users are also supported. Database administrators can use these protocols and resource APIs such as `DBMS_XDB` to access Oracle Automatic Storage Management (Oracle ASM) files and folders in the repository virtual folder `/sys/asm`.

> **See Also:** Chapter 21, "Accessing Oracle XML DB Repository Data" for more information about accessing repository information, and restrictions on that access

Note:

- When accessing virtual folder `/sys/asm` using Oracle XML DB protocols, you must log in with the privileges of role `DBA` but as a user other than `SYS`.
- Oracle XML DB protocols are *not* supported on EBCDIC platforms.

Session Pooling

Oracle XML DB protocol server maintains a shared pool of sessions. Each protocol connection is associated with one session from this pool. After a connection is closed the session is put back into the shared pool and can be used to serve later connections.

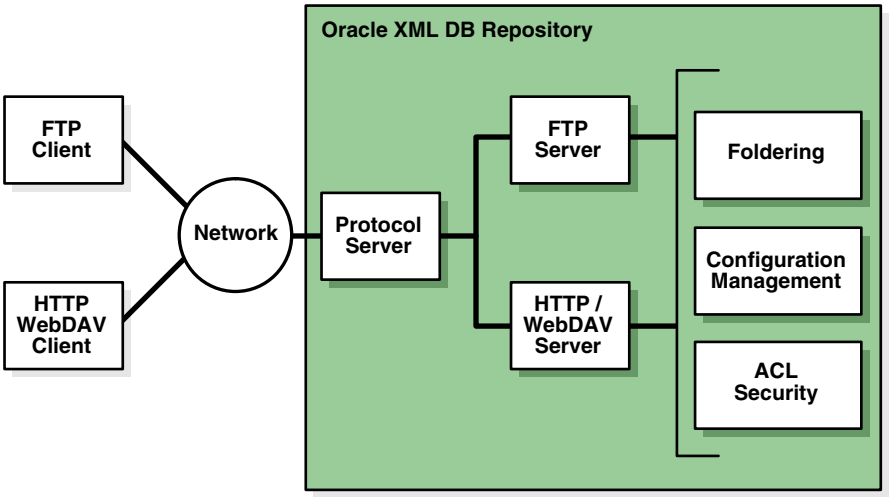
Session pooling improves performance of HTTP(S) by avoiding the cost of re-creating session states, especially when using HTTP 1.0, which creates new connections for each request. For example, a couple of small files can be retrieved by an existing HTTP/1.1 connection in the time necessary to create a database session. You can tune the number of sessions in the pool by setting session-pool-size in Oracle XML DB `xdbconfig.xml` file, or disable it by setting pool size to zero.

Session pooling can affect users writing *Java servlets*, because other users can see session state initialized by another request for a different user. Hence, servlet writers should only use session memory, such as Java static variables, to hold data for the entire application rather than for a particular user. State for each user must be stored in the database or in a lookup table, rather than assuming that a session only exists for a single user.

See Also: Chapter 32, "Writing Oracle XML DB Applications in Java"

Figure 28–1 illustrates the Oracle XML DB protocol server components and how they are used to access files in Oracle XML DB Repository and other data. Only the relevant components of the repository are shown

Figure 28–1 Oracle XML DB Architecture: Protocol Server



Oracle XML DB Protocol Server Configuration Management

Oracle XML DB protocol server uses configuration parameters stored in `/xdbconfig.xml` to initialize its startup state and manage session level configuration. The following section describes the protocol-specific configuration parameters that you can configure in the Oracle XML DB configuration file. The session pool size and timeout parameters cannot be changed dynamically, that is, you must restart the database in order for these changes to take effect.

> **See Also:** "Configuring Oracle XML DB using xdbconfig.xml" on page 34-5

Configuring Protocol Server Parameters

Figure 28–1 shows the parameters common to all protocols. All parameter names in this table, except those starting with `/xdbconfig`, are relative to the following XPath in the Oracle XML DB configuration schema:

```
/xdbconfig/sysconfig/protocolconfig/common
```

- *FTP-specific parameters* – Table 28–2 shows the FTP-specific parameters. These are relative to the following XPath in the Oracle XML DB configuration schema:

  ```
  /xdbconfig/sysconfig/protocolconfig/ftpconfig
  ```

- *HTTP(S)/WebDAV specific parameters, except servlet-related parameters* – Table 28–3 shows the HTTP(S)/WebDAV-specific parameters. These parameters are relative to the following XPath in the Oracle XML DB configuration schema:

  ```
  /xdbconfig/sysconfig/protocolconfig/httpconfig
  ```

> **See Also:**
>
> - Chapter 34, "Administering Oracle XML DB" for more information about the configuration file `xdbconfig.xml`
> - "xdbconfig.xsd: XML Schema for Configuring Oracle XML DB" on page A-16
> - "Configuring Default Namespace to Schema Location Mappings" on page 34-10 for more information about the `schemaLocation-mappings` parameter
> - "Configuring XML File Extensions" on page 34-12 for more information about the `xml-extensions` parameter

For examples of the usage of these parameters, see the configuration file, `xdbconfig.xml`.

Table 28–1 Common Protocol Configuration Parameters

Parameter	Description
`extension-mappings/mime-mappings`	Specifies the mapping of file extensions to mime types. When a resource is stored in Oracle XML DB Repository, and its mime type is not specified, this list of mappings is used to set its mime type.

Table 28–1 (Cont.) Common Protocol Configuration Parameters

Parameter	Description
`extension-mappings/lang-mappings`	Specifies the mapping of file extensions to languages. When a resource is stored in Oracle XML DB Repository, and its language is not specified, this list of mappings is used to set its language.
`extension-mappings/encoding-mappings`	Specifies the mapping of file extensions to encodings. When a resource is stored in Oracle XML DB Repository, and its encoding is not specified, this list of mappings is used to set its encoding.
`xml-extensions`	Specifies the list of filename extensions that are treated as XML content by Oracle XML DB.
`session-pool-size`	Maximum number of sessions that are kept in the protocol server session pool
`/xdbconfig/sysconfig/call-timeout`	If a connection is idle for this time (in hundredths of a second), then the shared server serving the connection is freed up to serve other connections.
`session-timeout`	Time (in hundredths of a second) after which a session (and consequently the corresponding connection) is terminated by the protocol server if the connection has been idle for that time. This parameter is used only if the specific protocol session timeout is not present in the configuration
`schemaLocation-mappings`	Specifies the default schema location for a given namespace. This is used if the instance XML document does not contain an explicit `xsi:schemaLocation` attribute.
`/xdbconfig/sysconfig/default-lock-timeout`	Time period after which a WebDAV lock on a resource becomes invalid. This could be overridden by a Timeout specified by the client that locks the resource.

Table 28–2 Configuration Parameters Specific to FTP

Parameter	Description
`buffer-size`	Size of the buffer, in bytes, used to read data from the network during an FTP `put` operation. Set `buffer-size` to larger values for higher `put` performance. There is a trade-off between `put` performance and memory usage. The value can be from 1024 to 1048496, inclusive. The default value is 8192.

Table 28–2 (Cont.) Configuration Parameters Specific to FTP

Parameter	Description
`ftp-port`	Port on which FTP server listens. By default, this is `0`, which means that FTP is *disabled*. FTP is disabled by default because the FTP specification requires that passwords be transmitted in clear text, which can present a security hazard. To enable FTP, set this parameter to the FTP port to use, such as `2100`.
`ftp-protocol`	Protocol over which the FTP server runs. By default, this is `tcp`.
`ftp-welcome-message`	A user-defined welcome message that is displayed whenever an FTP client connects to the server. If this parameter is empty or missing, then the following default welcome message is displayed: "Unauthorized use of this FTP server is prohibited and may be subject to civil and criminal prosecution."
`session-timeout`	Time (in hundredths of a second) after which an FTP connection is terminated by the protocol server if the connection has been idle for that time.

Table 28–3 Configuration Parameters Specific to HTTP(S)/WebDAV (Except Servlet Parameters)

Parameter	Description
`http-port`	Port on which the HTTP(S)/WebDAV server listens, using protocol `http-protocol`. By default, this is `0`, which means that HTTP is *disabled*. If this parameter is empty (`<http-port/>`), then the default value of `0` applies. An empty parameter is *not* recommended. This parameter must be present, whether or not it is empty. Otherwise, validation of `xdbconfig.xml` against XML schema `xdbconfig.xsd` fails. The value must be different from the value of `http2-port`. Otherwise, an error is raised.
`http2-port`	Port on which the HTTP(S)/WebDAV server listens, using protocol `http2-protocol`. This parameter is *optional*, but, if present, then `http2-protocol` must also be present. Otherwise, an error is raised. The value must be different from the value of `http-port`. Otherwise, an error is raised. An empty parameter (`<http2-port/>`) also raises an error.
`http-protocol`	Protocol over which the HTTP(S)/WebDAV server runs on port `http-port`. Must be either `TCP` or `TCPS`. This parameter must be present. Otherwise, validation of `xdbconfig.xml` against XML schema `xdbconfig.xsd` fails. An empty parameter (`<http-protocol/>`) also raises an error.

Table 28–3 (Cont.) Configuration Parameters Specific to HTTP(S)/WebDAV (Except Servlet Parameters)

Parameter	Description
`http2-protocol`	Protocol over which the HTTP(S)/WebDAV server runs on port `http2-port`. Must be either `TCP` or `TCPS`. If this parameter is empty (`<http2-protocol/>`), then the default value of `TCP` applies. (An empty parameter is *not* recommended.) This parameter is *optional*, but, if present, then `http2-port` must also be present. Otherwise, an error is raised.
`session-timeout`	Time (in hundredths of a second) after which an HTTP(S) session (and consequently the corresponding connection) is terminated by the protocol server if the connection has been idle for that time.
`max-header-size`	Maximum size (in bytes) of an HTTP(S) header
`max-request-body`	Maximum size (in bytes) of an HTTP(S) request body
`webappconfig/welcome-file-list`	List of filenames that are considered welcome files. When an HTTP(S) `get` request for a container is received, the server first checks if there is a resource in the container with any of these names. If so, then the contents of that file are sent, instead of a list of resources in the container.
`default-url-charset`	The character set in which an HTTP(S) protocol server assumes incoming URL is encoded when it is not encoded in UTF-8 or the Content-Type field Charset parameter of the request.
`allow-repository-anonymous-access`	Indication of whether or not anonymous HTTP access to Oracle XML DB Repository data is allowed using an unlocked `ANONYMOUS` user account. The default value is `false`, meaning that unauthenticated access to repository data is *blocked*. See "Anonymous Access to Oracle XML DB Repository using HTTP" on page 28-17.
`expire`	HTTP header that specifies the expiration date and time for a URL. See "Controlling URL Expiration Time" on page 28-17.

Configuring Secure HTTP (HTTPS)

To enable Oracle XML DB Repository to use *secure* HTTP connections (HTTPS), a database administrator (DBA) must configure the database accordingly: configure parameters `http2-port` and `http2-protocol`, enable the HTTP Listener to use SSL, and enable launching of the TCPS Dispatcher. After doing this, the DBA must stop, then restart, the database and the listener.

> **See Also:** "Configuring Oracle XML DB using xdbconfig.xml" on page 34-5 for information about configuring Oracle XML DB parameters

Enable the HTTP Listener to Use SSL

A database administrator must carry out the following steps, to configure the HTTP Listener for SSL.

> **Note:** Configuring the HTTP Listener for SSL requires that you have the option Oracle Advanced Security.

1. *Create a wallet for the server and import a certificate* – Use Oracle Wallet Manager to do the following:
   a. Create a wallet for the server.
   b. If a valid certificate with distinguished name (DN) of the server is not available, create a certificate request and submit it to a certificate authority. Obtain a valid certificate from the authority.
   c. Import a valid certificate with the distinguished name (DN) of the server into the server.
   d. Save the new wallet in *obfuscated* form, so that it can be opened without a password.

      See Also: *Oracle Database Advanced Security Administrator's Guide* for information about how to create a wallet

2. *Specify the wallet location to the server* – Use Oracle Net Manager to do this. Ensure that the configuration is saved to disk. This step updates files `sqlnet.ora` and `listener.ora`.
3. *Disable client authentication* at the server, since most Web clients do not have certificates. Use Oracle Net Manager to do this. This step updates file `sqlnet.ora`.
4. *Create a listening end point that uses TCP/IP with SSL* – Use Oracle Net Manager to do this. This step updates file `listener.ora`.

   See Also: *Oracle Database Advanced Security Administrator's Guide* for detailed information regarding steps 1 through 4

Enable TCPS Dispatcher

A database administrator must edit the database `pfile` to enable launching of a TCPS dispatcher during database startup. The following line must be added to the file, where *`SID`* is the SID of the database:

```
dispatchers=(protocol=tcps)(service=SIDxdb)
```

The database `pfile` location depends on your operating system, as follows:

- *MS Windows* – *`PARENT`*`/admin/orcl/pfile`, where *`PARENT`* is the parent folder of folder `ORACLE_HOME`
- *UNIX, Linux* – `$ORACLE_HOME/admin/$ORACLE_SID/pfile`

Interaction with Oracle XML DB File-System Resources

The protocol specifications, RFC 959 (FTP), RFC 2616 (HTTP), and RFC 2518 (WebDAV) implicitly assume an abstract, hierarchical file system on the server side. This is mapped to Oracle XML DB Repository. The repository provides:

- Name resolution.
- Security based on access control lists (ACLs). An ACL is a list of access control entries that determine which principals have access to a given resource or resources. See also Chapter 27, "Repository Access Control".
- The ability to store and retrieve any content. The repository can store both binary data input through FTP and XML schema-based documents.

See Also:

- http://www.ietf.org/rfc/rfc959.txt
- http://www.ietf.org/rfc/rfc2616.txt
- http://www.ietf.org/rfc/rfc2518.txt

Protocol Server Handles XML Schema-Based or Non-Schema-Based XML Documents

Oracle XML DB protocol server enhances the protocols by always checking if XML documents being inserted are based on XML schemas registered in Oracle XML DB Repository.

- If the incoming XML document specifies an XML schema, then the Oracle XML DB storage to use is determined by that XML schema. This functionality is especially useful when you must store XML documents object-relationally in the database using simple protocols like FTP or WebDAV instead of using SQL statements.
- If the incoming XML document is not XML schema-based, then it is stored as a binary document.

Event-Based Logging

In certain cases, it may be useful to log the requests received and responses sent by a protocol server. This can be achieved by setting event number `31098` to level `2`. To set this event, add the following line to your `init.ora` file and restart the database:

```
event="31098 trace name context forever, level 2"
```

Using FTP and Oracle XML DB Protocol Server

The following sections describe FTP features supported by Oracle XML DB.

Oracle XML DB Protocol Server: FTP Features

File Transfer Protocol (FTP) is one of the oldest and most popular protocols on the net. FTP is specified in RFC959 and provides access to heterogeneous file systems in a uniform manner. FTP works by providing well-defined commands (methods) for communication between the client and the server. The transfer of command messages and the return of status happens on a single connection. However, a new connection is opened between the client and the server for data transfer. With HTTP(S), commands and data are transferred using a single connection.

FTP is implemented by dedicated clients at the operating system level, file-system explorer clients, and browsers. FTP is typically session-oriented: a user session is created through an explicit logon, a number of files or directories are downloaded and browsed, and then the connection is closed.

Note: For security reasons, FTP is *disabled*, by default. This is because the IETF FTP protocol specification requires that passwords be transmitted in clear text. Disabling is done by configuring the FTP server port as zero (0). To enable FTP, set the `ftp-port` parameter to the FTP port to use, such as 2100.

See Also:

- RFC 959: FTP Protocol Specification – `http://www.ietf.org/rfc/rfc959.txt`
- "Configuring Oracle XML DB using xdbconfig.xml" on page 34-5 for information about configuring parameters

FTP Features That Are Not Supported

Oracle XML DB implements FTP, as defined by RFC 959, with the *exception* of the following optional features:

- Record-oriented files, for example, only the `FILE` structure of the `STRU` method is supported. This is the most widely used structure for transfer of files. It is also the default specified by the specification. Structure mount is not supported.
- Append.
- Allocate. This pre-allocates space before file transfer.
- Account. This uses the insecure Telnet protocol.
- Abort.

Supported FTP Client Methods

For access to the repository, Oracle XML DB supports the following FTP client methods.

- `cdup` – change working directory to parent directory
- `cwd` – change working directory
- `dele` – delete file (not directory)
- `list`, `nlst` – list files in working directory
- `mkd` – create directory
- `noop` – do nothing (but timeout counter on connection is reset)
- `pasv`, `port` – establish a TCP data connection
- `pwd` – get working directory
- `quit` – close connection and quit FTP session
- `retr` – retrieve data using an established connection
- `rmd` – remove directory
- `rnfr`, `rnto` – rename file (two-step process: from file, to file)
- `stor` – store data using an established connection
- `syst` – get system version
- `type` – change data type: `ascii` or `image` binary types only

- `user`, `pass` – user login

  See Also:

  - "FTP Quote Methods" for supported FTP `quote` methods
  - "Using FTP with Oracle ASM Files" on page 28-11 for an example of using FTP method `proxy`

FTP Quote Methods

Oracle Database supports several FTP `quote` methods, which provide information directly to Oracle XML DB.

- **`rm_r`** – Remove file or folder *<resource_name>*. If a folder, recursively remove all files and folders contained in *<resource_name>*.

  ```
  quote rm_r <resource_name>
  ```

- **`rm_f`** – Forcibly remove a resource.

  ```
  quote rm_f <resource_name>
  ```

- **`rm_rf`** – Combines `rm_r` and `rm_f`: Forcibly and recursively removes files and folders.

  ```
  quote rm_rf <resource_name>
  ```

- **`set_nls_locale`** – Specify the character-set encoding (*<charset_name>*) to be used for file and directory names in FTP methods (including names in method responses).

  ```
  quote set_nls_locale {<charset_name> | NULL}
  ```

  Only IANA character-set names can be specified for *<charset_name>*. If `nls_locale` is set to `NULL` or is not set, then the database character set is used.

- **`set_charset`** – Specify the character set of the data to be sent to the server.

  ```
  quote set_charset  {<charset_name> | NULL}
  ```

  The `set_charset` method applies to only *text* files, not binary files, as determined by the file-extension mapping to MIME types that is defined in configuration file `xdbconfig.xml`.

  If the parameter provided to `set_charset` is ***<charset_name>*** (*not* `NULL`), then it specifies the character set of the data.

  If the parameter provided to `set_charset` is `NULL`, or if no `set_charset` command is given, then the *MIME type* of the data determines the character set for the data.

  - If the MIME type is *not* `text/xml`), then the data is not assumed to be XML. The database character set is used.
  - If the MIME type is **`text/xml`**, then the data represents an XML document.

    If a *byte order mark*[1] (BOM) is present in the XML document, then it determines the character set of the data.

    If there is *no* BOM, then:

[1] BOM is a Unicode-standard signature that indicates the order of the stream of bytes that follows it.

* If there is an *encoding declaration* in the XML document, then it determines the character set of the data.
* If there is *no* encoding declaration, then the UTF-8 character set is used.

Using FTP with Oracle ASM Files

Oracle Automatic Storage Management (Oracle ASM) organizes database files into disk groups for simplified management and added benefits such as database mirroring and I/O balancing. Database administrators can use protocols and resource APIs to access Oracle ASM files in the Oracle XML DB repository *virtual folder* `/sys/asm`. All files in `/sys/asm` are binary.

Typical uses are listing, copying, moving, creating, and deleting Oracle ASM files and folders. Example 28–1 is an example of navigating the Oracle ASM virtual folder and listing the files in a subfolder.

The structure of the Oracle ASM virtual folder, `/sys/asm`, is described in Chapter 21, "Accessing Oracle XML DB Repository Data". In Example 28–1, the disk groups are `DATA` and `RECOVERY`; the database name is `MFG`; and the directories created for aliases are `dbs` and `tmp`. This example navigates to a subfolder, lists its files, and copies a file to the local file system.

Example 28–1 Navigating Oracle ASM Folders

```
ftp> open myhost 7777
ftp> user system
Password required for SYSTEM
Password: password
ftp> cd /sys/asm
ftp> ls
DATA
RECOVERY
ftp> cd DATA
ftp> ls
dbs
MFG
ftp> cd dbs
ftp> ls
t_dbl.f
t_axl.f
ftp> binary
ftp> get t_dbl.f, t_axl.f
ftp> put my_db2.f
```

In Example 28–1, after connecting to and logging onto database `myhost` (first four lines), FTP methods `cd` and `ls` are used to navigate and list folders, respectively. When in folder `/sys/asm/DATA/dbs`, FTP command `get` is used to copy files `t_db1.f` and `t_ax1.f` to the current folder of the local file system. Then, FTP command `put` is used to copy file `my_db2.f` from the local file system to folder `/sys/asm/DATA/dbs`.

Database administrators can copy Oracle Automatic Storage Management (Oracle ASM) files from *one database server to another* or between the database and a local file system. Example 28–2 shows copying between two databases. For this, the `proxy` FTP client method can be used, if available. The `proxy` method provides a *direct* connection to two different remote FTP servers.

Example 28–2 copies an Oracle ASM file from one database to another. Terms with the suffix `1` correspond to database `server1`. Terms with the suffix `2` correspond to database `server2`. Note that, depending on your FTP client, the passwords you type

might be echoed on your screen. Take the necessary precautions so that others do not see these passwords.

Example 28–2 Transferring Oracle ASM Files Between Databases with FTP proxy Method

```
 1 ftp> open server1 port1
 2 ftp> user username1
 3 Password required for USERNAME1
 4 Password: password-for-username1
 5 ftp> cd /sys/asm/DATAFILE/MFG/DATAFILE
 6 ftp> proxy open server2 port2
 7 ftp> proxy user username2
 8 Password required for USERNAME2
 9 Password: password-for-username2
10 ftp> proxy cd /sys/asm/DATAFILE/MFG/DATAFILE
11 ftp> proxy put dbs2.f tmp1.f
12 ftp> proxy get dbs1.f tmp2.f
```

In Example 28–2:

- Line 1 opens an FTP control connection to the Oracle XML DB FTP server, `server1`.
- Lines 2–4 log the database administrator onto `server1` as `USERNAME1`.
- Line 5 navigates to `/sys/asm/DATAFILE/MFG/DATAFILE` on `server1`.
- Line 6 opens an FTP control connection to the second database server, `server2`. At this point, the FTP command `proxy ?` could be issued to see the available FTP commands on the secondary connection. (This is not shown.)
- Lines 7–9 log the database administrator onto `server2` as `USERNAME2`.
- Line 10 navigates to `/sys/asm/DATAFILE/MFG/DATAFILE` on `server2`.
- Line 11 copies Oracle ASM file `dbs2.f` from `server2` to Oracle ASM file `tmp1.f` on `server1`.
- Line 12 copies Oracle ASM file `dbs1.f` from `server1` to Oracle ASM file `tmp2.f` on `server2`.

Using FTP on the Standard Port Instead of the Oracle XML DB Default Port

You can use the Oracle XML DB configuration file, `/xdbconfig.xml`, to configure FTP to listen on any port. By default, FTP listens on a nonstandard, unprotected port. To use FTP on the standard port, 21, your database administrator must do the following:

1. (UNIX only) Use this shell command to ensure that the owner and group of executable file `tnslsnr` are `root`:

   ```
   % chown root:root $ORACLE_HOME/bin/tnslsnr
   ```

2. (UNIX only) Add the following entry to the listener file, `listener.ora`, where *`hostname`* is your host name:

   ```
   (DESCRIPTION =
     (ADDRESS = (PROTOCOL = TCP) (HOST = hostname) (PORT = 21))
     (PROTOCOL_STACK = (PRESENTATION = FTP) (SESSION = RAW)))
   ```

3. (UNIX only) Use shell command `id` to determine the *`user_id`* and *`group_id`* that were used to install Oracle Database. *`oracle_installation_user`* is the name of the user who installed the database.

```
% id oracle_installation_user
uid=user_id(oracle_installation_user) gid=group_id(dba)
```

4. (UNIX only) Stop, then restart the listener, using the following shell commands, where *`user_id`* and *`group_id`* are the UNIX user and group identifiers obtained in step 3.

```
% lsnrctl stop
% tnslsnr LISTENER -user user_id -group group_id &
```

   Use the ampersand (`&`), to execute the second command in the background. Do not use `lsnrctl start` to start the listener.

5. Use PL/SQL procedure `DBMS_XDB.setftpport` with `SYS as SYSDBA` to set the FTP port number to 21 in the Oracle XML DB configuration file `/xdbconfig.xml`.

```
SQL> exec DBMS_XDB.setFTPPort(21);
```

6. Force the database to reregister with the listener, using this SQL statement:

```
SQL> ALTER SYSTEM REGISTER;
```

7. Check that the listener is correctly configured, using this shell command:

```
% lsnrctl status
```

See Also:

- *Oracle Database Net Services Reference* for information about listener parameters and file `listener.ora`
- *Oracle Database Net Services Reference*, section "Port Number Limitations" for information about running on privileged ports

Using IPv6 IP Addresses with FTP

Starting with 11g Release 2 (11.2), Oracle Database supports the use of Internet Protocol Version 6, IPv6 (in addition to Internet Protocol Version 4). Example 28–3 shows how to make an FTP connection with the IPv6 address `2001::0db8:ffff:ffff:ffff`.

Example 28–3 FTP Connection Using IPv6

```
ftp> open 2001::0db8:ffff:ffff:ffff 1521
Connected to 2001::0db8:ffff:ffff:ffff.
220- xmlhost.example.com
Unauthorized use of this FTP server is prohibited and may be subject to civil and
criminal prosecution.
220- xmlhost.example.com FTP server (Oracle XML DB/Oracle Database) ready.
User (2001::0db8:ffff:ffff:ffff:(none)): username
331 pass required for USERNAME
Password: password-for-username
230 USERNAME logged in
ftp>
```

See Also: *Oracle Database Net Services Reference* for information about IPv6

FTP Server Session Management

Oracle XML DB protocol server also provides session management for this protocol. After a short wait for a new command, FTP returns to the protocol layer and the shared server is freed up to serve other connections. The duration of this short wait is configurable by changing parameter `call-timeout` in the Oracle XML DB configuration file. For high traffic sites, `call-timeout` should be shorter, so that more connections can be served. When new data arrives on the connection, the FTP server is re-invoked with fresh data. So, the long running nature of FTP does not affect the number of connections which can be made to the protocol server.

> **See Also:** "Configuring Oracle XML DB using xdbconfig.xml" on page 34-5 for information about configuring Oracle XML DB parameters

Handling Error 421. Modifying the Default Timeout Value of an FTP Session

If you are frequently disconnected from the server and must reconnect and traverse the entire directory before doing the next operation, you may need to modify the default timeout value for FTP sessions. If the session is idle for more than this period, it gets disconnected. You can increase the timeout value (default = 6000 centiseconds) by modifying the configuration document as follows and then restart the database:

Example 28–4 Modifying the Default Timeout Value of an FTP Session

```
DECLARE
  newconfig XMLType;
BEGIN
  SELECT
    updateXML(
      DBMS_XDB.cfg_get(),
      '/xdbconfig/sysconfig/protocolconfig/ftpconfig/session-timeout/text()',
      123456789)
    INTO newconfig
    FROM DUAL;
  DBMS_XDB.cfg_update(newconfig);
END;
/
COMMIT;
```

FTP Client Failure in Passive Mode

Do not use FTP in *passive mode* to connect remotely to a server that has `HOSTNAME` configured in `listener.ora` as `localhost` or `127.0.0.1`. If the `HOSTNAME` specified in server file `listener.ora` is `localhost` or `127.0.0.1`, then the server is configured for *local use only*. If you try to connect remotely to the server using FTP in passive mode, the FTP client fails. This is because the server passes IP address `127.0.0.1` (derived from `HOSTNAME`) to the client, which makes the client try to connect to itself, not to the server.

Using HTTP(S) and Oracle XML DB Protocol Server

Oracle XML DB implements HyperText Transfer Protocol (HTTP), HTTP 1.1 as defined in the RFC2616 specification.

Oracle XML DB Protocol Server: HTTP(S) Features

The Oracle XML DB HTTP(S) component in the Oracle XML DB protocol server implements the RFC2616 specification with the *exception* of the following optional features:

- `gzip` and compress transfer encodings
- byte-range headers
- The `TRACE` method (used for proxy error debugging)
- Cache-control directives (these require you to specify expiration dates for content, and are not generally used)
- TE, Trailer, Vary & Warning headers
- Weak entity tags
- Web common log format
- Multi-homed Web server

> **See Also:** RFC 2616: HTTP 1.1 Protocol Specification—`http://www.ietf.org/rfc/rfc2616.txt`

HTTP(S) Features That Are Not Supported

Digest Authentication (RFC 2617) is *not* supported. Oracle XML DB supports Basic Authentication, where a client sends the user name and password in clear text in the Authorization header.

Supported HTTP(S) Client Methods

For access to the repository, Oracle XML DB supports the following HTTP(S) client methods.

- `OPTIONS` – get information about available communication options
- `GET` – get document/data (including headers)
- `HEAD` – get headers only, without document body
- `PUT` – store data in resource
- `DELETE` – delete resource

The semantics of these HTTP(S) methods is in accordance with WebDAV. Servlets and Web services may support additional HTTP(S) methods, such as `POST`.

> **See Also:** "Supported WebDAV Client Methods" on page 28-21 for supported HTTP(S) client methods involving WebDAV

Using HTTP(S) on a Standard Port Instead of an Oracle XML DB Default Port

You can use the Oracle XML DB configuration file, `/xdbconfig.xml`, to configure HTTP(S) to listen on any port. By default, HTTP(S) listens on a nonstandard, unprotected port. To use HTTP or HTTPS on a standard port (80 for HTTP, 443 for HTTPS), your database administrator must do the following:

1. (UNIX only) Use this shell command to ensure that the owner and group of executable file `tnslsnr` are `root`:

   ```
   % chown root:root $ORACLE_HOME/bin/tnslsnr
   ```

2. (UNIX only) Add the following entry to the listener file, `listener.ora`, where *`hostname`* is your host name, and *`port_number`* is `80` for HTTP or `443` for HTTPS:

```
(DESCRIPTION =
  (ADDRESS = (PROTOCOL = TCP) (HOST = hostname) (PORT = port_number))
  (PROTOCOL_STACK = (PRESENTATION = HTTP) (SESSION = RAW)))
```

3. (UNIX only) Use shell command `id` to determine the *`user_id`* and *`group_id`* that were used to install Oracle Database. *`oracle_installation_user`* is the name of the user who installed the database.

```
% id oracle_installation_user
uid=user_id(oracle_installation_user) gid=group_id(dba)
```

4. (UNIX only) Stop, then restart the listener, using the following shell commands, where *`user_id`* and *`group_id`* are the UNIX user and group identifiers obtained in step 3.

```
% lsnrctl stop
% tnslsnr LISTENER -user user_id -group group_id &
```

   Use the ampersand (`&`), to execute the second command in the background. Do not use `lsnrctl start` to start the listener.

5. Use PL/SQL procedure `DBMS_XDB.sethtpport` with `SYS as SYSDBA` to set the HTTP(S) port number to *`port_number`* in the Oracle XML DB configuration file `/xdbconfig.xml`, where *`port_number`* is `80` for HTTP or `443` for HTTPS:

```
SQL> exec DBMS_XDB.setHTTPPort(port_number);
```

6. Force the database to reregister with the listener, using this SQL statement:

```
SQL> ALTER SYSTEM REGISTER;
```

7. Check that the listener is correctly configured:

```
% lsnrctl status
```

See Also:

- *Oracle Database Net Services Reference* for information about listener parameters and file `listener.ora`
- *Oracle Database Net Services Reference*, section "Port Number Limitations" for information about running on privileged ports

Using IPv6 IP Addresses with HTTP(S)

Starting with 11g Release 2 (11.2), Oracle Database supports the use of Internet Protocol Version 6, IPv6 (in addition to Internet Protocol Version 4). IPv6 addresses in URLs are enclosed in brackets (`[]`). Here is an example:

```
http://[2001::0db8:ffff:ffff:ffff]:8080/
```

See Also: *Oracle Database Net Services Reference* for information about IPv6

HTTPS: Support for Secure HTTP

If properly configured, you can access Oracle XML DB Repository in a *secure* fashion, using HTTPS. See "Configuring Secure HTTP (HTTPS)" on page 28-6 for configuration information.

Note: If Oracle Database is installed on Microsoft Windows XP with *Service Pack 2* (SP2), then you must use HTTPS for WebDAV access to Oracle XML DB Repository, or else you must make appropriate modifications to the Windows XP Registry. For information about the latter, see http://www.microsoft.com/technet/prodtechnol/winxppro/maintain/sp2netwk.mspx#XSLTsection129121120120

Controlling URL Expiration Time

Optional configuration parameter `expire` specifies an HTTP `Expires` header. This header acts as a directive to the HTTP client, to specify the expiration date and time for a URL.

If cached, the document targeted by a URL can be fetched from the client cache rather than from the server, until this expiration time has passed. After that time, the cache copy is out-of-date and a new copy must be obtained from the source (server).

The Oracle XML DB syntax for the `Expires` header, which is used in the `expire` configuration element, is a subset of the so-called alternate syntax defined for the `ExpiresDefault` directive of the Apache module `mod_expires`. See http://httpd.apache.org/docs/2.0/mod/mod_expires.html#AltSyn for that syntax.

These are the Oracle XML DB restrictions to the `ExpiresDefault` syntax:

- You cannot use `access` as the `<base>`. Only `now` and `modification` are allowed.
- The `<type>` values must appear in order of decreasing time period. For example, `year` must appear before, not after, `month`, since a year is a longer time period than a month.
- You can use at most one occurrence of each of the different `<type>` values. For example, you cannot have multiple `year` entries or multiple `day` entries.

Anonymous Access to Oracle XML DB Repository using HTTP

Optional configuration parameter `allow-repository-anonymous-access` controls whether or not anonymous HTTP access to Oracle XML DB Repository data is allowed using an unlocked `ANONYMOUS` user account. The default value is `false`, meaning that unauthenticated access to repository data is *blocked*. To allow anonymous HTTP access to the repository, you must set this parameter to `true`, and unlock the `ANONYMOUS` user account.

Caution: There is an inherent *security risk* associated with allowing anonymous access to the repository.

Parameter `allow-repository-anonymous-access` does *not* control anonymous access to the repository using *servlets*. Each servlet has its own `security-role-ref` parameter value to control its access.

See Also:

- Table 28–3 on page 28-5 for information about parameter `allow-repository-anonymous-access`
- "Configuring Oracle XML DB using xdbconfig.xml" on page 34-5 for information about configuring Oracle XML DB parameters
- "Configuring Oracle XML DB Servlets" on page 32-3 for information about parameter `security-role-ref`

Using Java Servlets with HTTP(S)

Oracle XML DB supports Java servlets. To use a Java servlet, it must be registered with a unique name in the Oracle XML DB configuration file, along with parameters to customize its action. It should be compiled, and loaded into the database. Finally, the servlet name must be associated with a pattern, which can be an extension such as `*.jsp` or a path name such as `/a/b/c` or `/sys/*`, as described in Java servlet application program interface (API) version 2.2.

While processing an HTTP(S) request, the path name for the request is matched with the registered patterns. If there is a match, then the protocol server invokes the corresponding servlet with the appropriate initialization parameters. For Java servlets, the existing Java Virtual Machine (JVM) infrastructure is used. This starts the JVM if need be, which in turn invokes a Java method to initialize the servlet, create response, and request objects, pass these on to the servlet, and run it.

See Also: Chapter 32, "Writing Oracle XML DB Applications in Java"

Embedded PL/SQL Gateway

You can use the PL/SQL gateway to implement a Web application entirely in PL/SQL. There are two implementations of the PL/SQL gateway:

- *mod_plsql* – a plug-in of *Oracle HTTP Server* that lets you invoke PL/SQL stored procedures using HTTP(S). Oracle HTTP Server is a component of both Oracle Application Server and Oracle Database. Do not confuse Oracle HTTP Server with the HTTP component of the Oracle XML DB protocol server.
- the *embedded* PL/SQL gateway – a gateway implementation that runs in the Oracle XML DB HTTP listener.

With the PL/SQL gateway (either implementation), a Web browser sends an HTTP(S) request in the form of a URL that identifies a stored procedure and provides it with parameter values. The gateway translates the URL, calls the stored procedure with the parameter values, and returns output (typically HTML) to the Web-browser client.

Using the embedded PL/SQL gateway simplifies installation, configuration, and administration of PL/SQL based Web applications. The embedded gateway uses the Oracle XML DB protocol server, not Oracle HTTP Server. Its configuration is defined by the Oracle XML DB configuration file, `/xdbconfig.xml`. However, the *recommended* way to configure the embedded gateway is to use the procedures in PL/SQL package `DBMS_EPG`, *not* to edit file `/xdbconfig.xml`.

See Also:

- *Oracle Database Advanced Application Developer's Guide* for information on using and configuring the embedded PL/SQL gateway
- Chapter 34, "Administering Oracle XML DB" for information on the configuration definition of the embedded gateway in `/xdbconfig.xml`
- *Oracle Fusion Middleware Administrator's Guide for Oracle HTTP Server* for conceptual information about using the PL/SQL gateway
- *Oracle HTTP Server mod_plsql User's Guide* for information about mod_plsql

Sending Multibyte Data From a Client

When a client sends multibyte data in a URL, RFC 2718 specifies that the client should send the URL using the `%HH` format, where *HH* is the hexadecimal notation of the byte value in UTF-8 encoding. The following are URL examples that can be sent to Oracle XML DB in an HTTP(S) or WebDAV context:

```
http://urltest/xyz%E3%81%82%E3%82%A2
http://%E3%81%82%E3%82%A2
http://%E3%81%82%E3%82%A2/abc%E3%81%86%E3%83%8F.xml
```

Oracle XML DB processes the requested URL, any URLs within an `IF` header, any URLs within the `DESTINATION` header, and any URLs in the `REFERRED` header that contains multibyte data.

The `default-url-charset` configuration parameter can be used to accept requests from some clients that use other, nonconforming, forms of URL, with characters that are not ASCII. If a request with such characters fails, try setting this value to the native character set of the client environment. The character set used in such URL fields must be specified with an IANA charset name.

`default-url-charset` controls the encoding for nonconforming URLs. It is not required to be set unless a nonconforming client that does not send the `Content-Type` charset is used.

See Also:

- RFC 2616: HTTP 1.1 Protocol Specification, `http://www.ietf.org/rfc/rfc2616.txt`
- "Configuring Oracle XML DB using xdbconfig.xml" on page 34-5 for information about configuring Oracle XML DB parameters

Characters That Are Not ASCII in URLs

Characters that are not ASCII that appear in URLs passed to an HTTP server should be converted to `UTF-8` and escaped in the `%HH` format, where *HH* is the hexadecimal notation of the byte value. For flexibility, the Oracle XML DB protocol server interprets the incoming URLs by testing whether it is encoded in one of the following character sets in the order presented here:

- UTF-8
- Charset parameter of the `Content-Type` field of the request, if specified

- Character set, if specified, in the `default-url-charset` configuration parameter
- Character set of the database

> **See Also:** "Configuring Oracle XML DB using xdbconfig.xml" on page 34-5 for information about configuring Oracle XML DB parameters

Controlling Character Sets for HTTP(S)

The following sections describe how character sets are controlled for data transferred using HTTP(S).

Request Character Set The character set of the HTTP(S) request body is determined with the following algorithm:

- The Content-Type header is evaluated. If the Content-Type header specifies a charset value, the specified charset is used.
- The MIME type of the document is evaluated as follows:
  - If the MIME type is "`*/xml`", the character set is determined as follows:

    - If a BOM is present, then `UTF-16` is used.

    - If an encoding declaration is present, the specified encoding is used.

    - If neither a BOM nor an encoding declaration is present, `UTF-8` is used.
  - If the MIME type is text, `ISO8859-1` is used.
  - If the MIME type is neither "`*/xml`" nor `text`, the database character set is used.

There is a difference between HTTP(S) and SQL or FTP. For text documents, the default is ISO8859-1, as specified by the IETF.org *RFC 2616: HTTP 1.1 Protocol Specification*.

Response Character Set

The response generated by Oracle XML DB HTTP Server is in the character set specified in the `Accept-Charset` field of the request. `Accept-Charset` can have a list of character sets. Based on the q-value, Oracle XML DB chooses one that does not require conversion. This might not necessarily be the charset with the highest q-value. If Oracle XML DB cannot find one, then the conversion is based on the highest q-value.

Using WebDAV and Oracle XML DB

Web Distributed Authoring and Versioning (WebDAV) is an IETF standard protocol used to provide users with a file-system interface to Oracle XML Repository over the Internet. The most popular way of accessing a WebDAV server folder is through WebFolders using Microsoft Windows.

WebDAV is an extension to the HTTP 1.1 protocol that lets an HTTP server act as a file server. It lets clients perform remote Web content authoring through a coherent set of methods, headers, request body formats and response body formats. For example, a DAV-enabled editor can interact with an HTTP/WebDAV server as if it were a file system. WebDAV provides operations to store and retrieve resources, create and list contents of resource collections, lock resources for concurrent access in a coordinated manner, and to set and retrieve resource properties.

Oracle XML DB WebDAV Features

Oracle XML DB supports the following WebDAV features:

- Foldering, specified by RFC2518
- Access Control

WebDAV is a set of extensions to the HTTP(S) protocol that allow you to edit or manage your files on remote Web servers. WebDAV can also be used, for example, to:

- Share documents over the Internet
- Edit content over the Internet

> **See Also:** RFC 2518: WebDAV Protocol Specification, `http://www.ietf.org/rfc/rfc2518.txt`

WebDAV Features That Are Not Supported

Oracle XML DB supports the contents of RFC2518, with the following exceptions:

- Lock-`NULL` resources create zero-length resources in the file system, and cannot be converted to folders.
- Methods `COPY`, `MOVE` and `DELETE` comply with section 2 of the Internet Draft titled 'Binding Extensions to WebDAV'.
- Depth-infinity locks
- Only Basic Authentication is supported.

Supported WebDAV Client Methods

For access to the repository, Oracle XML DB supports the following HTTP(S)/WebDAV client methods.

- `PROPFIND` (WebDAV-specific) – get properties for a resource
- `PROPPATCH` (WebDAV-specific) – set or remove resource properties
- `LOCK` (WebDAV-specific) – lock a resource (create or refresh a lock)
- `UNLOCK` (WebDAV-specific) – unlock a resource (remove a lock)
- `COPY` (WebDAV-specific) – copy a resource
- `MOVE` (WebDAV-specific) – move a resource
- `MKCOL` (WebDAV-specific) – create a folder resource (collection)

> **See Also:**
>
> - "Supported HTTP(S) Client Methods" on page 28-15 for additional supported HTTP(S) client methods
> - "Privileges" on page 27-5 for information about WebDAV privileges
> - "Adding Metadata using WebDAV PROPPATCH" on page 29-8

Using WebDAV with Microsoft Windows XP SP2

If Oracle Database is installed on Microsoft Windows XP with *Service Pack 2* (SP2), then you must use a secure connection (HTTPS) for WebDAV access to Oracle XML DB Repository, or else you must make appropriate modifications to the Windows XP Registry.

See Also:

- http://www.microsoft.com/technet/prodtechnol/winxppro/maintain/sp2netwk.mspx#XSLTsection129121120120 for information about making necessary modifications to the Windows XP registry
- "Configuring Secure HTTP (HTTPS)" on page 28-6

Creating a WebFolder in Microsoft Windows using Oracle XML DB and WebDAV

To create a WebFolder in Windows 2000, follow these steps:

1. **Start** > **My Network Places**.
2. Double-click **Add Network Place**.
3. Click **Next**.
4. Type the location of the folder, for example:

   `http://Oracle_server_name:HTTP_port_number`

   See Figure 28–2.
5. Click **Next**.
6. Enter any name to identify this WebFolder
7. Click **Finish**.

You can access Oracle XML DB Repository the same way that you access any Windows folder.

Figure 28–2 Creating a WebFolder in Microsoft Windows

29

User-Defined Repository Metadata

This chapter describes how to create and use XML metadata, which you associate with XML data and store in Oracle XML DB Repository.

This chapter contains these topics:

- Overview of Metadata and XML
- XML Schemas to Define Resource Metadata
- Adding, Updating, and Deleting Resource Metadata
- Querying XML Schema-Based Resource Metadata
- XML Image Metadata from Binary Image Metadata
- Adding Non-Schema-Based Resource Metadata
- PL/SQL Procedures Affecting Resource Metadata

Overview of Metadata and XML

Data that you use is often associated with additional information that is not part of the content. To process it in different ways, you can use such **metadata** to group or classify data. For example, you might have a collection of digital photographs, and you might associate metadata with each picture, such as information about the photographic characteristics (color composition, focal length) or context (location, kind of subject: landscape, people).

An Oracle XML DB repository **resource** is an XML document that contains both metadata and data. The data is the contents of element `Contents`. All other elements in the resource contain metadata. The data of a resource can be XML, but it need not be.

You can associate resources in the Oracle XML DB repository with metadata that you define. In addition to such *user-defined metadata*, each repository resource also has associated metadata that Oracle XML DB creates automatically and uses (transparently) to manage the resource. Such *system-defined metadata* includes properties such as the owner and creation date of each resource.

Except for system-defined metadata, you decide which resource information should be treated as data and which should be treated as metadata. For a photo resource, supplemental information about the photo is normally not considered to be part of the photo data, which is a binary image. For text, however, you sometimes have a choice of whether to include particular information in the resource contents (data) or keep it separate and associate it with the contents as metadata—that choice is often influenced by the applications that use or produce the data.

Kinds of Metadata – Uses of the Term

In addition to resource metadata (system-defined and user-defined), the term "metadata" is sometimes used to refer to the following:

- An XML *schema* is metadata that describes a class of XML documents.
- An XML *tag* (element or attribute name) is metadata that is used to label and organize the element content or attribute value.

You can associate metadata with an XML document that is the content of a repository resource in any of these ways:

- You can add additional XML elements containing the metadata information to the resource *contents*. For example, you could wrap digital image data in an XML document that also includes elements describing the photo. In this case, the data and its metadata are associated by being in the contents of the same resource. It is up to applications to separate the two and relate them correctly.
- You can add metadata information for a particular resource to the repository as the contents of a *separate resource*. In this case, it is up to applications to treat this resource as metadata and associate it with the data.
- You can add metadata information for a resource as repository *resource metadata*. In this case, Oracle XML DB recognizes the metadata as such. Applications can *discover* this metadata by querying the repository for it. They need not be informed separately of its existence and its association with the data.

See Also: "Oracle XML DB Repository Resources" on page 21-5

User-Defined Resource Metadata

Of these different ways of considering metadata, this chapter is about only the last of those just listed: *user-defined resource metadata*. Such metadata is itself *represented as XML*: it is XML data that is associated with other XML data, describing it or providing supplementary, related information.

User-defined metadata for resources can be either XML schema-based or not:

- Resource metadata that is *schema-based* is stored in separate (out-of-line) tables. These are related to the resource table by the resource OID, which is stored in the hidden object column `RESID` of the metadata tables.
- Resource metadata that is *not* schema-based is stored in a `CLOB` column in the resource table.

You can take advantage of schema-based metadata, in particular, to perform efficient queries and DML operations on resources. In this chapter, you learn how to perform the following tasks involving schema-based resource metadata:

- Create and register an *XML schema* that defines the metadata for a particular kind of resource.
- *Add* metadata to a repository resource, and *update* (modify) such metadata.
- *Query* resource metadata to find associated content.
- *Delete* specific metadata associated with a resource and *purge* all metadata associated with a resource.

In addition, you learn how to add non-schema-based metadata to a resource.

You can generally use user-defined resource metadata just as you would use resource data. In particular, versioning and access control management apply.

Typical uses of resource metadata include workflow applications, enforcing user rights management, tracking resource ownership, and controlling resource validity dates.

Scenario: Metadata for a Photo Collection

To illustrate the use of schema-based resource metadata, this chapter uses metadata associated with photographic image files that are stored in repository resources. You can create any number of different kinds of metadata to be associated with the same resource. For image files, examples create metadata for information about both 1) the technical aspects of a photo and 2) the photo subject or the uses to which a photo might be put. These two kinds of associated metadata are used to query photo resources.

XML Schemas to Define Resource Metadata

This section first defines the metadata to associate with each photo resource using XML Schema. An XML schema is created and registered for each kind (technique, category) of metadata.

The XML schema in Example 29–1 defines metadata used to describe the technical aspects of a photo image file. It uses PL/SQL procedure `DBMS_XMLSCHEMA.registerSchema` to register the XML schema. To identify this schema as defining repository resource *metadata*, it uses `ENABLE_HIERARCHY_RESMETADATA` as the value for parameter `enableHierarchy`. Resource contents (data) are defined by using value `ENABLE_HIERARCHY_CONTENTS` (the default value), instead.

The properties defined in Example 29–1 are the image height, width, color depth, title, and brief description.

Example 29–1 Register an XML Schema for Technical Photo Information

```
BEGIN
  DBMS_XMLSCHEMA.registerSchema(
    SCHEMAURL       => 'imagetechnique.xsd',
    SCHEMADOC       => '<xsd:schema targetNamespace="inamespace"
                                    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                                    xmlns:xdb="http://xmlns.oracle.com/xdb"
                                    xmlns="inamespace">
                          <xsd:element name="ImgTechMetadata"
                                       xdb:defaultTable="IMGTECHMETADATATABLE">
                            <xsd:complexType>
                              <xsd:sequence>
                                <xsd:element name="Height"      type="xsd:float"/>
                                <xsd:element name="Width"       type="xsd:float"/>
                                <xsd:element name="ColorDepth"  type="xsd:integer"/>
                                <xsd:element name="Title"       type="xsd:string"/>
                                <xsd:element name="Description" type="xsd:string"/>
                              </xsd:sequence>
                            </xsd:complexType>
                          </xsd:element>
                        </xsd:schema>',
    enableHierarchy => DBMS_XMLSCHEMA.ENABLE_HIERARCHY_RESMETADATA);
END;
/
```

The XML schema in Example 29–2 defines metadata used to categorize a photo image file: to describe its content or possible uses. This simple example defines a single, general property for classification, named `Category`.

Example 29–2 Register an XML Schema for Photo Categorization

```
BEGIN
  DBMS_XMLSCHEMA.registerSchema(
    SCHEMAURL       => 'imagecategories.xsd',
    SCHEMADOC       => '<xsd:schema targetNamespace="cnamespace"
                                    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                                    xmlns:xdb="http://xmlns.oracle.com/xdb"
                                    xmlns="cnamespace">
                          <xsd:element name="ImgCatMetadata"
                                       xdb:defaultTable="IMGCATMETADATATABLE">
                            <xsd:complexType>
                              <xsd:sequence>
                                <xsd:element name="Categories"
                                             type="CategoriesType"/>
                              </xsd:sequence>
                            </xsd:complexType>
                          </xsd:element>
                          <xsd:complexType name="CategoriesType">
                            <xsd:sequence>
                              <xsd:element name="Category" type="xsd:string"
                                           maxOccurs="unbounded"/>
                            </xsd:sequence>
                          </xsd:complexType>
                        </xsd:schema>',
    enableHierarchy => DBMS_XMLSCHEMA.ENABLE_HIERARCHY_RESMETADATA);
END;
/
```

Notice that there is nothing in the XML schema definitions of metadata that restrict that information to being associated with any particular kind of data. You are free to associate any type of metadata with any type of resource. And multiple types of metadata can be associated with the same resource.

Notice, too, that the XML schema does not, by itself, define its associated data as being metadata—it is the schema *registration* that makes this characterization, through `enableHierarchy` value `ENABLE_HIERARCHY_RESMETADATA`. If the same schema were registered instead with `enableHierarchy` value `ENABLE_HIERARCHY_CONTENTS` (the default value), then it would define not metadata for resources, but resource *contents* with the same information. The same XML schema cannot be registered more than once under the same name.

Note: XML schema-based user-defined metadata is stored as a `CLOB`, by default. You can store it as binary XML, instead, by setting the `OPTIONS` parameter for XML schema registration to `REGISTER_BINARYXML`.

Adding, Updating, and Deleting Resource Metadata

You can add, update, and delete user-defined resource metadata in the following ways:

- Use PL/SQL procedures in package `DBMS_XDB`:
  - `appendResourceMetadata` – add metadata to a resource
  - `updateResourceMetadata` – modify resource metadata
  - `deleteResourceMetadata` – delete specific metadata from a resource

  - `purgeResourceMetadata` – delete *all* metadata from a resource
- Use SQL DML statements `INSERT`, `UPDATE`, and `DELETE` to update the resource directly
- Use WebDAV protocol method `PROPPATCH`

You use SQL DM statements and WebDAV method `PROPPATCH` to update or delete metadata in the same way as you add metadata. If you supply a complete `Resource` element for one of these operations, then keep in mind that each resource metadata property must be a child (not just a descendant) of element `Resource`—if you want multiple metadata elements of the same kind, you must collect them as children of a single parent metadata element. The order among such top-level user-defined resource metadata properties is unimportant and is not necessarily maintained by Oracle XML DB.

The separate PL/SQL procedures in package `DBMS_XDB` are similar in their use. Each can be used with either XML schema-based or non-schema-based metadata. Some forms (signatures) of some of the procedures apply only to schema-based metadata. Procedures `appendResourceMetadata` and `deleteResourceMetadata` are illustrated here with examples.

> **See Also:** *Oracle Database PL/SQL Packages and Types Reference* for information about the procedures in PL/SQL package `DBMS_XDB`

Adding Metadata using APPENDRESOURCEMETADATA

You can use procedure `DBMS_XDB.appendResourceMetadata` to add user-defined metadata to resources.

Example 29–3 creates a photo resource and adds XML schema-based metadata of type `ImgTechMetadata` to it, recording the technical information about the photo.

Example 29–3 Add Metadata to a Resource – Technical Photo Information

```
DECLARE
  returnbool BOOLEAN;
BEGIN
  returnbool := DBMS_XDB.createResource(
                  '/public/horse_with_pig.jpg',
                  bfilename('MYDIR', 'horse_with_pig.jpg'));
  DBMS_XDB.appendResourceMetadata(
    '/public/horse_with_pig.jpg',
    XMLType('<i:ImgTechMetadata
                 xmlns:i="inamespace"
                 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                 xsi:schemaLocation="inamespace imagetechnique.xsd">
               <Height>1024</Height>
               <Width>768</Width>
               <ColorDepth>24</ColorDepth>
               <Title>Pig Riding Horse</Title>
               <Description>Picture of a pig riding a horse on the beach,
taken outside hotel window.</Description>
             </i:ImgTechMetadata>'));
END;
/
```

Example 29–4 adds metadata of type `ImgTechMetadata` to the same resource as Example 29–3, placing the photo in several user-defined content categories.

Example 29–4 Add Metadata to a Resource – Photo Content Categories

```
BEGIN
  DBMS_XDB.appendResourceMetadata(
    '/public/horse_with_pig.jpg',
    XMLType('<c:ImgCatMetadata
                 xmlns:c="cnamespace"
                 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                 xsi:schemaLocation="cnamespace imagecategories.xsd">
               <Categories>
                 <Category>Vacation</Category>
                 <Category>Animals</Category>
                 <Category>Humor</Category>
                 <Category>2005</Category>
               </Categories>
             </c:ImgCatMetadata>'));
END;
/

PL/SQL procedure successfully completed.

SELECT * FROM imgcatmetadatatable;

SYS_NC_ROWINFO$
--------------------------------------------------------------------------------
<c:ImgCatMetadata xmlns:c="cnamespace" xmlns:xsi="http://www.w3.org/2001/XMLSche
ma-instance" xsi:schemaLocation="cnamespace imagecategories.xsd">
  <Categories>
    <Category>Vacation</Category>
    <Category>Animals</Category>
    <Category>Humor</Category>
    <Category>2005</Category>
  </Categories>
</c:ImgCatMetadata>

1 row selected.
```

Deleting Metadata using DELETERESOURCEMETADATA

You can use procedure `DBMS_XDB.deleteResourceMetadata` to delete specific metadata associated with a resource. To delete *all* of the metadata associated with a resource, you can use procedure `DBMS_XDB.purgeResourceMetadata`.

Example 29–5 deletes the category metadata that was added to the photo resource in Example 29–4. By default, both the resource link (`REF`) to the metadata and the metadata table identified by that link are deleted. An optional parameter can be used to specify that only the link is to be deleted. The metadata table is then left as is but becomes unrelated to the resource. In this example, the default behavior is used.

Example 29–5 Delete Specific Metadata from a Resource

```
BEGIN
  DBMS_XDB.deleteResourceMetadata('/public/horse_with_pig.jpg',
                                  'cnamespace',
                                  'ImgCatMetadata');
END;
/

PL/SQL procedure successfully completed.

SELECT * FROM imgcatmetadatatable;
```

```
no rows selected
```

Adding Metadata using SQL DML

An alternative to using procedure `DBMS_XDB.appendResourceMetadata` to add, update, or delete resource metadata is to update the `RESOURCE_VIEW` directly using DML statements `INSERT` and `UPDATE`.

Adding resource metadata in this way is illustrated by Example 29–6. It shows how to accomplish the same thing as Example 29–3 by inserting the metadata directly into `RESOURCE_VIEW` using SQL statement `UPDATE`. Other SQL DML statements may be used similarly.

Example 29–6 Adding Metadata to a Resource using DML with RESOURCE_VIEW

```
UPDATE RESOURCE_VIEW
  SET RES =
        insertChildXML(
          RES,
          '/r:Resource',
          'c:ImgCatMetadata',
          XMLType('<c:ImgCatMetadata
                      xmlns:c="cnamespace"
                      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                      xsi:schemaLocation="cnamespace imagecategories.xsd">
                    <Categories>
                      <Category>Vacation</Category>
                      <Category>Animals</Category>
                      <Category>Humor</Category>
                      <Category>2005</Category>
                    </Categories>
                  </c:ImgCatMetadata>'),
          'xmlns:r="http://xmlns.oracle.com/xdb/XDBResource.xsd"
           xmlns:c="cnamespace"')
  WHERE equals_path(RES, '/public/horse_with_pig.jpg') = 1;
/

SELECT * FROM imgcatmetadatatable;

SYS_NC_ROWINFO$
--------------------------------------------------------------------------------
<c:ImgCatMetadata xmlns:c="cnamespace" xmlns:xsi="http://www.w3.org/2001/XMLSche
ma-instance" xsi:schemaLocation="cnamespace imagecategories.xsd">
  <Categories>
    <Category>Vacation</Category>
    <Category>Animals</Category>
    <Category>Humor</Category>
    <Category>2005</Category>
  </Categories>
</c:ImgCatMetadata>

1 row selected.
```

The following query extracts the inserted metadata using `RESOURCE_VIEW`, rather than directly using metadata table `imgcatmetadatatable`. (The result is shown here pretty-printed, for clarity.)

```
SELECT XMLQuery('declare namespace r
                   = "http://xmlns.oracle.com/xdb/XDBResource.xsd"; (: :)
```

```
                 declare namespace c
                   = "cnamespace"; (: :)
                 /r:Resource/c:ImgCatMetadata'
                PASSING RES RETURNING CONTENT)
  FROM RESOURCE_VIEW
  WHERE equals_path(RES, '/public/horse_with_pig.jpg') = 1;

XMLQUERY('DECLARENAMESPACER="HTTP://XMLNS.ORACLE.COM/XDB/XDBRESOURCE.XSD";(::)DE
--------------------------------------------------------------------------------
<c:ImgCatMetadata xmlns:c="cnamespace"
                  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                  xsi:schemaLocation="cnamespace imagecategories.xsd">
  <Categories>
    <Category>Vacation</Category>
    <Category>Animals</Category>
    <Category>Humor</Category>
    <Category>2005</Category>
  </Categories>
</c:ImgCatMetadata>

1 row selected.
```

Adding Metadata using WebDAV PROPPATCH

Another alternative to using procedure `DBMS_XDB.appendResourceMetadata` to add resource metadata is to use WebDAV method `PROPPATCH`. This is illustrated in Example 29–7. You can update and delete metadata similarly.

Example 29–7 shows how to accomplish the same thing as Example 29–4 by inserting the metadata using WebDAV method `PROPPATCH`. Using appropriate tools, your application creates such a `PROPPATCH` WebDAV request and sends it to the WebDAV server for processing.

To update user-defined metadata, you proceed in the same way. To *delete* user-defined metadata, the WebDAV request is similar, but it has `D:remove` in place of `D:set`.

Example 29–7 Adding Metadata using WebDAV PROPPATCH

```
PROPPATCH /public/horse_with_pig.jpg HTTP/1.1
Host: www.example.com
Content-Type: text/xml; charset="utf-8"
Content-Length: 609
Authorization: Basic dGRhZHhkYl9tZXRhOnRkYWR4ZGJfbWV0YQ==
Connection: close

<?xml version="1.0" encoding="utf-8" ?>
<D:propertyupdate xmlns:D="DAV:" xmlns:Z="http://www.w3.com/standards/z39.50/">
  <D:set>
    <D:prop>
      <c:ImgCatMetadata
          xmlns:c="cnamespace"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="cnamespace imagecategories.xsd">
        <Categories>
          <Category>Vacation</Category>
          <Category>Animals</Category>
          <Category>Humor</Category>
          <Category>2005</Category>
        </Categories>
      </c:ImgCatMetadata>
```

```
    </D:prop>
  </D:set>
</D:propertyupdate>
```

Querying XML Schema-Based Resource Metadata

When you register an XML schema using the `enableHierarchy` value `ENABLE_HIERARCHY_RESMETADATA`, an additional column, `RESID`, is added automatically to the `XMLType` tables used to store the metadata. This column stores the object identifier (OID) of the resource associated with the metadata. You can use column `RESID` when querying metadata, to join the metadata with the associated data.

You can query metadata in these ways:

- Query `RESOURCE_VIEW` for the metadata. For example:

```
SELECT count(*) FROM RESOURCE_VIEW
  WHERE
    XMLExists(
      'declare namespace r
         = "http://xmlns.oracle.com/xdb/XDBResource.xsd"; (: :)
       declare namespace c
         = "cnamespace"; (: :)
       /r:Resource/c:ImgCatMetadata/Categories/Category[text()="Vacation"]'
      PASSING RES);

  COUNT(*)
----------
         1

1 row selected.
```

- Query the XML schema-based table for the user-defined metadata directly, and join this metadata back to the resource table, identifying which resource to select. Use column `RESID` of the metadata table to do this. For example:

```
SELECT COUNT(*) FROM RESOURCE_VIEW rs, imgcatmetadatatable ct
  WHERE
    XMLExists(
      'declare namespace r
         = "http://xmlns.oracle.com/xdb/XDBResource.xsd"; (: :)
       declare namespace c
         = "cnamespace"; (: :)
       /r:Resource/c:ImgCatMetadata/Categories/Category'
      PASSING RES)
    AND rs.RESID = ct.RESID;

  COUNT(*)
----------
         1

1 row selected.
```

Oracle recommends querying for user-defined metadata directly, for performance reasons. Direct queries of the `RESOURCE_VIEW` alone *cannot be optimized* using XPath rewrite, because there is no way to determine whether or not target elements like `Category` are stored in the `CLOB` value or in an out-of-line table.

To improve performance further, create an index on each metadata column you intend to query.

Example 29–8 queries both kinds of photo resource metadata, retrieving the paths to the resources that are categorized as vacation photos and have the title "Pig Riding Horse".

Example 29–8 Query XML Schema-Based Resource Metadata

```
SELECT ANY_PATH
  FROM RESOURCE_VIEW rs, imgcatmetadatatable ct, imgtechmetadatatable tt
  WHERE XMLExists(
          'declare namespace r
             = "http://xmlns.oracle.com/xdb/XDBResource.xsd"; (: :)
           declare namespace c
             = "cnamespace"; (: :)
           /r:Resource/c:ImgCatMetadata/Categories/Category[text()="Vacation"]'
          PASSING RES)
    AND XMLExists(
          'declare namespace r
             = "http://xmlns.oracle.com/xdb/XDBResource.xsd"; (: :)
           declare namespace i
             = "inamespace"; (: :)
           /r:Resource/i:ImgTechMetadata/Title[text()="Pig Riding Horse"]'
          PASSING RES)
    AND rs.RESID = ct.RESID
    AND rs.RESID = tt.RESID;

ANY_PATH
--------------------------
/public/horse_with_pig.jpg

1 row selected.
```

XML Image Metadata from Binary Image Metadata

Previous sections of this chapter use a simple user-defined XML schema that defines technical image metadata, `imagetechnique.xsd`, to illustrate ways of adding and changing repository resource metadata. That simple XML schema is not intended to be realistic with respect to technical photographic image information.

However, most digital cameras include image metadata as part of the binary image files they produce, and Oracle *inter*Media, which is part of Oracle Database, provides tools for extracting and converting this binary metadata to XML. Oracle *inter*Media XML schemas are automatically registered with Oracle XML DB Repository to convert binary image metadata of the followings kinds to XML data:

- EXIF – Exchangeable Image File Format
- IPTC-NAA IIM – International Press Telecommunications Council-Newspaper Association of America Information Interchange Model
- XMP – Extensible Metadata Platform

EXIF is the metadata standard for digital still cameras. EXIF metadata is stored in TIFF and JPEG image files. IPTC and XMP metadata is commonly embedded in image files by desktop image-processing software.

See Also:

- *Oracle Multimedia User's Guide* for information about working with digital image metadata, including examples of extracting binary image metadata and converting it to XML
- *Oracle Multimedia Reference* for information about the XML schemas supported by Oracle *inter*Media for use with image metadata

Adding Non-Schema-Based Resource Metadata

You store user-defined resource metadata that is *not* schema-based as a `CLOB` instance under the `Resource` element of the associated resource. The default XML schema for a resource has a top-level element **`any`** (declared with `maxOccurs= "unbounded"`), which admits any valid XML data as part of the resource document. This metadata is stored in a `CLOB` column of the resource table.

The following skeleton shows the structure and position of non-schema-based resource metadata:

```
<Resource xmlns="http://xmlns.oracle.com/xdb/XDBResource.xsd"
  <Owner>DESELBY</Owner>
  ... <!-- other system-defined metadata -->
  <!-- contents of the resource>
  <Contents>
    ...
  </Contents>
  <!-- User-defined metadata (appearing within different namespace) -->
  <MyOwnMetadata xmlns="http://www.example.com/custommetadata">
    <MyElement1>value1</MyElement1>
    <MyElement2>value2</MyElement2>
  </MyOwnMetadata>
</Resource>
```

You can set and access non-schema-based resource metadata belonging to namespaces other than `XDBResource.xsd` by using any of the means described previously for accessing XML schema-based resource metadata.

Example 29–9 illustrates this for the case of SQL DML operations, adding user-defined metadata directly to the `<RESOURCE>` *document*. It shows how to add non-schema-based metadata to a resource using SQL DML.

Example 29–9 Add Non-Schema-Based Metadata to a Resource

```
DECLARE
  res BOOLEAN;
BEGIN
  res := DBMS_XDB.createResource('/public/NurseryRhyme.txt',
                                 bfilename('MYDIR',
                                           'tdadxdb-xdb_repos_meta-011.txt'),
                                 nls_charset_id('AL32UTF8'));
UPDATE RESOURCE_VIEW
  SET RES = insertChildXML(RES,
                           '/r:Resource',
                           'n:NurseryMetadata',
                           XMLType('<n:NurseryMetadata xmlns:n="nurserynamespace">
                                      <Author>Mother Goose</Author>
                                    <n:NurseryMetadata>'),
                           'xmlns:r="http://xmlns.oracle.com/xdb/XDBResource.xsd"
```

```
                            xmlns:n="nurserynamespace"')
  WHERE equals_path(RES, '/public/NurseryRhyme.txt') = 1;
END;
/

PL/SQL procedure successfully completed.


SELECT XMLSerialize(DOCUMENT rs.RES AS CLOB) FROM RESOURCE_VIEW rs
  WHERE equals_path(RES, '/public/NurseryRhyme.txt') = 1;

XMLSERIALIZE(DOCUMENTRS.RESASCLOB)
--------------------------------------------------------------------------------
<Resource xmlns="http://xmlns.oracle.com/xdb/XDBResource.xsd" Hidden="false" Inv
alid="false" Container="false" CustomRslv="false" VersionHistory="false" StickyR
ef="true">
  <CreationDate>2005-05-24T13:51:48.043234</CreationDate>
  <ModificationDate>2005-05-24T13:51:48.290144</ModificationDate>
  <DisplayName>NurseryRhyme.txt</DisplayName>
  <Language>en-US</Language>
  <CharacterSet>UTF-8</CharacterSet>
  <ContentType>text/plain</ContentType>
  <RefCount>1</RefCount>
  <ACL>
    <acl description="Public:All privileges to PUBLIC" xmlns="http://xmlns.oracl
e.com/xdb/acl.xsd" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:sch
emaLocation="http://xmlns.oracle.com/xdb/acl.xsd                           http:
//xmlns.oracle.com/xdb/acl.xsd" shared="true">
      <ace>
        <principal>PUBLIC</principal>
        <grant>true</grant>
        <privilege>
          <all/>
        </privilege>
      </ace>
    </acl>
  </ACL>
  <Owner>TDADXDB_META</Owner>
  <Creator>TDADXDB_META</Creator>
  <LastModifier>TDADXDB_META</LastModifier>
  <SchemaElement>http://xmlns.oracle.com/xdb/XDBSchema.xsd#text</SchemaElement>
  <Contents>
    <text>Mary had a little lamb
Its fleece was white as snow
and everywhere that Mary went
that lamb was sure to go
</text>
  </Contents>
  <n:NurseryMetadata xmlns:n="nurserynamespace">
    <Author xmlns="">Mother Goose</Author>
  </n:NurseryMetadata>
</Resource>

1 row selected.
```

PL/SQL Procedures Affecting Resource Metadata

The following PL/SQL procedures perform resource metadata operations:

- `DBMS_XMLSCHEMA.registerSchema` – Register an XML schema. Parameter `ENABLEHIERARCHY` affects resource metadata.
- `DBMS_XDBZ.enable_hierarchy` – Enable repository support for an `XMLType` table or view. Use parameter `HIERARCHY_TYPE` with a value of `DBMS_XDBZ.ENABLE_HIERARCHY_RESMETADATA` to enable resource metadata. This adds column `RESID` to track the resource associated with the metadata.
- `DBMS_XDBZ.disable_hierarchy` – Disable all repository support for an `XMLType` table or view.
- `DBMS_XDBZ.is_hierarchy_enabled` – Tests, using parameter `HIERARCHY_TYPE`, whether the specified type of hierarchy is currently enabled for the specified `XMLType` table or view. Value `DBMS_XDBZ.IS_ENABLED_RESMETADATA` for `HIERARCHY_TYPE` tests whether resource metadata is enabled.
- `DBMS_XDB.appendResourceMetadata` – Add metadata to a resource.
- `DBMS_XDB.deleteResourceMetadata` – Delete specified metadata from a resource.
- `DBMS_XDB.purgeResourceMetadata` – Delete all user-defined metadata from a resource. For schema-based resources, optional parameter `DELETE_OPTION` can be used to specify whether or not to delete the metadata information, in addition to unlinking it.
- `DBMS_XDB.updateResourceMetadata` – Update the metadata for a resource.

> **See Also:** *Oracle Database PL/SQL Packages and Types Reference* for detailed information about these PL/SQL procedures

30

Oracle XML DB Repository Events

This chapter describes repository events and how to use them. It contains these topics:

- Overview of Repository Events
- Possible Repository Events
- Repository Operations and Events
- Repository Event Handler Considerations
- Configuring Repository Events

Overview of Repository Events

You can use Oracle XML DB Repository to store and access data of any kind in the form of repository resources: files and folders. Repository resource operations include creating, deleting, locking, unlocking, rendering, linking, unlinking, placing under version control, checking in, checking out, unchecking out (reverting a checked out version), opening, and updating. You can access data in the repository from any application. Sometimes your application needs to perform certain actions whenever a particular repository operation occurs. For example, you might want to perform some move-to-wastebasket or other backup action whenever a resource is deleted.

Repository Events: Use Cases

The following are examples of cases where repository events can be used:

- *Wastebasket* – You can use an `UnLink` pre-event handler to effectively move a resource to a wastebasket instead of deleting it. Create a link in a wastebasket folder before removing the original link. The link in the wastebasket ensures that the resource is not removed. When you subsequently undelete a resource from the waste basket, the original link can be created again and the wastebasket link removed. The wastebasket link name can be different from the name of the link being removed because a resource at a certain path could be unlinked more than once from that path. The wastebasket would then have multiple links corresponding to that path, with different link properties and possibly pointing to different resources.
- *Categorization* – An application might categorize the resources it manages based on MIME type or other properties. It might keep track of GIF, text, and XML files by maintaining links to them from repository folders `/my-app/gif`, `/my-app/txt`, and `/my-app/xml`. Three post-event handlers could be used here: `LinkIn`, `UnlinkIn`, and `Update`. The `LinkIn` post-event handler would examine the resource and create a link in the appropriate category folder, if not already present. The `UnlinkIn` post-event handler would remove the link from the category

folder. The `Update` post-event handler would effectively move the resource from one category folder to another if its category changes.

Repository Events and Database Triggers

Repository events are reminiscent of database triggers, but they offer additional functionality. You cannot use a database trigger to let your application react to repository operations. A given repository operation can consist of multiple database operations on multiple underlying, internal tables. Because these underlying tables are internal to Oracle XML DB, you cannot easily map them to specific repository operations. For example, internal table `XDB$H_INDEX` might be updated by either a database update operation, if an ACL is changed, or a link operation. Even in cases where you might be able to accomplish the same thing using database triggers, you would not want to do that: A repository event is a higher-level abstraction than would be a set of database triggers on the underlying tables.

When a repository event occurs, information associated with the operation, such as the resource path used, can be passed to the corresponding event handler. Such information is not readily available using database triggers.

Repository events and database triggers can both be applied to XML data. You can use triggers on `XMLType` tables, for instance. However, if an `XMLType` table is also a repository table (hierarchy-enabled), then do not duplicate in an event handler any trigger code that applies to the table. Otherwise, that code is executed twice.

Repository Event Listeners and Event Handlers

Each repository operation is associated with one or more repository events. Your application can configure listeners for the events associated with resources it is concerned with. A repository **event listener** is a Java class or a PL/SQL package or object type. It comprises a set of PL/SQL procedures or Java methods, each of which is called an **event handler**. Each event handler processes a single event. A repository event listener can be configured for a particular resource or for the entire repository. A listener can be further restricted to apply only when a given node-existence precondition is met.

You associate a repository event listener with a resource by mapping a **resource configuration file** to the resource. You use PL/SQL package `DBMS_RESCONFIG` to manipulate resource configuration files, including associating them with the resources they configure. In particular, PL/SQL function `DBMS_RESCONFIG.getListeners` lists all event listeners for a given resource.

Repository Event Configuration

A given resource can be configured by multiple resource configuration files. These are stored in a **resource configuration list**, and they are processed in list order. The repository as a whole can also be configured by multiple resource configuration files. The repository itself has a resource configuration list. Event handling that is configured for the repository as a whole takes effect before any resource-specific event handling. All applicable repository-wide events are processed before any resource-specific events.

A given resource configuration file can define multiple event listeners for the resources it configures, and each event listener can define multiple event handlers.

See Also:

- "Configuring Repository Events" on page 30-8
- "Resource Configuration Files Configure a Resource" on page 22-1 for general information about resource configuration and resource configuration lists

Possible Repository Events

A rendering operation is associated with a single repository event. Except for rendering, all repository operations are associated with one or more *pairs* of events. For example, a resource creation is associated with three pairs of events, with the events occurring in this order:

1. Pre-creation event
2. Post-creation event
3. Pre-link-in event
4. Pre-link-to event
5. Post-link-to event
6. Post-link-in event

Table 30–1 lists the events associated with each repository operation. Their order is indicated in the handler columns.

Table 30–1 Predefined Repository Events

Repository Event Type	Description	Pre Handler Execution	Post Handler Execution
`Render`	A `Render` event occurs only for *file* resources, never for folder resources. Occurs when resource contents are accessed using any of the following: ■ Protocols ■ `XDBURIType` methods `getCLOB()`, `getBLOB()`, and `getXML()` Does *not* occur when resource contents are accessed using any of the following: ■ `SELECT ... FROM RESOURCE_VIEW` ■ `XDBURIType` method `getResource()` Only one handler for a `Render` event can set the rendered output. The first handler to call `setRenderStream` or `setRenderPath` controls the rendering.	N/A	N/A
`Create`	Occurs when a resource is created. The pre and post handlers executed are those defined on the folder of the new resource.	After pre-parsing, after validating the parent resource ACL and locks, and before assigning default values to undefined properties.	After inserting the resource into the system resource table.
`Delete`	Occurs when the resource and its contents are removed from disk, that is, when the resource `REF` count is zero (0).	After validating the resource ACL and locks and before removing the resource from disk.	After removing the resource and its contents from disk and after touching the parent folder to update its last modifier and modification time.

Table 30–1 (Cont.) Predefined Repository Events

Repository Event Type	Description	Pre Handler Execution	Post Handler Execution
`Update`	Occurs when a resource is updated on disk.	After validating the resource ACL and locks and before updating the last modifier and modification time.	After writing the resource to disk.
`Lock`	Occurs during a lock-resource operation.	After validating the resource ACL and locks and before creating the new lock on the resource.	After creating the new lock.
`Unlock`	Occurs during an unlock-resource operation.	After validating the resource ACL and delete token.	After removing the lock.
`LinkIn`	Occurs before a `LinkTo` event during a link operation. The event target is the folder in which the link is created. Always accompanied by a `LinkTo` event.	After validating the resource ACL and locks and before creating the link.	After executing `LinkTo` post handler.
`LinkTo`	Occurs after a `LinkIn` event during a link operation. The event target is the resource that is the link destination.	After executing `LinkIn` pre handler and before creating the link.	After creating the link and after updating the last modifier and modification time of the parent folder.
`UnLinkIn`	Occurs before an `UnlinkFrom` event during an unlink operation. Always accompanied by an `UnlinkFrom` event.	After validating the resource ACL and locks and before removing the link.	After executing the `UnlinkFrom` post handler.
`UnlinkFrom`	Occurs after an `UnlinkIn` event during an unlink operation.	After executing the `UnlinkIn` pre handler.	After removing the link.
`CheckIn`	Occurs during check-in of a resource.	After validating the resource ACL and locks and after verifying that the resource is version-controlled and has been checked out.	After checking in the resource.
`CheckOut`	Occurs during check-out of a resource.	After validating the resource ACL and locks and after verifying that the resource is version-controlled and is not already checked out.	After checking out the resource.
`UncheckOut`	Occurs during uncheck-out of a resource.	Before removing the record that the resource is checked out.	After unchecking out the resource.
`VersionControl`	Occurs when a version history is created for a resource. Note: You can call `DBMS_XDB.MakeVersioned()` multiple times, but the version history is created only at the first call. Subsequent calls have no effect, so no `VersionControl` event occurs.	Before creating the version history for the resource.	After creating the first version of the resource.

For simplicity, the presentation in this chapter generally treats both members of a repository event pair together, referring, for example, to the `LinkIn` event type as shorthand for the pre-link-in and post-link-in event types. For the same reason, the event-type names used here are derived from the Java interface `XDBRepositoryEventListener` by dropping the prefixes `handlePre` and `handlePost`.

> **See Also:** *Oracle Database PL/SQL Packages and Types Reference* for the PL/SQL repository event types

Repository Operations and Events

The same repository event can occur with different Oracle XML DB Repository operations, and a given repository operation can produce more than one repository event. Table 30–2 lists the events that are associated with each repository operation. See Table 30–1 for the event order when multiple repository events occur for the same operations.

Table 30–2 Oracle XML DB Repository Operations and Events

Operation	Repository Events Occurring
Get binary representation of resource contents by path name	`Render`
Get XML representation of resource contents by path name	`Render`
Create or update a resource	If the resource already exists: `Create`, `LinkIn`, `LinkTo` If resource doe not yet exist (HTTP and FTP only): `Update`
Create a folder	`Create`, `LinkIn`, `LinkTo`
Create a link to an existing resource	`LinkIn` on the folder containing the link target, `LinkTo` on the target resource to be linked
Unlink a file resource or an empty folder resource. (Decrement `RefCount`, and if it becomes zero then delete the resource from disk.)	`UnlinkIn`, `UnlinkFrom`, and, if `RefCount` is zero, `Delete`
Forcibly delete a folder and its contents	Recursively produce events for unlinking a resource. Folder child resources are deleted recursively, then the folder is deleted.
Forcibly remove all links to a resource	Produce unlinking events for each link removed.
Update the contents, properties, or ACL of a resource by path name	`Update`
Put a depth-zero WebDAV lock on a resource	`Lock`
Remove a depth-zero WebDAV lock from a resource	`Lock`
Rename (move) a resource	`LinkIn` and `LinkTo` on the new location, `UnlinkIn` and `UnlinkFrom` on the old location
Copy a resource	`Create`, `LinkIn`, and `LinkTo` on the new location
Check out a resource	`CheckOut`
Check in a resource	`CheckIn`
Place a resource under version control	`VersionControl`
Uncheck out a resource	`UncheckOut`

All operations listed in Table 30–2 are atomic, except for these:

- Forced deletion of a folder and its contents
- Update of resource properties by path name using HTTP (only)
- Copy of a folder using HTTP (only)

> **See Also:** Table 21–3, " Accessing Oracle XML DB Repository: API Options" on page 21-15 for information on accessing resources using APIs and protocols

Repository Event Handler Considerations

This section mentions some things to keep in mind when you define handlers for Oracle XML DB Repository events.

- In any handler: Do not use `COMMIT`, `ROLLBACK`, or data definition language (DDL) statements in a handler. Do not call PL/SQL functions or procedures, such as `DBMS_XMLSCHEMA.registerSchema`, that behave similarly to DDL statements. In a `Render` handler: Do not use data manipulation language (DML) statements.

  To work around these restrictions, a handler can use such statements inside an autonomous transaction, but it must ensure that lock conflicts cannot arise.
- In a `Render` handler, do not close an output stream. (You can append to a stream.)
- Do *not* use modifier methods from class `XDBResource` in a handler, unless it is a `Pre-Create` or `Pre-Update` handler. Do *not* use method `XDBResource.save()` in any handler.
- Oracle recommends that you develop only safe repository event handlers. In particular:
  - Write only resource properties that are in namespaces owned by your application, never in the `xdb` namespace.
  - Do not delete a resource while it is being created.
- A repository event handler is passed an `XDBRepositoryEvent` object, which exists only during the current SQL statement or protocol operation. You can use PL/SQL procedures and Java methods on this object to obtain information about the resource, the event, and the associated event handlers.
- When an event handler performs operations that cause other repository events to occur, those cascading events occur immediately. They are not queued to occur after the handlers for the current event are finished. Each event thus occurs in the context of its corresponding operation.
- Repository event handlers are called synchronously. They are executed in the process, session, and transaction context of the corresponding operation. However, handlers can use Oracle Streams Advanced Queuing (AQ) to queue repository events that are then handled asynchronously by some other process.
- Because a repository event handler is executed in the transaction context of its corresponding operation, any locks acquired by that operation, or by other operations run previously in the transaction, are still active. An event handler must not start a separate session or transaction that tries to acquire such a lock. Otherwise, the handler hangs.
- Repository event handlers are called in the order that they appear in a resource configuration file. If preconditions are defined for a resource configuration, then only those handlers are called for which the precondition is satisfied.
- Although handlers are called in the order they are defined in a configuration file, avoid letting your code depend upon this. If the user who is current when a handler is invoked has privilege `write-config`, then the handler invocation order could be changed inside an executing handler.
- The entire list of handlers applicable to a given repository event occurrence is determined before any of the handlers is invoked. This means, in particular, that the precondition for each handler is evaluated before any handlers are invoked.
- The following considerations apply to *error handling* for repository events:

  - A pre-operation event handler is never invoked if access checks for the operation fail.
  - All handlers for a given event are checked before any of them are called. If any of them is not usable (for example, no longer exists), then *none* of them are called.
  - If an error is raised during event handling, then other, subsequent event handlers are not invoked for the same SQL statement or protocol operation. The current statement or operation is canceled and all of its changes are rolled back.
- The following considerations apply to *resource security* for repository events:
  - An event handler can have invoker's rights or definer rights. You specify the execution rights of a PL/SQL package when you create the package. You specify the execution rights of Java classes when you load them into the database using the `loadjava` utility. If you specify invoker's rights, but a given handler is not configured for invoker's rights, then an insufficient-privilege error is raised.
  - Within an event handler, the current user privileges, whether obtained by invoker or definer rights, are determined in detail for a given resource by its ACL. These privileges determine what the handler can do with the resource. For example, if the current user has privileges `read-properties` and `read-contents` for a particular resource, then an event handler can read that resource.
- The following considerations apply to repository events for *linking* and *unlinking*:
  - After creating a link to a resource, if you want any resource configuration files of the parent folder to also apply to the linked resource, then use procedure `DBMS_RESCONFIG.appendResConfig` to add the configuration files to the linked resource. You can invoke this procedure from a `Post-LinkTo` event handler for the linked resource.
  - After unlinking a resource, if you want to remove any such resource configuration files added when linking, then use procedure `DBMS_RESCONFIG.deleteResConfig` to remove them from the unlinked resource. You can invoke this procedure from a `Post-UnlinkFrom` event handler for the unlinked resource.
- Do not define handlers for events on folder `/sys/schemas` or on resources under this folder. Events do not occur for any such resources, so such event handlers are ignored. This implies that XML schema operations that affect the repository (registration, deletion, and so on) do not produce events.

See Also:

- *Oracle Database PL/SQL Packages and Types Reference* for information about PL/SQL functions and procedures for manipulating repository events
- *Oracle Database XML Java API Reference*, classes `XDBRepositoryEvent` and `XDBEvent` for information about Java methods for manipulating repository events
- "Configuring Repository Events" on page 30-8 for information about defining repository event handlers with invoker's rights

Configuring Repository Events

You configure event treatment for Oracle XML DB Repository resources as you would configure any other treatment of repository resources—see "Configuring a Resource" on page 22-2.

By default, repository events are enabled, but you can disable them by setting parameter `XML_DB_EVENTS` to `DISABLE`. To disable repository events at the session level, use the following SQL*Plus command. You must have role `XDBADMIN` to do this.

```
ALTER SESSION SET XML_DB_EVENTS = DISABLE;
```

To disable repository events at the system level, use the following SQL*Plus command, and then restart your database. Repository events are disabled for subsequent sessions. You must have privilege `ALTER SYSTEM` to do this.

```
ALTER SYSTEM SET XML_DB_EVENTS = DISABLE;
```

To enable repository events again, set the value of `XML_DB_EVENTS` to `ENABLE`.

The rest of this section describes the resource configuration file that you use as a resource to configure event processing for other resources.

A resource configuration file is an XML file that conforms to the XML schema `XDBResConfig.xsd`, which is accessible in Oracle XML DB Repository at path `/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/XDBResConfig.xsd`. You use element `event-listeners`, child of element `ResConfig`, to configure repository event handling.

> **See Also:** Chapter 22, "Configuring Oracle XML DB Repository" for general information about configuring repository resources

Configuration Element event-listeners

Each resource configuration file can have one `event-listeners` element, as a child of element `ResConfig`. This configures all event handling for the target resource. If the resource configuration file applies to the entire repository, not to a particular resource, then it defines event handling for all resources in the repository.

Element `event-listeners` has the following optional attributes:

- `set-invoker` – Set this to `true` to if the resource configuration defines one or more repository event handlers to have invoker's rights. The default value is `false`, meaning that definer rights are used.

  To define an invoker-rights repository event handler, you must have database role `XDB_SET_INVOKER`. This role is granted to `DBA`, but not to `XDBADMIN`. Role `XDB_SET_INVOKER` is checked only when a resource configuration file is created or updated. Only attribute `set-invoker`, not role `XDB_SET_INVOKER`, is checked at run time to ensure sufficient privilege.

  > **See Also:** "Repository Event Handler Considerations" on page 30-6 for information about insufficient-privilege errors

- `default-schema` – The default schema value, used for listeners for which no `schema` element is defined.
- `default-language` –The default language value, used for listeners for which no `language` element is defined.

Element `event-listeners` has a sequence of `listener` elements as children. These configure individual repository event listeners. The listeners are processed at run time in the order of the `listener` elements.

Configuration Element listener

Each `listener` element has the following child elements. All of these are optional except `source`, and they can appear in any order (their order is irrelevant).

- `description` – Description of the listener.
- `schema` – Database schema for the Java or PL/SQL implementation of the repository event handlers. If neither this nor `default-schema` is defined, then an error is raised.
- `source` (required) – Name of the Java class, PL/SQL package, or object type that provides the handler methods. Java class names must be qualified with a package name. Use an empty `source` element to indicate that the repository event handlers are standalone PL/SQL stored procedures.
- `language` – Implementation language of the listener class (Java) or package (PL/SQL). If neither this nor `default-language` is defined, then an error is raised.
- `pre-condition` – Precondition to be met for any repository event handlers in this listener to be executed. This is identical to the `pre-condition` child of general resource configuration element `configuration` – see "Configuration Element defaultChildConfig" on page 22-3.
- `events` – Sequence of unique repository event type names: `Render`, `Pre-Create`, and so on. Only handlers for repository events of these types are enabled for the listener. See "Possible Repository Events" on page 30-3 for the list of possible repository event types. If element `events` is not present, then handlers of repository events of all types are enabled for the listener, which can be wasteful. Provide element `events` to eliminate handler invocations for insignificant repository events.

Repository Events Configuration Examples

Example 30–1 shows the content of a resource configuration file that defines two listeners. Each listener defines handlers for repository events of types `Post-LinkIn`, `Post-UnlinkIn`, and `Post-Update`. It defines preconditions, the default language (Java) and default database schema.

Example 30–1 Resource Configuration File for Java Event Listeners with Preconditions

```
<ResConfig xmlns="http://xmlns.oracle.com/xdb/XDBResConfig.xsd"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="http://xmlns.oracle.com/xdb/XDBResConfig.xsd
                               http://xmlns.oracle.com/xdb/XDBResConfig.xsd">
  <event-listeners default-language="Java" default-schema="IFS">
    <listener>
      <description>Category application</description>
      <schema>CM</schema>
      <source>oracle.cm.category</source>
      <events>
        <Post-LinkIn/>
        <Post-UnlinkIn/>
        <Post-Update/>
      </events>
```

```
      <pre-condition>
        <existsNode>
          <XPath>/Resource[ContentType="image/gif"]</XPath>
        </existsNode>
      </pre-condition>
    </listener>
    <listener>
      <description>Check quota</description>
      <source>oracle.ifs.quota</source>
      <events>
        <Post-LinkIn/>
        <Post-UnlinkIn/>
        <Post-Update/>
      </events>
      <pre-condition>
        <existsNode>
          <XPath>r:/Resource/[ns:type="ifs-file"]</XPath>
          <namespace>xmlns:r="http://xmlns.oracle.com/xdb/XDBResource.xsd"
                     xmlns:ns="http://foo.xsd"
          </namespace>
        </existsNode>
      </pre-condition>
    </listener>
  </event-listeners>
  <defaultChildConfig>
    <configuration>
      <path>/sys/xdb/resconfig/user_rc.xml</path>
    </configuration>
  </defaultChildConfig>
  <applicationData>
    <foo:data xmlns:foo="http://foo.xsd">
      <foo:item1>1234</foo:item1>
    </foo:data>
  </applicationData>
</ResConfig>
```

The implementation of the handlers of the first listener is in Java class `oracle.cm.quota` defined in database schema `CM`. These handlers are invoked only for events on resources of `ContentType image/gif`.

The implementation of the handlers of the second listener is in Java class `oracle.ifs.quota` defined in database schema `IFS` (the default schema for this resource configuration file). These handlers are invoked only for events on resources of type `ifs-file` in namespace `http://foo.xsd`.

> **See Also:** "Configuration Element defaultChildConfig" on page 22-3 for a description of elements `defaultChildConfig` and `applicationData`

As a simple end-to-end illustration, suppose that an application needs to categorize the resources in folder `/public/res-app` according to their MIME types. It creates links to resources in folders `/public/app/XML-TXT`, `/public/app/IMG`, and `/public/app/FOLDER`, depending on whether the resource MIME type is `text/xml`, `image/gif`, or `application/octet-stream`, respectively. This is illustrated in Example 30–2, Example 30–3, and Example 30–5.

Example 30–2 shows the PL/SQL code to create the configuration file for this categorization illustration. It defines a single listener that handles events of types

`Pre-UnlinkIn` and `Post-LinkIn`. It explicitly defines the language (PL/SQL) and database schema. No preconditions are defined.

Example 30–2 Resource Configuration File for PL/SQL Event Listeners with No Preconditions

```
DECLARE
  b BOOLEAN := FALSE;
BEGIN
  b := DBMS_XDB.createFolder('/public/resconfig');
  b := DBMS_XDB.createResource(
         '/public/resconfig/appcatg-rc1.xml',
         '<ResConfig xmlns="http://xmlns.oracle.com/xdb/XDBResConfig.xsd"
                     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                     xsi:schemaLocation="http://xmlns.oracle.com/xdb/XDBResConfig.xsd
                                         http://xmlns.oracle.com/xdb/XDBResConfig.xsd">
            <event-listeners>
              <listener>
                <description>Category application</description>
                <schema>APPCATGUSER1</schema>
                <source>APPCATG_EVT_PKG1</source>
                <language>PL/SQL</language>
                <events>
                  <Pre-UnlinkIn/>
                  <Post-LinkIn/>
                </events>
              </listener>
            </event-listeners>
            <defaultChildConfig>
              <configuration>
                <path>/public/resconfig/appcatg-rc1.xml</path>
              </configuration>
            </defaultChildConfig>
          </ResConfig>',
         'http://xmlns.oracle.com/xdb/XDBResConfig.xsd',
         'ResConfig');
END;
/
BEGIN
  DBMS_RESCONFIG.appendResConfig('/public/res-app',
                                 '/public/resconfig/appcatg-rc1.xml',
                                 DBMS_RESCONFIG.APPEND_RECURSIVE);
END;
/
```

Example 30–3 shows the PL/SQL code that implements the event handlers that are configured in Example 30–2. The `Post-LinkIn` event handler creates a link to the `eventObject` resource in one of the folders `/public/app/XML-TXT`, `/public/app/IMG`, and `/public/app/FOLDER`, depending on the resource MIME type. The `Pre-UnlinkIn` event handler deletes the links that are created by the `Post-LinkIn` event handler.

Example 30–3 PL/SQL Code Implementing Event Listeners

```
CREATE OR REPLACE PACKAGE appcatg_evt_pkg1 AS

  PROCEDURE handlePreUnlinkIn (eventObject DBMS_XEVENT.XDBRepositoryEvent);
  PROCEDURE handlePostLinkIn (eventObject DBMS_XEVENT.XDBRepositoryEvent);

END;
/
```

```
CREATE OR REPLACE PACKAGE BODY appcatg_evt_pkg1 AS

  PROCEDURE handlePreUnlinkIn (eventObject DBMS_XEVENT.XDBRepositoryEvent) AS
      XDBResourceObj DBMS_XDBRESOURCE.XDBResource;
      ResDisplayName VARCHAR2(100);
      ResPath        VARCHAR2(1000);
      ResOwner       VARCHAR2(1000);
      ResDeletedBy   VARCHAR2(1000);
      XDBPathobj     DBMS_XEVENT.XDBPath;
      XDBEventobj    DBMS_XEVENT.XDBEvent;
      SeqChar        VARCHAR2(1000);
      LinkName       VARCHAR2(10000);
      ResType        VARCHAR2(100);
      LinkFolder     VARCHAR2(100);
    BEGIN
      XDBResourceObj := DBMS_XEVENT.getResource(eventObject);
      ResDisplayName := DBMS_XDBRESOURCE.getDisplayName(XDBResourceObj);
      ResOwner       := DBMS_XDBRESOURCE.getOwner(XDBResourceObj);
      XDBPathobj     := DBMS_XEVENT.getPath(eventObject);
      ResPath        := DBMS_XEVENT.getName(XDBPathObj);
      XDBEventobj    := DBMS_XEVENT.getXDBEvent(eventObject);
      ResDeletedBy   := DBMS_XEVENT.getCurrentUser(XDBEventobj);
      BEGIN
        SELECT XMLCast(
                 XMLQuery(
                   'declare namespace ns = "http://xmlns.oracle.com/xdb/XDBResource.xsd";
                    /ns:Resource/ns:ContentType'
                   PASSING r.RES RETURNING CONTENT) AS VARCHAR2(100))
          INTO ResType
          FROM PATH_VIEW r WHERE r.PATH=ResPath;
          EXCEPTION WHEN OTHERS THEN NULL;
      END;
      IF ResType = 'text/xml' THEN LinkFolder := '/public/app/XML-TXT/';
      END IF;
      IF ResType = 'image/gif' THEN LinkFolder := '/public/app/IMG/';
      END IF;
      IF ResType = 'application/octet-stream' THEN LinkFolder := '/public/app/FOLDER/';
      END IF;
      DBMS_XDB.deleteResource(LinkFolder || ResDisplayName);
  END;

  PROCEDURE handlePostLinkIn (eventObject DBMS_XEVENT.XDBRepositoryEvent) AS
      XDBResourceObj DBMS_XDBRESOURCE.XDBResource;
      ResDisplayName VARCHAR2(100);
      ResPath        VARCHAR2(1000);
      ResOwner       VARCHAR2(1000);
      ResDeletedBy   VARCHAR2(1000);
      XDBPathobj     DBMS_XEVENT.XDBPath;
      XDBEventobj    DBMS_XEVENT.XDBEvent;
      SeqChar        VARCHAR2(1000);
      LinkName       VARCHAR2(10000);
      ResType        VARCHAR2(100);
      LinkFolder     VARCHAR2(100);
    BEGIN
      XDBResourceObj := DBMS_XEVENT.getResource(eventObject);
      ResDisplayName := DBMS_XDBRESOURCE.getDisplayName(XDBResourceObj);
      ResOwner       := DBMS_XDBRESOURCE.getOwner(XDBResourceObj);
      XDBPathobj     := DBMS_XEVENT.getPath(eventObject);
      ResPath        := DBMS_XEVENT.getName(XDBPathObj);
      XDBEventobj    := DBMS_XEVENT.getXDBEvent(eventObject);
```

```
    ResDeletedBy   := DBMS_XEVENT.getCurrentUser(XDBEventobj);
    SELECT XMLCast(
             XMLQuery(
               'declare namespace ns = "http://xmlns.oracle.com/xdb/XDBResource.xsd";
                /ns:Resource/ns:ContentType'
               PASSING r.RES RETURNING CONTENT) AS VARCHAR2(100))
      INTO ResType
      FROM PATH_VIEW r WHERE r.PATH=ResPath;
    IF ResType = 'text/xml' THEN LinkFolder := '/public/app/XML-TXT';
    END IF;
    IF ResType = 'image/gif' THEN LinkFolder := '/public/app/IMG';
    END IF;
    IF ResType = 'application/octet-stream' THEN LinkFolder := '/public/app/FOLDER';
    END IF;
    DBMS_XDB.link(ResPath, LinkFolder, ResDisplayName);
  END;
END;
/
```

See Also:

- *Oracle Database PL/SQL Packages and Types Reference* for information about PL/SQL package DBMS_XDBRESOURCE
- *Oracle Database PL/SQL Packages and Types Reference* for information about PL/SQL package DBMS_XEVENT

A Java example would be configured the same as in Example 30–2, with the exception of these two lines, which would replace the elements with the same names in Example 30–2:

```
<source>category</source>
<language>Java</language>
```

Example 30–4 shows the Java code that implements the event handlers. The logic is identical to that in Example 30–3.

Example 30–4 Java Code Implementing Event Listeners

```
import oracle.xdb.event.*;
import oracle.xdb.spi.*;
import java.sql.*;
import java.io.*;
import java.net.*;
import oracle.jdbc.*;
import oracle.sql.*;
import oracle.xdb.XMLType;
import oracle.xdb.dom.*;

public class category
extends oracle.xdb.event.XDBBasicEventListener
{
  public Connection connectToDB() throws java.sql.SQLException
  {
    try
    {
      String strUrl="jdbc:oracle:kprb:";
      String strUname="appcatguser1";
      String strPwd="appcatguser1 ";
      Connection conn=null;
```

```
    OraclePreparedStatement stmt=null;
    DriverManager.registerDriver(new oracle.jdbc.OracleDriver());
    conn = DriverManager.getConnection(strUrl, strUname, strPwd);
    return conn;
  }
  catch(Exception e1)
  {
    System.out.println("Exception in connectToDB java function");
    System.out.println("e1:" + e1.toString());
    return null;
  }
}
public void handlePostLinkIn (XDBRepositoryEvent eventObject)
{
  XDBPath objXDBPath = null;
  String strPathName="";
  objXDBPath = eventObject.getPath();
  strPathName = objXDBPath.getName();
  XDBResource objXDBResource1;
  objXDBResource1 = eventObject.getResource();
  String textResDisplayName = objXDBResource1.getDisplayName();
  String resType = objXDBResource1.getContentType();
  String linkFolder="";
  System.out.println("resType" + resType+"sumit");
  System.out.println("strPathName:" + strPathName);
  System.out.println("textResDisplayName:" + textResDisplayName);
  if (resType.equals("text/xml")) linkFolder = "/public/app/XML-TXT/";
  else if (resType.equals("image/gif")) linkFolder = "/public/app/IMG/";
  else if (resType.equals("application/octet-stream"))
          linkFolder = "/public/app/FOLDER/";
  System.out.println("linkFolder:" + linkFolder);
  try
  {
    Connection con1 = connectToDB();
    OraclePreparedStatement stmt=null;
    stmt = (OraclePreparedStatement)con1.prepareStatement(
             "CALL DBMS_XDB.link(?,?,?)");
    stmt.setString(1,strPathName);
    stmt.setString(2,linkFolder);
    stmt.setString(3,textResDisplayName);
    stmt.execute();
    stmt.close();
    con1.close();
  }
  catch(java.sql.SQLException ej1)
  {
    System.out.println("ej1:" + ej1.toString());
  }

/* Make sure the link is not in the category folders.
   Then check the target resource's mime type and create a link
   in the appropriate category folder. */
}
public void handlePreUnlinkIn (XDBRepositoryEvent eventObject)
{
  XDBPath objXDBPath = null;
  String strPathName="";
  objXDBPath = eventObject.getPath();
  strPathName = objXDBPath.getName();
  XDBResource objXDBResource1;
```

```
    objXDBResource1 = eventObject.getResource();
    String textResDisplayName = objXDBResource1.getDisplayName();
    String resType = objXDBResource1.getContentType();
    String linkFolder="";
    if (resType.equals("text/xml")) linkFolder = "/public/app/XML-TXT/";
    else if (resType.equals("image/gif")) linkFolder = "/public/app/IMG/";
    else if (resType.equals("application/octet-stream"))
            linkFolder = "/public/app/FOLDER/";
    try
    {
      Connection con1 = connectToDB();
      OraclePreparedStatement stmt=null;
      stmt = (OraclePreparedStatement)con1.prepareStatement(
               "CALL DBMS_XDB.deleteResource(?)");
      stmt.setString(1,linkFolder+textResDisplayName);
      stmt.execute();
      stmt.close();
      con1.close();
    }
    catch(java.sql.SQLException ej1)
    {
      System.out.println("ej1:" + ej1.toString());
    }
  }
}
```

Example 30–5 demonstrates the invocation of the event handlers that are implemented in Example 30–3 or Example 30–4.

Example 30–5 Invoking Event Handlers

```
DECLARE
  ret BOOLEAN;
BEGIN
  ret := DBMS_XDB.createResource('/public/res-app/res1.xml',
                                 '<name>TestForEventType-1</name>');
END;
/
DECLARE
  b BOOLEAN := FALSE;
  dummy_data CLOB := 'AAA';
BEGIN
  b := DBMS_XDB.createResource('/public/res-app/res2.gif', dummy_data);
END;
/
DECLARE
  b BOOLEAN := FALSE;
  dummy_data CLOB := 'AAA';
BEGIN
  b := DBMS_XDB.createFolder('/public/res-app/res-appfolder1');
END;

SELECT PATH FROM PATH_VIEW WHERE PATH LIKE '/public/app/%' ORDER BY PATH;

PATH
---------------------------------
/public/app/FOLDER
/public/app/FOLDER/res-appfolder1
/public/app/IMG
/public/app/IMG/res2.gif
```

```
/public/app/XML-TXT
/public/app/XML-TXT/res1.xml

6 rows selected.

-- Delete the /res-app resources. The /app resources are deleted also.
EXEC DBMS_XDB.deleteResource('/public/res-app/res2.gif');
EXEC DBMS_XDB.deleteResource('/public/res-app/res1.xml');
EXEC DBMS_XDB.deleteResource('/public/res-app/res-appfolder1');

SELECT PATH FROM PATH_VIEW WHERE PATH LIKE '/public/app/%' ORDER BY PATH;

PATH
-------------------
/public/app/FOLDER
/public/app/IMG
/public/app/XML-TXT

3 rows selected.
```

31

Using Oracle XML DB Content Connector

This chapter describes how to use Oracle XML DB Content Connector to access Oracle XML DB Repository.

Oracle XML DB Content Connector implements Content Repository API for Java (sometimes referred to as JCR), a Java API standard developed by the Java community as JSR-170.

This chapter contains these topics:

- Overview of JCR and Oracle XML DB Content Connector
- How Oracle XML DB Repository Is Exposed in JCR
- How to Use Oracle XML DB Content Connector
- Using XML Schemas with JCR

Overview of JCR and Oracle XML DB Content Connector

This section contains the following topics:

- About the Content Repository API for Java (JCR)
- About Oracle XML DB Content Connector

About the Content Repository API for Java (JCR)

JCR 1.0 defines a standard Java API for applications to interact with content repositories.

> **See Also:** Java Community Process, "Content Repository for Java technology API", `http://jcp.org/en/jsr/detail?id=170`. Chapter 4 of the JSR-170 specification provides a concise introduction to JCR 1.0

JCR models the data in a content repository as a tree of nodes. Each node may have one or more child nodes. Every node has exactly one parent node, except for the root node, which has no parent.

In addition to child nodes, a node may also have one or more properties. A property is a simple name/value pair. For example, a node representing a particular file in the content repository has a property named `jcr:created` whose value is the date the file was created.

Each property has a property type. For example, the `jcr:created` property has the `DATE` property type, requiring its value to be a valid date/time.

Similarly, each node has a node type. For example, a node representing a file has node type `nt:file`. The node type controls what child nodes and properties the node may have or must have. For example, all nodes of type `nt:file` must have a `jcr:created` property.

Because nodes and properties are named, they can be addressed by path. JCR supports both absolute and relative paths. For example, the absolute path

```
/My Documents/pictures/puppy.jpg/jcr:created
```

resolves to the `jcr:created` property of file `puppy.jpg`. This property can also be addressed relative to the `My Documents` folder by the following relative path:

```
pictures/puppy.jpg/jcr:created
```

Node and property names can be namespace qualified. Like XML, JCR uses colon-delimited namespace prefixes to express namespace-qualified names, for example, `jcr:created`. Unlike XML, JCR records the namespace prefix-to-URI mappings in a repository-wide namespace registry, which, for example, maps the `jcr` prefix to the URI `http://www.jcp.org/jcr/1.0`.

About Oracle XML DB Content Connector

Oracle XML DB Content Connector lets you access Oracle XML DB Repository using the JCR 1.0 Java API. Your applications can run either in a standalone Java Virtual Machine or a J2EE container.

> **Note:** Using Oracle XML DB Content Connector in the database Oracle JVM (the Java Virtual Machine available within a database process) is not supported. To use the content connector in the database tier, you must use either a standalone Java Virtual Machine or a J2EE container.

Files and folders in Oracle XML DB Repository are represented as JCR nodes (and properties of those nodes). They can be created, retrieved, and updated using the JCR APIs.

How Oracle XML DB Repository Is Exposed in JCR

Oracle XML DB Content Connector represents data in Oracle XML DB Repository as JCR nodes and properties. Files and folders are represented as nodes of type `nt:file` and `nt:folder`, respectively. Their content and metadata is exposed as nodes of node type `nt:resource`.

This section contains the following topics:

- Example of How Files and Folders are Exposed in JCR
- Oracle Extensions to JCR Node Types
- Binary and XML Content
- System-Defined Metadata
- User-Defined Metadata
- Hard Links and Weak Links

Example of How Files and Folders are Exposed in JCR

The folder `MyFolder` is stored in the root folder of Oracle XML DB Repository. It contains two files, `Address.xml` and `Car.jpg`.

File `Address.xml` has the following XML content:

```
<Address country="US">
  <name>Alice Smith</name>
  <street>123 Maple Street</street>
  <city>Mill Valley</city>
  <state>CA</state>
  <zip>90952</zip>
</Address>
```

File `Car.jpg` has binary content: a picture of an automobile. It also has the following user-defined XML metadata:

```
<i:ImageMetadata>
  <Height>640</Height>
  <Width>480</Width>
  <RGB R="44" G="123" B="74"/>
</i:ImageMetadata>
```

Oracle XML DB Content Connector exposes `MyFolder`, `Address.xml`, and `Car.jpg` as JCR nodes and properties.

Example 31–1 shows folder `MyFolder` represented as a tree of JCR nodes and properties. In this representation, **`bold`** type indicates a node, *`italic`* type indicates a node type, `regular` type indicates a property, and italic type with angle brackets (<>) indicates omitted data, such as binary data.

Example 31–1 JCR Node Representation of MyFolder

```
[root] (nt:folder)
  jcr:created="2001-01-01T00:00:00.000Z"
  jcr:content (nt:resource)
    jcr:data=null
    jcr:lastModified="2001-01-01T00:00:00.000Z"
    ojcr:owner="SYS"
    ojcr:creator="SYS"
    ojcr:lastModifier="SYS"
    ojcr:displayName=""
    ojcr:language="en-US"
  MyFolder (nt:folder)
    jcr:created="2001-01-01T00:00:00.000Z"
    jcr:content (nt:resource)
      jcr:data=null
      jcr:lastModified="2001-01-01T00:00:00.000Z"
      ojcr:owner="ALICE"
      ojcr:creator="BOB"
      ojcr:lastModifier="CHARLIE"
      ojcr:author="BOB"
      ojcr:comment="An application folder"
      ojcr:displayName="MyFolder"
      ojcr:language="en-US"
      ojcr:links (ojcr:links)
        ojcr:folderLink (ojcr:linkProperties)
          ojcr:linkType="Hard"
          ojcr:linkSource=<RESID of the root folder>
          ojcr:linkTarget=<RESID of folder MyFolder>
          ojcr:linkName="MyFolder"
```

```
Address.xml (nt:file)
  jcr:created="2005-09-01T12:34:56.789Z"
  jcr:content (nt:resource)
    jcr:encoding="UTF-8"
    jcr:mimeType="text/xml"
    jcr:data=<binary representation of the XML content>
    jcr:lastModified="2005-09-01T12:34:56.789Z"
    ojcr:owner="ALICE"
    ojcr:creator="BOB"
    ojcr:lastModifier="CHARLIE"
    ojcr:author="BOB"
    ojcr:displayName="Address.xml"
    ojcr:language="en-US"
    ojcr:xmlContent (nt:unstructured)
      Address
        country="US"
        name
          jcr:xmltext
            jcr:xmlcharacters="Alice Smith"
        street
          jcr:xmltext
            jcr:xmlcharacters="123 Maple Street"
        city
          jcr:xmltext
            jcr:xmlcharacters="Mill Valley"
        state
          jcr:xmltext
            jcr:xmlcharacters="CA"
        zip
          jcr:xmltext
            jcr:xmlcharacters="90952"
    ojcr:links (ojcr:links)
      ojcr:folderLink (ojcr:linkProperties)
        ojcr:linkType="Hard"
        ojcr:linkSource=<RESID of folder MyFolder>
        ojcr:linkTarget=<RESID of file Address.xml>
        ojcr:linkName="Address.xml"
Car.jpg (nt:file)
  jcr:created="2004-02-12T16:15:23.247Z"
  jcr:content (nt:resource)
    jcr:mimeType="image/jpeg"
    jcr:data=<binary content of file Car.jpg>
    jcr:lastModified="2004-02-12T17:20:25.314Z"
    ojcr:owner="ALICE"
    ojcr:creator="BOB"
    ojcr:lastModifier="CHARLIE"
    ojcr:author="BOB"
    ojcr:displayName="A shiny red car!"
    ojcr:language="en-US"
    i:ImageMetadata
      Height
        jcr:xmltext
          jcr:xmlcharacters="640"
      Width
        jcr:xmltext
          jcr:xmlcharacters="480"
      RGB
        R="44"
        G="123"
        B="74"
```

```
ojcr:links (ojcr:links)
  ojcr:folderLink (ojcr:linkProperties)
    ojcr:linkType="Hard"
    ojcr:linkSource=<RESID of folder MyFolder>
    ojcr:linkTarget=<RESID of file Car.jpg>
    ojcr:linkName="Car.jpg"
```

Oracle Extensions to JCR Node Types

Oracle XML DB Content Connector augments the definitions of node types `nt:file`, `nt:folder`, and `nt:resource` to include additional information held in Oracle XML DB Repository. Node type `ojcr:folder` is added as a supertype of `nt:folder`, and node type `ojcr:resource` is added as a supertype of `nt:resource`. All Oracle extensions are in the namespace `http://xmlns.oracle.com/jcr/1.0`, which is mapped to namespace prefix **`ojcr`**.

In addition, node type `mix:referenceable` is added as a supertype of `nt:file` and `nt:folder` to allow all files and folders to be accessed by their resource id.

Binary and XML Content

Property `jcr:data` contains the binary content of a file. Note that `jcr:data` is a property not of node `nt:file`, but rather of its child node `jcr:content`.

For files containing XML content, node `jcr:content` has a child node `ojcr:xmlContent`, under which the XML content can be accessed as a set of JCR nodes and properties. File `Address.xml`, referenced in Example 31–1 on page 31-3, is such a file. The XML content of an XML file in the repository is mapped to JCR nodes and properties using the **document view serialization** defined by JCR, in which:

- XML elements are exposed as JCR nodes.
- XML attributes are exposed as JCR properties.
- XML text is exposed as JCR properties named `jcr:xmlcharacters` within nodes named `jcr:xmltext`.

System-Defined Metadata

Oracle XML DB Repository maintains metadata for each repository file and folder. In database views `RESOURCE_VIEW` and `PATH_VIEW`, this metadata is represented as a set of XML elements within `XMLType` column `RES`. In JCR, this metadata is mapped to properties in namespaces `jcr` and `ojcr`. Table 31–1 describes this mapping.

Table 31–1 Oracle XML DB Resource to JCR Mappings

XPath	Relative Path From Node nt:file or nt:folder
`/Resource/CreationDate`	`jcr:created`
`/Resource/ModificationDate`	`jcr:content/jcr:lastModified`
`/Resource/Author`	`jcr:content/ojcr:author`
`/Resource/DisplayName`	`jcr:content/ojcr:displayName`
`/Resource/Comment`	`jcr:content/ojcr:comment`
`/Resource/Language`	`jcr:content/ojcr:language`
`/Resource/CharacterSet`	`jcr:content/jcr:encoding`

Table 31–1 (Cont.) Oracle XML DB Resource to JCR Mappings

XPath	Relative Path From Node nt:file or nt:folder
`/Resource/ContentType`	`jcr:content/jcr:mimeType`
`/Resource/Owner`	`jcr:content/ojcr:owner`
`/Resource/Creator`	`jcr:content/ojcr:creator`
`/Resource/LastModifier`	`jcr:content/ojcr:lastModifier`

User-Defined Metadata

User-defined XML metadata is exposed as JCR nodes and properties under the `jcr:content` child node of the repository file or folder. As with XML file content, XML metadata is mapped to JCR nodes and properties using the document view serialization that is defined by JCR. See "Binary and XML Content" on page 31-5 for a description of this serialization.

In Example 31–1, file `Car.jpg` has this user-defined metadata:

```
<i:ImageMetadata>
  <Height>640</Height>
  <Width>480</Width>
  <RGB R="44" G="123" B="74"/>
</i:ImageMetadata>
```

The following JCR path retrieves the `Width` value:

```
/My Folder/Car.jpg/jcr:content/i:ImageMetadata/
   Width/jcr:xmltext/jcr:xmlcharacters
```

Hard Links and Weak Links

In JCR, each node and property has exactly one parent node, except for the root node, which has no parent. Consequently, there is exactly one absolute path to each JCR node or property.

However, in Oracle XML DB Repository, a resource (file or folder) can be linked to more than one parent folder, either by hard links, which control the life span of the child, or by weak links, which do not. Consequently, there can be more than one path to a resource, and a resource can have more than one parent.

In resolving a path, Oracle XML DB Content Connector traverses both hard and weak links. If there is more than one path to a resource, JCR method `getPath()` returns the path by which that resource was first discovered, subsequent to the most recent call to either `save()` or `refresh(boolean)` by that session. JCR method `getParent()` returns the folder targeted by that path.

It is often useful to obtain a list of all parents of a resource, if the resource is the target of more than one link and therefore has more than one parent folder. Oracle XML DB Content Connector presents this as nodes of type `ojcr:linkProperties` with path `jcr:content/ojcr:links/ojcr:folderLink` relative to node `nt:file` or `nt:folder`. There is one `ojcr:folderLink` node for each parent of the resource.

Node `ojcr:folderLink` has the following properties:

- `ojcr:linkType`: Link type (`Hard` or `Weak`)
- `ojcr:linkSource`: Resource id of the parent folder
- `ojcr:linkTarget`: Resource id of the child file or folder

- `ojcr:linkName`: Name of the child file or folder in that parent

How to Use Oracle XML DB Content Connector

This section describes how to use Oracle XML DB Content Connector to access information in Oracle XML DB Repository. It has the following topics:

- Setting CLASSPATH
- Obtaining the JCR Repository Object
- Sample Code to Upload File
- Additional Code Samples
- Logging API for Oracle XML DB Content Connector
- Supported JCR Compliance Levels
- Oracle XML DB Content Connector Restrictions

Setting CLASSPATH

Oracle XML DB Content Connector requires the following entries in the Java `CLASSPATH` variable:

- `$ORACLE_HOME/lib/jcr-1.0.jar`
- `$ORACLE_HOME/lib/ojcr.jar`
- `$ORACLE_HOME/lib/xmlparserv2.jar`
- `$ORACLE_HOME/jlib/xquery.jar`

Obtaining the JCR Repository Object

In Oracle XML DB Content Connector, `oracle.jcr.OracleRepository` implements the JCR interface `javax.jcr.Repository`, which provides the entry point to a JCR repository. The code fragment in Example 31–2 shows how to obtain a `Repository` object for Oracle XML DB Repository.

Example 31–2 Code Fragment Showing How to Get a Repository Object

```
import oracle.jcr.OracleRepository;
import oracle.jcr.OracleRepositoryFactory;
import oracle.jcr.xdb.XDBRepositoryConfiguration;
import oracle.jdbc.pool.OracleDataSource;
...
XDBRepositoryConfiguration configuration =
  new XDBRepositoryConfiguration();
OracleDataSource ods =
  (OracleDataSource)configuration.getDataSource();
// databaseURL is a JDBC database URL.
ods.setURL(databaseURL);
// OracleRepository implements javax.jcr.Repository.
OracleRepository repository =
  OracleRepositoryFactory.createOracleRepository(configuration);
```

`OracleRepository` implements both `java.io.Serializable` and `javax.naming.Referenceable`. This lets you create and configure an `OracleRepository` object upon application deployment, and store the ready-to-use `OracleRepository` object in a JNDI directory. At run-time, your application can

retrieve the preconfigured `OracleRepository` object from the JNDI directory. This approach, recommended by the JCR specification, separates deployment and run-time concerns.

In Oracle XML DB Content Connector, the set of prefix-to-URI mappings forming the JCR namespace registry is stored as part of the `OracleRepository` configuration.

> **See Also:** *Oracle Database XML Java API Reference*, package `oracle.jcr`

Sample Code to Upload File

Example 31–3 is a Java program that uploads a file from the local file system to Oracle XML DB Repository using Oracle XML DB Content Connector.

Example 31–3 Uploading a File using Oracle XML DB Content Connector

```
import java.io.FileInputStream;

import javax.jcr.Node;
import javax.jcr.Session;
import javax.jcr.SimpleCredentials;

import oracle.jcr.OracleRepository;
import oracle.jcr.OracleRepositoryFactory;

import oracle.jcr.xdb.XDBRepositoryConfiguration;

import oracle.jdbc.pool.OracleDataSource;

public class UploadFile
{
  public static void main(String[] args)
    throws Exception
  {
    String databaseURL = args[0];
    String userName = args[1];
    String password = args[2];
    String parentPath = args[3];
    String fileName = args[4];
    String mimeType = args[5];

    // Get the JCR Repository object.
    XDBRepositoryConfiguration configuration =
      new XDBRepositoryConfiguration();

    OracleDataSource ods =
      (OracleDataSource)configuration.getDataSource();

    ods.setURL(databaseURL);

    OracleRepository repository =
      OracleRepositoryFactory.createOracleRepository(configuration);

    // Create a JCR Session.
    SimpleCredentials sc =
      new SimpleCredentials(userName, password.toCharArray());

    Session session = repository.login(sc);
```

```
    // Get the parent node.
    Node parentNode = (Node)session.getItem(parentPath);

    // Get the child contents.
    FileInputStream inputStream = new FileInputStream(fileName);

    // Create child node.
    Node node = parentNode.addNode(fileName, "nt:file");
    Node contentNode = node.getNode("jcr:content");
    contentNode.setProperty("jcr:mimeType", mimeType);
    contentNode.setProperty("jcr:data", inputStream);

    // Save changes and logout.
    session.save();
    session.logout();
  }
}

// EOF
```

You can compile and run Example 31–3 from the command line. The program requires the following command-line arguments:

- JDBC database URL
- User ID
- User password
- Folder in Oracle XML DB Repository into which to upload the file
- File to be uploaded
- MIME type

Example 31–4 illustrates this.

Example 31–4 Uploading a File Using the Command Line

```
export CLASSPATH=.:$ORACLE_HOME/lib/jcr-1.0.jar:$ORACLE_HOME/lib/ojcr.jar:$ORACLE_
HOME/lib/xmlparserv2.jar:$ORACLE_HOME/jlib/xquery.jar

javac UploadFile.java

java UploadFile jdbc:oracle:oci:@ quine password /public MyFile.txt text/plain
```

Additional Code Samples

You can find additional sample code at the following location:

```
$ORACLE_HOME/xdk/demo/java/jcr
```

For each code sample, a `README` file describes its purpose and use.

Logging API for Oracle XML DB Content Connector

Oracle XML DB Content Connector uses the standard `java.util.logging` framework. You can use the logging API provided by that framework to control logging behavior. For example, the following Java code fragment disables all logging.

```
import java.util.logging.LogManager;
...
LogManager.getLogManager().reset();
```

Supported JCR Compliance Levels

The JSR-170 standard, which defines JCR version 1.0, defines two compliance levels and a set of optional features. Oracle XML DB Content Connector supports Level 1 (read functions) and Level 2 (write functions).

Oracle XML DB Content Connector Restrictions

This section describes certain restrictions of Oracle XML DB Content Connector.

Default Workspace Name

A single workspace is supported. In calling the `login(Credentials, String)` or `login(String)` methods of `javax.jcr.Repository`, the workspace name must be either an empty-string (`""`) or `NULL`.

Operations Restricted to Specific Node Types

Methods `save()` and `refresh()` of `javax.jcr.Item` can be called only on nodes whose type is `nt:file` or `nt:folder`. Method `move()` of `javax.jcr.Session` and methods `copy()` and `move()` of `javax.jcr.Workspace` can be called only on `nt:file` and `nt:folder` nodes.

Determining the State of Files or Folders

Methods `isNew()` and `isModified()` of `javax.jcr.Item` return the state of the file or folder containing the item, not the item itself. Method `isNew()` returns `true` if the file or folder has been created in the JCR transient layer but not saved. Method `isModified()` returns `true` if the file or folder has been changed in the transient layer but not saved.

Interaction Between Binary and XML Content

The `jcr:data` property contains the binary-format content of a file. If the file content is XML, there is also an `ojcr:xmlContent` node under which its XML content is exposed as JCR nodes and properties. Changes you make to the `ojcr:xmlContent` subtree are not reflected in the `jcr:data` property until those changes are saved. If you change both the `jcr:data` property and the `ojcr:xmlContent` subtree, then the `ojcr:xmlContent` subtree takes precedence when those changes are saved.

Order in Which Changes Are Saved

Method `save` of class `javax.jcr.Session` or class `javax.jcr.Item` saves changes made in the transient layer. If more than one node or property has been changed, then JCR does not specify the order in which the changes are stored. Oracle XML DB Content Connector saves changes in the following:

1. Apply updates to existing files and folders, in path-sorted order.
2. Create new files and folders, in path-sorted order.
3. Move existing files and folders, in reverse path-sorted order.
4. Delete existing files and folders, in reverse path-sorted order.

Undefined Properties

Properties that have definitions of type `UNDEFINED` are stored as `STRING` values.

Node Type nt:base Is Abstract

Node type `nt:base` is abstract and cannot be specified as the type of a new node.

Node jcr:content Is Created Automatically

When you create a node of type `nt:file` or `nt:folder`, a `jcr:content` node is created automatically as a child.

Saving Normalizes Node jcr:xmltext

Saving combines successive `jcr:xmltext` nodes, which represent text within XML content or user-defined metadata, into a single `jcr:xmltext` node.

Node Type mix:referenceable

Node types `nt:file`, `nt:folder`, and `nt:resource` are subtypes of mix-in node type `mix:referenceable`. Consequently, all `nt:file`, `nt:folder`, and `nt:resource` nodes can be referenced by UUID. You cannot add `mix:referenceable` to nodes of any type.

Full-Text Indexing

You can create a full-text index on file content using PL/SQL package `DBMS_XDBT`. This lets queries apply function `jcr:contains` to property `jcr:data` of a `jcr:content` node. Full-text indexes on other properties are not supported.

Using XML Schemas with JCR

Oracle XML DB Content Connector can create JCR node types from XML schemas.

This section has the following topics:

- Why Register XML Schemas for Use with JCR?
- How to Register an XML Schema with JCR
- How JCR Node Types are Generated from XML Schemas

Why Register XML Schemas for Use with JCR?

XML data can be stored in Oracle XML DB Repository as either file content or user-defined metadata. In either case, the XML data can be based on an XML schema. XML schema-based data is validated against an XML schema that has been registered with Oracle XML DB.

By default, the JCR nodes corresponding to XML document content and user-defined metadata are of node type `nt:unstructured`, a generic node type defined by JCR, even if the XML data is XML schema-based. Oracle XML DB Repository still validates any changes made through the Oracle XML DB Content Connector against the XML schema, but it is not possible to access or specify typing metadata through JCR.

However, Oracle XML DB Content Connector lets XML schemas be registered for use in JCR. This causes the content connector to generate JCR node types for the XML-schema simple types, complex types, and global element declarations in the registered XML schema.

In exposing XML data as JCR nodes, the content connector determines whether the XML data conforms to an XML schema registered for JCR use, based on the value of XML attribute `xsi:schemaLocation` or `xsi:noNamespaceSchemaLocation` of its root element. If the XML data conforms to a JCR registered XML schema, then the

XML data is exposed as JCR nodes of the node types generated from the XML schema, instead of using the generic node type `nt:unstructured`.

You can also use the generated JCR node types to create or update XML document content and user-defined metadata.

Example 31–5 shows an XML document with XML schema-based content.

Example 31–5 XML Document with XML Schema-Based Content

```
<Address country="US"
   xmlns:xsi=
     "http://www.w3.org/2001/XMLSchema-instance"
   xsi:noNamespaceSchemaLocation=
     "http://www.example.com/Address">
  <name>Alice Smith</name>
  <street>123 Maple Street</street>
  <city>Mill Valley</city>
  <state>CA</state>
  <zip>90952</zip>
</Address>
```

The content of Example 31–5 is valid with respect to the XML schema shown in Example 31–6.

Example 31–6 XML Schema

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:complexType name="USAddress">
    <xsd:sequence>
      <xsd:element name="name"   type="xsd:string"/>
      <xsd:element name="street" type="xsd:string"/>
      <xsd:element name="city"   type="xsd:string"/>
      <xsd:element name="state"  type="xsd:string"/>
      <xsd:element name="zip"    type="xsd:long"/>
    </xsd:sequence>
    <xsd:attribute name="country" type="xsd:NMTOKEN"
                   fixed="US"/>
  </xsd:complexType>
</xsd:schema>
```

Initially, this XML schema is not registered for JCR use. The JCR nodes and properties representing the XML content are shown in Example 31–7.

Example 31–7 JCR Representation of XML Content Not Registered for JCR Use

```
 ojcr:xmlContent (nt:unstructured)
   Address (nt:unstructured)
     country="US" (String)
     name (nt:unstructured)
       jcr:xmltext(ojcr:xmltext)
         jcr:xmlcharacters="Alice Smith" (String)
     street (nt:unstructured)
       jcr:xmltext(ojcr:xmltext)
         jcr:xmlcharacters="123 Maple Street" (String)
     city (nt:unstructured)
       jcr:xmltext(ojcr:xmltext)
         jcr:xmlcharacters="Mill Valley" (String)
     state (nt:unstructured)
       jcr:xmltext(ojcr:xmltext)
         jcr:xmlcharacters="CA" (String)
```

```
    zip (nt:unstructured)
      jcr:xmltext (ojcr:xmltext)
        jcr:xmlcharacters="90952" (String)
```

The XML schema is then registered for JCR use. The JCR nodes and properties are shown in Example 31–8.

Example 31–8 JCR Representation of XML Content Registered for JCR Use

```
ojcr:xmlContent (nt:unstructured)
  Address (USAddress)
    country="US" (String)
    name (xsd:string)
      jcr:xmltext(ojcr:xmltext)
        jcr:xmlcharacters="Alice Smith" (String)
    street (xsd:string)
      jcr:xmltext(ojcr:xmltext)
        jcr:xmlcharacters="123 Maple Street" (String)
    city (xsd:string)
      jcr:xmltext(ojcr:xmltext)
        jcr:xmlcharacters="Mill Valley" (String)
    state (xsd:string)
      jcr:xmltext(ojcr:xmltext)
        jcr:xmlcharacters="CA" (String)
    zip (xsd:long)
      jcr:xmltext (ojcr:xmltext)
        jcr:xmlcharacters="90952" (Long)
```

Node `Address` now has node type `USAddress`. Similarly, nodes `name`, `street`, `city`, and `state` have node type `xsd:string`. Node `zip` has node type `xsd:long`, and the `jcr:xmlcharacters` property of its `jcr:xmltext` child is a `LONG` property type.

See Also: Chapter 29, "User-Defined Repository Metadata"

How to Register an XML Schema with JCR

Before you register an XML schema for use with JCR, you must register it for use with Oracle XML DB, using PL/SQL procedure `DBMS_XMLSCHEMA.registerSchema`. For example, to register an XML schema with location `http://www.example.com/Address`, first register it for use with Oracle XML DB, as shown in Example 31–9. Then, register it for use with JCR, using Oracle XML DB Content Connector Java APIs, as shown in Example 31–10.

Example 31–9 Registering an XML Schema for Use with Oracle XML DB

```
BEGIN
  DBMS_XMLSCHEMA.registerSchema(
    SCHEMAURL       => 'http://www.example.com/Address',
    SCHEMADOC       => bfileContainingSchema,
    LOCAL           => FALSE,
    ENABLEHIERARCHY => DBMS_XMLSCHEMA.ENABLE_HIERARCHY_RESMETADATA);
END;
/
```

Note: You can use only globally registered XML schemas (`local => false`) with JCR.

Example 31–10 Registering an XML Schema for Use with JCR

```
import oracle.jcr.nodetype.OracleNodeTypeManager;
...
OracleNodeTypeManager ntm = (OracleNodeTypeManager)
  session.getWorkspace().getNodeTypeManager();

ntm.registerXMLSchema("http://www.example.com/Address", null);
```

The list of XML schemas registered for use with JCR is stored in the `OracleRepository` object. You can save this registration data by storing the `OracleRepository` object in a JNDI directory, as recommended by the JCR specification.

JCR requires that each node type have a unique name. By default, Oracle XML DB Content Connector generates JCR nodes types that correspond to a registered XML schema in the target namespace of the XML schema. If you wish to register two XML schemas with the same namespace, and the XML schemas declare types with the same names, you can avoid a name clash by overriding the namespace into which the JCR node types are generated. Refer to the Javadoc of method `registerXMLSchema()` for details.

See Also:

- "Managing XML Schemas with DBMS_XMLSCHEMA" on page 7-6 for information on registering XML schemas with Oracle XML DB
- *Oracle Database XML Java API Reference*, package `oracle.jcr`, for information on Java method `registerXMLSchema()`

How JCR Node Types are Generated from XML Schemas

This section describes how Oracle XML DB Content Connector generates JCR node types from XML schemas that are registered for JCR use.

The type models of JCR and XML Schema are similar but not equivalent. Some aspects of XML Schema have no representation in JCR. For example, some constraining facets of an XML-schema simple type are not discoverable through JCR. They are enforced by Oracle XML DB Content Connector nonetheless.

More generally, the JCR node types generated from an XML schema do not augment, detract, or alter the XML schema validation performed when XML data that conforms to that XML schema is created or updated, whether through JCR or other interfaces.

Built-In Simple Types

A JCR node type is provided for each XML Schema built-in type. For example, the JCR node type `xsd:decimal` corresponds to the built-in type `xsd:decimal`.

The inheritance hierarchy of the JCR node types follows that of the built-in types. For example, `xsd:integer` is a subtype of `xsd:decimal`.

Each XML Schema built-in type maps to a JCR property value type, which is used to represent values of that type in JCR.

Table 31–2 XML Schema Built-In Types Mapped to JCR Property Value Types

XML Schema Built-in Type	JCR Property Value Type
`xsd:anySimpleType`	`STRING`

Table 31–2 (Cont.) XML Schema Built-In Types Mapped to JCR Property Value Types

XML Schema Built-in Type	JCR Property Value Type
xsd:anyURI	STRING
xsd:base64Binary	BINARY
xsd:boolean	BOOLEAN
xsd:byte	LONG
xsd:date	DATE (1)
xsd:dateTime	DATE (1)
xsd:decimal	DOUBLE (2)
xsd:double	DOUBLE
xsd:duration	STRING
xsd:ENTITIES	STRING (3)
xsd:ENTITY	STRING
xsd:float	DOUBLE
xsd:gDay	STRING
xsd:gMonth	STRING
xsd:gMonthDay	STRING
xsd:gYear	STRING
xsd:gYearMonth	STRING
xsd:hexBinary	BINARY
xsd:ID	STRING
xsd:IDREF	STRING
xsd:IDREFS	STRING (3)
xsd:int	LONG
xsd:integer	LONG (2)
xsd:language	STRING
xsd:long	LONG
xsd:Name	STRING
xsd:NCName	STRING
xsd:negativeInteger	LONG
xsd:NMTOKEN	STRING
xsd:NMTOKENS	STRING (3)
xsd:nonNegativeInteger	LONG
xsd:nonPositiveInteger	LONG
xsd:normalizedString	STRING
xsd:NOTATION	STRING
xsd:positiveInteger	LONG
xsd:QName	STRING
xsd:short	LONG

Table 31–2 (Cont.) XML Schema Built-In Types Mapped to JCR Property Value Types

XML Schema Built-in Type	JCR Property Value Type
xsd:string	STRING
xsd:time	DATE (1)
xsd:token	STRING
xsd:unsignedByte	LONG
xsd:unsignedInt	LONG
xsd:unsignedLong	LONG
xsd:unsignedShort	LONG

Notes for Table 31–2:

1. The JCR DATE property type is accessed using java.util.Calendar objects. Since Calendar requires all fields to be set, a mask of 1970-01-01T00:00:00.000+00:00 is used to supply default values for missing fields when Property.getDate() or Value.getDate() is called. This includes omitted hour/minute/second values (for xsd:date), year/month/day values (for xsd:time), or time-zone values (for xsd:date, xsd:time, and xsd:dateTime). Calling Property.getString() or Value.getString() returns the unparsed string representation. Similarly, Property.setValue(String) or Property.setValue(valueFactory.createValue(String)) may be used to set DATE properties without applying the mask.
2. The value space of xsd:decimal and xsd:integer exceeds that of the corresponding JCR types, DOUBLE and LONG (accessed as Java double and long values). Consequently, some xsd:decimal and xsd:integer values can only be accessed in JCR as strings. For example, bigIntegerProperty.getLong() throws a javax.jcr.ValueFormatException, but bigIntegerProperty.getString() returns the unparsed string representation. Similarly, Property.setValue(String) or Property.setValue(valueFactory.createValue(String)) may be used to set DOUBLE or LONG properties to values outside the JCR value space.
3. xsd:ENTITIES, xsd:IDREFS, and xsd:NMTOKENS are represented in JCR as multi-valued STRING properties.

XML Schema-Defined Simple Types

A JCR node type is created for each simple type defined in an XML schema. The inheritance hierarchy of the JCR node types follows that of the XML schema types.

A derived-by-list simple type is represented by a multi-valued JCR property definition.

A derived-by-union simple type is represented by a JCR property definition with property type UNDEFINED.

The JCR node type corresponding to an anonymous simple type has a synthetic name anonymousNodeType#*sequenceNumber*. Your application should not rely on the synthesized name. It is not guaranteed to be the same across sessions, and it may change when an XML schema is registered or deregistered for JCR use or the definition of a registered XML schema is changed.

Complex Types

A JCR node type is created for each complex type defined in an XML schema. The inheritance hierarchy of the JCR node types follows that of the XML schema types.

For a JCR node type corresponding to an XML schema complex type:

- A property definition is created for each attribute declaration of the complex type. Attribute declarations or attribute groups referenced by name in a complex type are treated as though they were defined in line.
- A residual property definition is created if the complex type has an attribute wildcard.
- A child node definition is created for each uniquely-named element declaration in the complex type's content model. Element declarations or module groups referenced by name are treated as though they were defined in line. If an element declaration is the head of a substitution group, a child node definition is also created for each element declaration within the substitution group.
- A residual child node definition is created if the complex type has an element wildcard.
- A `jcr:xmltext` child node definition is created if the complex type permits XML text, either because `xsd:mixed = "true"` or it is an `xsd:simpleContent` definition.

The JCR node type for a complex type supports child node ordering.

It is not possible to determine whether a type was derived by extension or restriction using JCR.

The JCR node type corresponding to an anonymous complex type has a synthetic name `anonymousNodeType#`*`sequenceNumber`*. Your application should not rely on the synthesized name. It is not guaranteed to be the same across sessions, and it may change when an XML schema is registered or deregistered for JCR use or the definition of a registered XML schema is changed.

Global Element Declarations

A JCR node type is created for each global element declaration in an XML schema. The local name of the generated node type is formed by prepending an underscore (_) to the local name of the global element declaration. For example, in a namespace-qualified purchase order XML schema, a node type named `po:_purchaseOrder` is created for global element named `po:purchaseOrder`.

32

Writing Oracle XML DB Applications in Java

This chapter describes how to write Oracle XML DB applications in Java. It includes design guidelines for writing Java applications including servlets, and how to configure the Oracle XML DB servlets.

This chapter contains these topics:

- Overview of Oracle XML DB Java Applications
- Design Guidelines: Java Inside or Outside the Database?
- Writing Oracle XML DB HTTP Servlets in Java
- Configuring Oracle XML DB Servlets
- HTTP Request Processing for Oracle XML DB Servlets
- Session Pool and Oracle XML DB Servlets
- Native XML Stream Support
- Oracle XML DB Servlet APIs
- Oracle XML DB Servlet Example

Overview of Oracle XML DB Java Applications

You can use Java code in these ways:

- In the database using the Java Virtual Machine (VM).
- In a client or application server, using the OCI driver for JDBC.

Because Java runs in the database in the context of the database server process, the ways you can deploy and run Java code are restricted to the following:

- You can run Java code as a stored procedure invoked from SQL or PL/SQL.
- You can run a Java servlet.

Stored procedures are easier to integrate with SQL and PL/SQL code. They require Oracle Net Services as the protocol to access Oracle Database.

Servlets work better as the top-level entry point into Oracle Database, and require using HTTP(S) as the protocol to access Oracle Database.

Which Oracle XML DB APIs Are Available Inside and Outside the Database?

All Oracle XML DB application program interfaces (APIs) are available to applications running both in the server and outside the database, including:

- JDBC support for `XMLType`
- `XMLType` class
- Java DOM implementation

Design Guidelines: Java Inside or Outside the Database?

When choosing an architecture for writing Java Oracle XML DB applications, consider the following guidelines:

HTTP(S): Accessing Java Servlets or Directly Accessing XMLType Resources

If the downstream client wants to deal with XML in its textual representation, then using HTTP(S) to either access the Java servlets or directly access `XMLType` resources performs the best, especially if the XML node tree is not being manipulated much by the Java program.

The Java implementation in the server can natively move data from the database to the network without converting character data through UCS-2 Unicode (which is required by Java strings). In many cases data is copied directly from the database buffer cache to the HTTP(S) connection. There is no need to convert data from the buffer cache into the SQL serialization format used by Oracle Net Services, then move it to the JDBC client, and then convert to XML. Loading on demand and the LRU cache for `XMLType` are most effective inside the database server.

Accessing Many XMLType Object Elements: Use JDBC XMLType Support

If the downstream client is an application that programmatically accesses many or most of the elements of an `XMLType` instance using Java, then use JDBC `XMLType` support for best performance. It is often easier to debug Java programs outside of the database server, as well.

Use the Servlets to Manipulate and Write Out Data Quickly as XML

Oracle XML DB servlets are intended for writing HTTP stored procedures in Java that can be accessed using HTTP(S). They are not intended as a platform for developing an entire Internet application. In that case, the application servlet should be deployed in Oracle Application Server application server and access data in the database either using JDBC, or by using the `java.net.*` or similar APIs to get XML data through HTTP(S).

They are best used for applications that want to get into the database, manipulate the data, and write it out quickly as XML, not to format HTML pages for end-users.

Writing Oracle XML DB HTTP Servlets in Java

Oracle XML DB provides a protocol server that supports FTP, HTTP 1.1, WebDAV, and Java Servlets. Oracle XML DB supports Java Servlet version 2.2, with the following exceptions:

- The servlet WAR file (`web.xml`) is not supported in its entirety. Some `web.xml` configuration parameters must be handled manually. For example, creating roles must be done using the `SQL CREATE ROLE` command.
- `RequestDispatcher` and associated methods are *not* supported.
- Method `HTTPServletRequest.getCookies()` is *not* supported.

- Only one `ServletContext` (and one `web-app`) is currently supported.
- Stateful servlets (and thus the `HttpSession` class methods) are *not* supported. Servlets must maintain state in the database itself.

Configuring Oracle XML DB Servlets

Oracle XML DB servlets are configured using the `/xdbconfig.xml` file in Oracle XML DB Repository. Many of the XML elements in this file are the same as those defined by the Java Servlet 2.2 specification portion of Java 2 Enterprise Edition (J2EE), and have the same semantics. Table 32–1 lists the XML elements defined for the servlet deployment descriptor by the Java Servlet specification, along with extension elements supported by Oracle XML DB.

Table 32–1 XML Elements Defined for Servlet Deployment Descriptors

XML Element Name	Defined By	Supported?	Description	Comment
`auth-method`	Java	no	Specifies an HTTP authentication method required for access	--
`charset`	Oracle	yes	Specifies an IANA character set name	For example: ISO8859, UTF-8
`charset-mapping`	Oracle	yes	Specifies a mapping between a filename extension and a charset	--
`context-param`	Java	no	Specifies a parameter for a Web application	Not yet supported
`description`	Java	yes	A string for describing a servlet or Web application	Supported for servlets
`display-name`	Java	yes	A string to display with a servlet or Web application	Supported for servlets
`distributable`	Java	no	Indicates whether or not this servlet can function if all instances are not running in the same Java virtual machine	All servlets running in Oracle Database MUST be distributable.
`errnum`	Oracle	yes	Oracle error number	See *Oracle Database Error Messages*
`error-code`	Java	yes	HTTP(S) error code	Defined by RFC 2616
`error-page`	Java	yes	Defines a URL to redirect to if an error is encountered.	Can be specified through an HTTP(S) error, an uncaught Java exception, or through an uncaught Oracle error message
`exception-type`	Java	yes	Classname of a Java exception mapped to an error page	--
`extension`	Java	yes	A filename extension used to associate with MIME types, character sets, and so on.	--
`facility`	Oracle	yes	Oracle facility code for mapping error pages	For example: ORA, PLS, and so on.
`form-error-page`	Java	no	Error page for form login attempts	Not yet supported
`form-login-config`	Java	no	Config spec for form-based login	Not yet supported
`form-login-page`	Java	no	URL for the form-based login page	Not yet supported

Table 32–1 (Cont.) XML Elements Defined for Servlet Deployment Descriptors

XML Element Name	Defined By	Supported?	Description	Comment
`icon`	Java	Yes	URL of icon to associate with a servlet	Supported for servlets
`init-param`	Java	Yes	Initialization parameter for a servlet	--
`jsp-file`	Java	No	Java Server Page file to use for a servlet	Not supported
`lang`	Oracle	Yes	IANA language name	For example: en-US
`lang-mapping`	Oracle	Yes	Specifies a mapping between a filename extension and language content	--
`large-icon`	Java	Yes	Large sized icon for icon display	--
`load-on-startup`	Java	Yes	Specifies if a servlet is to be loaded on startup	--
`location`	Java	Yes	Specifies the URL for an error page	Can be a local path name or HTTP(S) URL
`login-config`	Java	No	Specifies a method for authentication	Not yet supported
`mime-mapping`	Java	Yes	Specifies a mapping between filename extension and the MIME type of the content	--
`mime-type`	Java	Yes	MIME type name for resource content	For example: text/xml or application/octet-stre am
`OracleError`	Oracle	Yes	Specifies an Oracle error to associate with an error page	--
`param-name`	Java	Yes	Name of a parameter for a Servlet or `ServletContext`	Supported for servlets
`param-value`	Java	Yes	Value of a parameter	--
`realm-name`	Java	No	HTTP(S) realm used for authentication	Not yet supported
`role-link`	Java	Yes	Specifies a role a particular user must have for accessing a servlet	Refers to a database role name. Make sure to capitalize by default!
`role-name`	Java	Yes	A servlet name for a role	Just another name to call the database role. Used by the Servlet APIs
`security-role`	Java	No	Defines a role for a servlet to use	Not supported. You must manually create roles using the SQL `CREATE ROLE`
`security-role-ref`	Java	Yes	A reference between a servlet and a role	--
`servlet`	Java	Yes	Configuration information for a servlet	--

Table 32–1 (Cont.) XML Elements Defined for Servlet Deployment Descriptors

XML Element Name	Defined By	Supported?	Description	Comment
`servlet-class`	Java	Yes	Specifies the classname for the Java servlet	--
`servlet-language`	Oracle	Yes	Specifies the programming language in which the servlet is written.	Either Java, C, or PL/SQL. Currently, only Java is supported for customer-defined servlets.
`servlet-mapping`	Java	Yes	Specifies a filename pattern with which to associate the servlet	All of the mappings defined by Java are supported
`servlet-name`	Java	Yes	String name for a servlet	Used by servlet APIs
`servlet-schema`	Oracle	Yes	The Oracle Schema in which the Java class is loaded. If not specified, then the schema is searched using the default resolver specification.	If this is not specified, then the servlet must be loaded into the `SYS` schema to ensure that everyone can access it, or the default Java class resolver must be altered. Note that the servlet schema is capitalized unless the value is enclosed in double-quotes.
`session-config`	Java	No	Configuration information for an `HTTPSession`	`HTTPSession` is not supported
`session-timeout`	Java	No	Timeout for an HTTP(S) session	`HTTPSession` is not supported
`small-icon`	Java	Yes	Small icon to associate with a servlet	--
`taglib`	Java	No	JSP tag library	JSPs currently not supported
`taglib-uri`	Java	No	URI for JSP tag library description file relative to file `web.xml`	JSPs currently not supported
`taglib-location`	Java	No	Path name relative to the root of the Web application where the tag library is stored	JSPs currently not supported
`url-pattern`	Java	Yes	URL pattern to associate with a servlet	See Section 10 of Java Servlet 2.2 spec
`web-app`	Java	No	Configuration for a Web application	Only one Web application is currently supported
`welcome-file`	Java	Yes	Specifies a welcome-file name	--
`welcome-file-list`	Java	Yes	Defines a list of files to display when a folder is referenced through an HTTP `GET` request	Example: `index.html`

Note:

- The following parameters defined for the `web.xml` file by Java are usable only by J2EE-compliant Enterprise Java Bean containers, and are not required for Java Servlet containers that do not support a full J2EE environment: `env-entry`, `env-entry-name`, `env-entry-value`, `env-entry-type`, `ejb-ref`, `ejb-ref-type`, `home`, `remote`, `ejb-link`, `resource-ref`, `res-ref-name`, res-type, `res-auth`.
- The following elements are used to define access control for resources: `security-constraint`, `web-resource-collection`, `web-resource-name`, `http-method`, `user-data-constraint`, `transport-guarantee`, `auth-constrain`. Oracle XML DB provides this functionality through access control lists (ACLs). An ACL is a list of access control entries (ACEs) that determines which principals have access to a given resource or resources. A future release will support using a `web.xml` file to generate ACLs.

See Also: Chapter 34, "Administering Oracle XML DB" for more information about configuring the `/xdbconfig.xml` file

HTTP Request Processing for Oracle XML DB Servlets

Oracle XML DB handles an HTTP request using the following steps:

1. If a connection has not yet been established, then Oracle Listener hands the connection to a shared server dispatcher.
2. When a new HTTP request arrives, the dispatcher wakes up a shared server.
3. The HTTP headers are parsed into appropriate structures.
4. The shared server attempts to allocate a database session from the Oracle XML DB session pool, if available, but otherwise creates a new session.
5. A new database call and a new database transaction are started.
6. If HTTP(S) has included authentication headers, then the session is authenticated as that database user (just as if the user logged into SQL*Plus). If no authentication information is included, and the request is `GET` or `HEAD`, then Oracle XML DB attempts to authenticate the session as the `ANONYMOUS` user. If that database user account is locked, then no unauthenticated access is allowed.
7. The URL in the HTTP request is matched against the servlets in the `xdbconfig.xml` file, as specified by the Java Servlet 2.2 specification.
8. The Oracle XML DB Servlet container is invoked in the Java VM inside Oracle. If the specified servlet has not been initialized yet, then the servlet is initialized.
9. The Servlet reads input from the `ServletInputStream`, and writes output to the `ServletOutputStream`, and returns from method `service()`.
10. If no uncaught Oracle error occurred, then the session is put back into the session pool.

See Also: Chapter 28, "Accessing the Repository using Protocols"

Session Pool and Oracle XML DB Servlets

Oracle Database uses one Java VM for each database session. A session that is reused from the session pool retains any state that is left over in the Java VM (Java static variables) from the last time that session was used.

This can be useful in caching Java state that is not user-specific, such as metadata, but Do not store secure user data in Java static memory. This could turn into a security hole inadvertently introduced by your application if you are not careful.

Native XML Stream Support

Node class `DOM` has an Oracle-specific method called `write()`, which takes the following arguments, returning void:

- `java.io.OutputStream` stream: A Java stream for writing the XML text.
- String `charEncoding`: The character encoding for writing the XML text. If `NULL`, then the database character set is used.
- `Short indent` The number of characters to indent nested XML elements.

Java method `write()` has a shortcut implementation if the stream provided is the `ServletOutputStream` provided inside the database. The contents of the Node are written in XML in native code directly to the output socket. This bypasses any conversions into and out of Java objects or Unicode (required for Java strings) and provides very high performance.

Oracle XML DB Servlet APIs

The APIs supported by Oracle XML DB servlets are defined by the Java Servlet 2.2 specification, the `Javadoc` for which is available, as of the time of writing this, online at: `http://java.sun.com/products/servlet/2.2/javadoc/index.html`

Table 32–2 lists Java Servlet 2.2 methods that are not implemented. They result in run-time exceptions.

Table 32–2 Java Servlet 2.2 Methods that Are Not Implemented

Interface	Methods Not Implemented
`HttpServletRequest`	`getSession()`, `isRequestedSessionIdValid()`
`HttpSession`	all
`HttpSessionBindingListener`	all

Oracle XML DB Servlet Example

Example 32–1 shows a simple servlet that prints the content of file resource `/public/test/foo1.text`.

Example 32–1 An Oracle XML DB Servlet

```
import javax.servlet.http.*;
import javax.servlet.*;
import java.util.*;
import java.io.*;
import java.util.*;
import java.io.IOException;
```

```
import java.io.OutputStreamWriter;
import java.io.Reader;
import java.io.Writer;
import java.sql.DriverManager;
import java.sql.SQLException;
import oracle.jdbc.OracleConnection;
import oracle.jdbc.OracleDriver;
import oracle.jdbc.OraclePreparedStatement;
import oracle.jdbc.OracleResultSet;
import oracle.sql.CLOB;
import oracle.xdb.XMLType;
import oracle.xdb.spi.XDBResource;

public class test extends HttpServlet {
  public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException,
           IOException {
      try {
        try {
          // Get the database connection for the current HTTP session
          DriverManager.registerDriver(new oracle.jdbc.OracleDriver());
          OracleDriver ora = new OracleDriver();
          OracleConnection databaseConnection =
            (OracleConnection) ora.defaultConnection();
          String statementText =
            "SELECT XDBURIType('/public/test/foo1.txt').getClob() FROM DUAL";
          OraclePreparedStatement statement =
            (OraclePreparedStatement)
            databaseConnection.prepareStatement(statementText);
          OracleResultSet resultSet = null;
          CLOB content = null;
          // Execute the statement
          resultSet = (OracleResultSet) statement.executeQuery();
          while (resultSet.next())
            {// The statement returns a CLOB.
             // Copy content of CLOB to server's output stream.
             content = resultSet.getCLOB(1);
             Reader reader = content.getCharacterStream();
             Writer writer =
               new OutputStreamWriter(response.getOutputStream());
             int bytesSent = 0;
             int n;
             char[] buffer = new char[CLOB.MAX_CHUNK_SIZE];
             while (-1 != (n = reader.read(buffer)))
               { bytesSent = bytesSent + n;
                 writer.write(buffer, 0, n); }
             writer.flush();
             if (content.isOpen()) { content.close(); }}
          resultSet.close();
          statement.close();
          databaseConnection.close();
          response.getOutputStream().write('\n'); }
        catch (SQLException sql)
          { throw new ServletException(sql); }}
      catch (ServletException se )
        { se.printStackTrace(); }
      finally
        { System.out.flush(); }}}
```

To install the servlet, you compile it, then load it into Oracle Database:

```
% loadjava -grant public -u quine/curry -r test.class
```

Finally, register and map the servlet, associating it with a URL, as shown in Example 32–2.

Example 32–2 Registering and Mapping an Oracle XML DB Servlet

```
EXEC DBMS_XDB.addServlet('TestServletFoo', 'Java', 'TestServletFoo',
                         NULL, NULL, 'test', NULL, NULL, 'XDB');

EXEC DBMS_XDB.addServletMapping('/public/test/foo1.txt', 'TestServletFoo');

COMMIT;
```

33

Using Native Oracle XML DB Web Services

This chapter contains these topics:

- Overview of Native Oracle XML DB Web Services
- Configuring and Enabling Web Services for Oracle XML DB
- Querying Oracle XML DB using a Web Service
- Accessing PL/SQL Stored Procedures using a Web Service

Overview of Native Oracle XML DB Web Services

Web services provide a standard way for applications to exchange information over the Internet and access services that implement business logic. Your applications can access Oracle Database using native Oracle XML DB Web services. One available service lets you issue SQL and XQuery queries and receive results as XML data. Another service provides access to all PL/SQL stored functions and procedures.

You can customize the input and output document formats when you use the latter service. If you do that then the WSDL is automatically generated by the native database Web services engine.

SOAP 1.1 is the version supported by Oracle XML DB. Applications use HTTP method `POST` to submit SOAP requests to native Oracle XML DB Web services. You can configure the locations of all native Oracle XML DB Web services and WSDL documents using the Oracle XML DB configuration file, `xdbconfig.xml`. You can also configure security settings for the Web services using the same configuration file.

You can use the `Accept-Charsets` field of the input HTTP header to specify the character set of Web-service responses. If this header field is omitted, then responses are in the database character set. The language of the input document and any error responses is the locale language of the database.

Error handling for native Oracle XML DB Web services uses the SOAP framework for faults.

See Also:

- http://www.w3.org/2002/ws/ for more information on Web services
- The W3C SOAP primer, http://www.w3.org/TR/2000/NOTE-SOAP-20000508, for more information on SOAP
- http://www.w3.org/TR/wsdl for information on the Web Services Description Language (WSDL)
- http://www.w3.org/TR/2003/REC-soap12-part0-20030624/#L11549 for information on SOAP fault handling
- "Configuring Oracle XML DB using xdbconfig.xml" on page 34-5

Configuring and Enabling Web Services for Oracle XML DB

For security reasons, Oracle XML DB is not preconfigured with the native Web services enabled. To make native Oracle XML DB Web services available, you must have the Oracle XML DB HTTP server up and running, and you must explicitly add Web service configuration. Then, to allow specific users to use Web services, you must grant them appropriate roles.

1. Configure Web services – see "Configuring Web Services for Oracle XML DB".
2. Enable Web services for specific users, by granting them appropriate roles – "Enabling Web Services for Specific Users".

See Also: "Using HTTP(S) and Oracle XML DB Protocol Server" on page 28-14 for information about Oracle XML DB HTTP server

Configuring Web Services for Oracle XML DB

To make Web services available for Oracle XML DB, log on as user SYS and add the servlet configuration that is shown as the query output of Example 33–2 to your Oracle XML DB configuration file, xdbconfig.xml.

Example 33–1 shows how to use procedures in PL/SQL package DBMS_XDB to add the servlet. Example 33–2 shows how to verify that the servlet was added correctly.

Example 33–1 Adding a Web Services Configuration Servlet

```
DECLARE
  SERVLET_NAME VARCHAR2(32) := 'orawsv';
BEGIN
  DBMS_XDB.deleteServletMapping(SERVLET_NAME);
  DBMS_XDB.deleteServlet(SERVLET_NAME);
  DBMS_XDB.addServlet(NAME     => SERVLET_NAME,
                      LANGUAGE => 'C',
                      DISPNAME => 'Oracle Query Web Service',
                      DESCRIPT => 'Servlet for issuing queries as a Web Service',
                      SCHEMA   => 'XDB');
  DBMS_XDB.addServletSecRole(SERVNAME => SERVLET_NAME,
                             ROLENAME => 'XDB_WEBSERVICES',
                             ROLELINK => 'XDB_WEBSERVICES');
  DBMS_XDB.addServletMapping(PATTERN => '/orawsv/*',
                             NAME    => SERVLET_NAME);
END;
/
```

Example 33–2 Verifying Addition of Web Services Configuration Servlet

```
XQUERY declare default element namespace "http://xmlns.oracle.com/xdb/xdbconfig.xsd"; (: :)
       (: This path is split over two lines for documentation purposes only.
          The path should actually be a single long line. :)
       for $doc in fn:doc("/xdbconfig.xml")/xdbconfig/sysconfig/protocolconfig/httpconfig/
        webappconfig/servletconfig/servlet-list/servlet[servlet-name='orawsv']
       return $doc
/

Result Sequence
-----------------------------------------------------------------------
<servlet xmlns="http://xmlns.oracle.com/xdb/xdbconfig.xsd">
  <servlet-name>orawsv</servlet-name>
  <servlet-language>C</servlet-language>
  <display-name>Oracle Query Web Service</display-name>
  <description>Servlet for issuing queries as a Web Service</description>
  <servlet-schema>XDB</servlet-schema>
  <security-role-ref>
    <description/>
    <role-name>XDB_WEBSERVICES</role-name>
    <role-link>XDB_WEBSERVICES</role-link>
  </security-role-ref>
</servlet>

1 item(s) selected.
```

See Also: "Configuring Oracle XML DB using xdbconfig.xml" on page 34-5 for more information about configuring Oracle XML DB with `xdbconfig.xml`

Enabling Web Services for Specific Users

To enable Web services for a specific user, log on as user `SYS` and grant the role `XDB_WEBSERVICES` to the user. This role enables Web services over HTTPS. This role is *required* to be able to use Web services.

User `SYS` can, in *addition*, grant one or both of the following roles to the user:

- `XDB_WEBSERVICES_OVER_HTTP` – Enable use of Web services over HTTP (not just HTTPS).
- `XDB_WEBSERVICES_WITH_PUBLIC` – Enable access, using Web services, to database objects that are accessible to `PUBLIC`.

If a user is not granted `XDB_WEBSERVICES_WITH_PUBLIC`, then the user has access, using Web services, to all database objects (regardless of owner) that would normally be available to the user, *except* for `PUBLIC` objects. To make `PUBLIC` objects accessible to a user through Web services, `SYS` must grant role `XDB_WEBSERVICES_WITH_PUBLIC` to the user. With this role, a user can access any `PUBLIC` objects that would normally be available to the user if logged on to the database.

Querying Oracle XML DB using a Web Service

The Oracle XML DB Web service for database queries is located at URL `http://`*`host`*`:`*`port`*`/`**`orawsv`**, where *`host`* and *`port`* are the host and HTTP(S) port properties of your database. This Web service has a WSDL document associated with it that specifies the formats of the incoming and outgoing documents using XML Schema. This WSDL document is located at URL `http://`*`host`*`:`*`port`*`/`**`orawsv?wsdl`**.

Your application sends database queries to the Web service as XML documents that conform to the XML schema listed in Example 33–3.

Example 33–3 XML Schema for Database Queries To Be Processed by Web Service

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:xdb="http://xmlns.oracle.com/xdb"
        targetNamespace="http://xmlns.oracle.com/orawsv">
  <element name="query">
    <complexType>
      <sequence>
        <element name="query_text">
          <complexType>
            <simpleContent>
              <extension base="string">
                <attribute name="type">
                  <simpleType>
                    <restriction base="NMTOKEN">
                      <enumeration value="SQL"/>
                      <enumeration value="XQUERY"/>
                    </restriction>
                  </simpleType>
                </attribute>
              </extension>
            </simpleContent>
          </complexType>
        </element>
        <choice maxOccurs="unbounded">
          <element name="bind">
            <complexType>
              <simpleContent>
                <extension base="string">
                  <attribute name="name" type="string"/>
                </extension>
              </simpleContent>
            </complexType>
          </element>
          <element name="bindXML" type="any"/>
        </choice>
        <element name="null_handling" minOccurs="0">
          <simpleType>
            <restriction base="NMTOKEN">
              <enumeration value="DROP_NULLS"/>
              <enumeration value="NULL_ATTR"/>
              <enumeration value="EMPTY_TAG"/>
            </restriction>
          </simpleType>
        </element>
        <element name="max_rows" type="positiveInteger" minOccurs="0"/>
        <element name="skip_rows" type="positiveInteger" minOccurs="0"/>
        <element name="pretty_print" type="boolean" minOccurs="0"/>
        <element name="indentation_width" type="positiveInteger" minOccurs="0"/>
        <element name="rowset_tag" type="string" minOccurs="0"/>
        <element name="row_tag" type="string" minOccurs="0"/>
        <element name="item_tags_for_coll" type="boolean" minOccurs="0"/>
      </sequence>
    </complexType>
  </element>
</schema>
```

This XML schema is contained in the WSDL document. The important parts of incoming query documents are as follows:

- `query_text` – The text of your query. Attribute `type` specifies the type of your query: either `SQL` or `XQUERY`.
- `bind` – A scalar bind-variable value. Attribute `name` names the variable.
- `bindXML` – An `XMLType` bind-variable value.
- `null_handling` – How `NULL` values returned by the query are to be treated:
  - `DROP_NULLS` – Put nothing in the output (no element). This is the default behavior.
  - `NULL_ATTR` – Use an empty element for `NULL`-value output. Use attribute `xsi:nil = "true"` in the element.
  - `EMPTY_TAG` – Use an empty element for `NULL`-value output, without a `nil` attribute.
- `max_rows` – The maximum number of rows to output for the query. By default, all rows are returned.
- `skip_rows` – The number of query output rows to skip, before including rows in the data returned in the SOAP message. You can use this in connection with `max_rows` to provide paginated output. The default value is zero (`0`).
- `pretty_print` – Whether the output document should be formatted for pretty-printing. The default value is `true`, meaning that the document is pretty-printed. When the value is `false`, no pretty-printing is done, and output rows are not broken with newline characters.
- `indentation_width` – The number of characters to indent nested elements that start a new line. The default value is one (`1`).
- `rowset_tag` – Name of the root element of the output document.
- `row_tag` – Name of the element whose value is a single row of query output.
- `item_tags_for_coll` – Whether to generate collection elements with name *`collection_name`*`_item`, where *`collection_name`* is the name of the collection.

These elements have the same meanings as corresponding parameters of procedures in PL/SQL package `DBMS_XMLGEN`.

Example 33–4 and Example 33–5 show the input and output of a simple SQL query.

Example 33–4 Input XML Document for SQL Query using Query Web Service

```
<?xml version="1.0" ?>
<env:Envelope xmlns:env="http://www.w3.org/2002/06/soap-envelope ">
  <env:Body>
    <query xmlns="http://xmlns.oracle.com/orawsv">
      <query_text type="SQL">
        <![CDATA[SELECT * FROM employees WHERE salary = :e]]>
      </query_text>
      <bind name="e">8300</bind>
      <pretty_print>false</pretty_print>
    </query>
  </env:Body>
</env:Envelope>
```

In Example 33–4, the query text is enclosed in `<![CDATA[...]]>`. Although not strictly necessary in this example, it is appropriate to do this generally, because queries often contain characters such as < and >. Element `bind` is used to bind a value (`8300`) to the bind variable named `e`. Element `pretty_print` turns off pretty-printing of the output.

Example 33–5 Output XML Document for SQL Query using Query Web Service

```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2002/06/soap-envelope">
<soap:Body>
<ROWSET><ROW><EMPLOYEE_ID>206</EMPLOYEE_ID><FIRST_NAME>William</FIRST_NAME><LAST_NAME>G
ietz</LAST_NAME><EMAIL>WGIETZ</EMAIL><PHONE_NUMBER>515.123.8181</PHONE_NUMBER><HIRE_DATE>07-JUN-
94</HIRE_DATE><JOB_ID>AC_ACCOUNT</JOB_ID><SALARY>8300</SALARY><MANAGER_ID>205</MANAGER_ID
><DEPARTMENT_ID>110</DEPARTMENT_ID></ROW></ROWSET>
</soap:Body>
</soap:Envelope>
```

Accessing PL/SQL Stored Procedures using a Web Service

The Oracle XML DB Web service for accessing PL/SQL stored functions and procedures is located at URL `http://`*`host`*`:`*`port`*`/`**`orawsv`**`/`*`dbschema`*`/`*`package`*`/`*`fn_or_proc`* or, for a function or procedure that is not in a package (standalone), `http://`*`host`*`:`*`port`*`/`**`orawsv`**`/`*`dbschema`*`/`*`fn_or_proc`*. Here, *`host`* and *`port`* are the host and HTTP(S) port properties of your database, *`fn_or_proc`* is the stored function or procedure name, *`package`* is the package it is in, and *`dbschema`* is the database schema owning that package.

The input XML document must contain the inputs needed by the function or procedure. The output XML document contains the return value and the values of all `OUT` variables.

The names of the XML elements in the input and output documents correspond to the variable names of the function or procedure. The generated WSDL document shows you the exact XML element names. This is the naming convention used:

- The XML element introducing the input to a PL/SQL function is named *`function-name`***`Input`**, where *`function-name`* is the name of the function (uppercase).
- The XML elements introducing input parameters for the function are named *`param-name`***`-`***`param-type`***`-`***`io-mode`*, where *`param-name`* is the name of the parameter (uppercase), *`param-type`* is its SQL data type, and *`io-mode`* is its input-output mode, as follows:
  - **`IN`** – `IN` mode
  - **`OUT`** – `OUT` mode
  - **`INOUT`** – `IN OUT` mode
- The XML element introducing the output from a PL/SQL function is named **`S`***`return-type`***`-`***`function-name`***`Output`**, where *`return-type`* is the SQL data type of the return value (uppercase), and *`function-name`* is the name of the function (uppercase).
- The XML elements introducing output parameters for the function are named the same as the output parameters themselves (uppercase). The element introducing the return value is named `RETURN`.

The return value of a function is in the `RETURN` element of the output document, which is always the first element in the document. This return-value position disambiguates it from any `OUT` parameter that might be named "`RETURN`".

Each stored function or procedure is associated with a separate, dynamic Web service that has its own, generated WSDL document. This WSDL document is located at URL `http://host:port/`**`orawsv`**`/dbschema/package/fn_or_proc`**`?wsdl`** or `http://host:port/`**`orawsv`**`/dbschema/fn_or_proc`**`?wsdl`**. In addition, you can optionally generate a single WSDL document to be used for all stored functions and procedures in a given package. The URL for that WSDL document is `http://host:port/`**`orawsv`**`/dbschema/package`**`?wsdl`**.

Data types in the incoming and outgoing XML documents are mapped to SQL data types for use by the stored function or procedure, according to Table 33–1.

Table 33–1 Web Service Mapping Between XML and SQL Data Types

SQL Data Type	XML Schema Data Type
`CHAR`, `VARCHAR2`, `VARCHAR`	`xsd:string`
`DATE` – Dates must be in the database format.	`xsd:date`
`TIMESTAMP`, `TIMESTAMP WITH TIMEZONE`, `TIMESTAMP WITH LOCAL TIMEZONE`	`xsd:dateTime`
`INTERVAL YEAR TO MONTH`, `INTERVAL DAY TO SECOND`	`xsd:duration`
`NUMBER`, `BINARY_DOUBLE`, `BINARY_FLOAT`	`xsd:double`
PL/SQL `BOOLEAN`	`xsd:boolean`
Object types	`complexType`

An object type is represented in XML as a complex-type element named the same as the object type. The object attributes are represented as children of this element.

Example of Using a PL/SQL Function with a Web Service

This section presents a simple PL/SQL function and its access using a Web service. The function takes as input a department ID and name, and it returns the salary total of all employees in the department. It also returns, as in-out and output parameters, respectively, the department name and the number of employees in the department. The default value of the department ID is `20`. In this simple example, the input value of the in-out parameter `dept_name` is not actually used. It is ignored, and the correct name is returned.

Example 33–6 shows the function definition. Example 33–7 shows the WSDL document that is created automatically from this function definition. Example 33–8 shows an input document that invokes the stored function. Example 33–9 shows the resulting output document.

Example 33–6 Definition of PL/SQL Function Used for Web-Service Access

```
CREATE OR REPLACE PACKAGE salary_calculator AUTHID CURRENT_USER AS
  FUNCTION TotalDepartmentSalary (dept_id    IN     NUMBER DEFAULT 20,
                                  dept_name  IN OUT VARCHAR2,
                                  nummembers OUT    NUMBER)
    RETURN NUMBER;
END salary_calculator;
/
```

```
CREATE OR REPLACE PACKAGE BODY salary_calculator AS
  FUNCTION TotalDepartmentSalary (dept_id    IN     NUMBER DEFAULT 20,
                                  dept_name  IN OUT VARCHAR2,
                                  nummembers OUT    NUMBER)
    RETURN NUMBER IS
      sum_sal NUMBER;
      BEGIN
        SELECT SUM(salary) INTO sum_sal FROM employees
          WHERE department_id = dept_id;
        SELECT department_name INTO dept_name FROM departments
          WHERE department_name = dept_name;
        SELECT count(*) INTO nummembers FROM employees
          WHERE department_id = dept_id;
    RETURN sum_sal;
  END;
END;
/
```

Example 33–7 WSDL Document Corresponding to a Stored PL/SQL Function

```
<definitions name="SALARY_CALCULATOR"
             targetNamespace="http://xmlns.oracle.com/orawsv/HR/SALARY_CALCULATOR"
             xmlns="http://schemas.xmlsoap.org/wsdl/"
             xmlns:tns="http://xmlns.oracle.com/orawsv/HR/SALARY_CALCULATOR"
             xmlns:xsd="http://www.w3.org/2001/XMLSchema"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/">
  <types>
    <xsd:schema targetNamespace="http://xmlns.oracle.com/orawsv/HR/SALARY_CALCULATOR"
                elementFormDefault="qualified">
      <xsd:element name="SNUMBER-TOTALDEPARTMENTSALARYInput">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="NUMMEMBERS-NUMBER-OUT">
              <xsd:complexType/>
            </xsd:element>
            <xsd:element name="DEPT_NAME-VARCHAR2-INOUT" type="xsd:string"/>
            <xsd:element name="DEPT_ID-NUMBER-IN" type="xsd:double"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="TOTALDEPARTMENTSALARYOutput">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="RETURN" type="xsd:double"/>
            <xsd:element name="NUMMEMBERS" type="xsd:double"/>
            <xsd:element name="DEPT_NAME" type="xsd:string"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:schema>
  </types>
  <message name="TOTALDEPARTMENTSALARYInputMessage">
    <part name="parameters" element="tns:SNUMBER-TOTALDEPARTMENTSALARYInput"/>
  </message>
  <message name="TOTALDEPARTMENTSALARYOutputMessage">
    <part name="parameters" element="tns:TOTALDEPARTMENTSALARYOutput"/>
  </message>
  <portType name="SALARY_CALCULATORPortType">
    <operation name="TOTALDEPARTMENTSALARY">
      <input message="tns:TOTALDEPARTMENTSALARYInputMessage"/>
```

```
      <output message="tns:TOTALDEPARTMENTSALARYOutputMessage"/>
    </operation>
  </portType>
 <binding name="SALARY_CALCULATORBinding" type="tns:SALARY_CALCULATORPortType">
    <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="TOTALDEPARTMENTSALARY">
      <soap:operation soapAction="TOTALDEPARTMENTSALARY"/>
      <input>
        <soap:body parts="parameters" use="literal"/>
      </input>
      <output>
        <soap:body parts="parameters" use="literal"/>
      </output>
    </operation>
  </binding>
  <service name="SALARY_CALCULATORService">
    <documentation>Oracle Web Service</documentation>
    <port name="SALARY_CALCULATORPort" binding="tns:SALARY_CALCULATORBinding">
      <soap:address location="httpS://example:8088/orawsv/HR/SALARY_CALCULATOR"/>
     </port>
  </service>
</definitions>
```

Example 33–8 Input XML Document for PL/SQL Query using Web Service

```
<?xml version="1.0" ?><soap:Envelope
xmlns:soap="http://www.w3.org/2002/06/soap-envelope"><soap:Body><SNUMBER-
TOTALDEPARTMENTSALARYinput
xmlns="http://xmlns.oracle.com/orawsv/HR/SALARY_CALCULATOR/TOTALDEPARTMENTSALARY">
<DEPT_ID-NUMBER-IN>30</DEPT_ID-NUMBER-IN><DEPT_NAME-VARCHAR2-INOUT>Purchasing
</DEPT_NAME-VARCHAR2-INOUT><NUMMEMBERS-NUMBER-OUT/></SNUMBER-
TOTALDEPARTMENTSALARYinput></soap:Body></soap:Envelope>
```

Example 33–9 Output XML Document for PL/SQL Query using Web Service

```
<?xml version="1.0" ?>
<soap:Envelope xmlns:soap="http://www.w3.org/2002/06/soap-envelope">
  <soap:Body>
    <TOTALDEPARTMENTSALARYOutput
       xmlns="http://xmlns.oracle.com/orawsv/HR/SALARY_CALCULATOR/TOTALDEPARTMENTSALARY">
      <RETURN>24900</RETURN>
      <NUMMEMBERS>6</NUMMEMBERS>
      <DEPT_NAME>Purchasing</DEPT_NAME>
    </TOTALDEPARTMENTSALARYOutput>
  </soap:Body>
</soap:Envelope>
```

Part VI

Oracle Tools that Support Oracle XML DB

Part VI of this manual provides information about Oracle tools that you can use with Oracle XML DB. It describes tools for managing Oracle XML DB, loading XML data, and exchanging XML data.

Part VI contains the following chapters:

- Chapter 34, "Administering Oracle XML DB"
- Chapter 35, "Loading XML Data using SQL*Loader"
- Chapter 36, "Exporting and Importing XMLType Tables"
- Chapter 37, "Exchanging XML Data using Oracle Streams AQ"

34

Administering Oracle XML DB

This chapter describes how to administer Oracle XML DB. It includes information about installing, upgrading, and configuring Oracle XML DB.

This chapter contains these topics:

- Installing Oracle XML DB
- Upgrading an Existing Oracle XML DB Installation
- Administering Oracle XML DB using Oracle Enterprise Manager
- Configuring Oracle XML DB using xdbconfig.xml

> **See Also:** "Configuring Resources for XLink and XInclude" on page 23-9 for information on configuring Oracle XML DB Repository resources for use with XLink and XInclude

Installing Oracle XML DB

You can perform a new installation of Oracle XML DB with or without Database Configuration Assistant (DBCA). You are required to install Oracle XML DB manually without DBCA if you upgrade an existing installation.

> **Caution:** Do *not* uninstall Oracle XML DB in a database that contains any `XMLType` data without first contacting Oracle Support.

Installing Oracle XML DB with Database Configuration Assistant

Oracle XML DB is part of the seed database and is installed by Database Configuration Assistant (DBCA) by default. No additional steps are required to install Oracle XML DB.

However, if you select the advanced database configuration, then you can configure the `SYSAUX` tablespace, which is used for Oracle XML DB Repository, and the port numbers for protocols FTP, HTTP(S), and WebDAV.

By default, DBCA performs the following tasks during installation:

- Creates tablespace `SYSAUX` for Oracle XML DB Repository
- Enables all protocol access

Tablespace `SYSAUX` holds the data stored in Oracle XML DB Repository, including data stored using:

- SQL, for example using `RESOURCE_VIEW` and `PATH_VIEW`

- Protocols such as FTP, HTTP(S), and WebDAV

You can store data in tables outside of this tablespace and access the data through Oracle XML DB Repository, by having `REF`s to that data stored in the tables in this tablespace.

> **See Also:** "Anonymous Access to Oracle XML DB Repository using HTTP" on page 28-17 for information about allowing unauthenticated access to the repository

Dynamic Protocol Registration of FTP and HTTP(S) Services with Local Listener

Oracle XML DB installation dynamically registers FTP and HTTP(S) services with the local listener. You can perform start, stop, and query with `lsnrctl`. For example:

- start: `lsnrctl start`
- stop: `lsnrctl stop`
- query: `lsnrctl status`

Changing FTP or HTTP(S) Port Numbers To change FTP and HTTP(S) port numbers, update elements `<ftp-port>`, `<http-port>`, and `<http2-port>` in file `/xdbconfig.xml` in Oracle XML DB Repository.

After updating the port numbers, dynamic protocol registration automatically stops FTP/HTTP(S) service on old port numbers and starts them on new port numbers if the local listener is up. If the local listener is not up, restart the listener after updating the port numbers.

> **See Also:** Chapter 28, "Accessing the Repository using Protocols" for information about configuring protocols

Post-installation Oracle XML DB uses dynamic protocol registration to set up FTP and HTTP listener services with the local listener. Ensure that the listener is up when you access Oracle XML DB protocols.

> **Note:** If the listener is running on a port that is not standard (for example, not 1521), then, in order for the protocols to register with the correct listener, the `init.ora` file must contain a `local_listener` entry. This references a `TNSNAME` entry that points to the correct listener. After editing the `init.ora` parameter, you must regenerate the `SPFILE` entry using `CREATE SPFILE`.

Installing Oracle XML DB Manually without DBCA

You can install Oracle XML DB manually, by running the `catqm` SQL script in directory `rdbms/admin` as database user `SYS`. Before running the script, you must install the database. If you expect Oracle XML DB Repository to contain a large amount of data, then you might also want to create a separate tablespace for Oracle XML DB.

Here is the syntax for `catqm`, where:

- *`xdb_password`* is the password
- *`xdb_ts_name`* is the tablespace to use for Oracle XML DB
- *`temp_ts_name`* is the temporary tablespace

- *secure_file_for_repo* is YES or NO (uppercase), YES meaning to use SecureFile LOB storage for Oracle XML DB Repository

```
catqm.sql xdb_password xdb_ts_name temp_ts_name secure_file_for_repos
```

For example:

```
catqm.sql change_on_install SYSAUX TEMP YES
```

Note:

- SecureFile LOB storage can be created only in a tablespace that uses automatic segment space management. See *Oracle Database Administrator's Guide*.
- Database compatibility must be 11.0.0.0 or greater, in order to use SecureFile LOB storage.

Note: Make sure that the database is started with Oracle9*i* release 2 (9.2.0) compatibility or higher and Java Virtual Machine (JVM) is installed.

Post-Installation

After the manual installation, carry out these tasks:

1. Add the following dispatcher entry to the `init.ora` file:

   ```
   dispatchers="(PROTOCOL=TCP) (SERVICE=<sid>XDB)"
   ```

2. Restart the database and the listener to enable Oracle XML DB protocol access.

See Also: "Anonymous Access to Oracle XML DB Repository using HTTP" on page 28-17 for information about allowing unauthenticated access to the repository

Upgrading an Existing Oracle XML DB Installation

The following considerations apply to all upgrades to Oracle Database 11*g*:

- Run script `catproc.sql`, as always.
- As a post-upgrade step, if you want Oracle XML DB functionality, then you must install Oracle XML DB manually as described in "Installing Oracle XML DB Manually without DBCA" on page 34-2.
- ACL security: In releases prior to Oracle Database 11*g* Release 1, conflicts among ACEs for the same principal and same privilege were resolved by giving priority to any ACE that had child `deny`, whether or not preceding ACEs had child `grant`, that is, ACE order did not matter. In Oracle Database 11*g* this `deny-trumps-grant` behavior is still available, but it is not the default behavior.

  See Also: "ACL and ACE Evaluation" on page 27-8 for information about conflicts among ACEs

Validation of ACL Documents and Configuration File

Access control list (ACL) documents are stored in table `XDB$ACL`. The Oracle XML DB configuration file, `xdbconfig.xml`, is stored in table `XDB$CONFIG`. Starting with Oracle Database 11*g* Release 1, these tables use the post-parse (binary XML) storage model. This implies that ACL documents and the configuration file are fully validated against their respective XML schemas. Validation takes place during upgrade, using your existing ACL documents and configuration file and the corresponding existing XML schemas.

If an ACL document fails to validate during upgrade, then the document is moved to table `XDB$INVALID_ACL`.

If validation of configuration file `xdbconfig.xml` fails during upgrade, then the file is saved in table `XDB$INVALID_CONFIG`, the default configuration file replaces it in table `XDB$CONFIG`, and the `XDB` component of the database is marked invalid. You must then start the database in normal mode and fix the `XDB` component, before trying to use the database.

To fix the `XDB` component, you can fix the invalid files to make them valid, and then call PL/SQL procedure `RecoverUpgrade`. After validating, this procedure moves the fixed files to tables `XDB$ACL` and `XDB$CONFIG`, and marks the `XDB` component valid.

As an option, you can call procedure `RecoverUpgrade` with parameter `use_default` set to `TRUE`, to abandon any invalid files. In this case, any valid files are moved to tables `XDB$ACL` and `XDB$CONFIG`, and any remaining invalid files are deleted. Default files are used in place of any invalid files. For ACLs, the default ACL document is used. For the configuration file, the default `xdbconfig.xml` is used (in which ACE order matters).

Caution: Use a `TRUE` value for parameter `use_default` *only* if you are certain that you no longer need the old ACL files or configuration file that are invalid. These files are *deleted*.

Administering Oracle XML DB using Oracle Enterprise Manager

Oracle Enterprise Manager is a graphical tool supplied with Oracle Database that lets you perform database administration tasks easily. You can use it to perform the following tasks related to Oracle XML DB:

- Configure Oracle XML DB. View or edit parameters for the Oracle XML DB configuration file, `/xdbconfig.xml`.

  For information about configuring Oracle XML DB without using Oracle Enterprise Manager, see "Configuring Oracle XML DB using xdbconfig.xml" on page 34-5.

- Search, create, edit, undelete Oracle XML DB Repository *resources* and their associated *access control lists* (ACLs).

  For information about creating and managing resources without using Oracle Enterprise Manager, see Part V, "Oracle XML DB Repository".

- Search, create, edit, and delete `XMLType` *tables* and *views*.

- Search, create, register, and delete *XML schemas*.

  For information about manipulating XML schemas without using Oracle Enterprise Manager, see Chapter 7, "XML Schema Storage and Query: Basic".

- Create *function-based indexes* based on XPath expressions.

  For information about creating function-based indexes without using Oracle Enterprise Manager, see Chapter 6, "Indexing XMLType Data".

  See Also: The online help available with Oracle Enterprise Manager, for information about using Enterprise Manager to perform these tasks

Configuring Oracle XML DB using xdbconfig.xml

Oracle XML DB is managed internally through a configuration file, `/xdbconfig.xml`, which is stored as a resource in Oracle XML DB Repository. As an alternative to using Oracle Enterprise Manager to configure Oracle XML DB, you can configure it directly using the Oracle XML DB configuration file.

The configuration file can be modified at run time. Updating the configuration file creates a new version of this repository resource. At the start of each session, the current version of the configuration file is bound to that session. The session uses this configuration-file version for its duration, unless you make an explicit call to refresh the session to the latest version.

Oracle XML DB Configuration File, xdbconfig.xml

The configuration of Oracle XML DB is defined and stored in an Oracle XML DB Repository resource, `/xdbconfig.xml`, which conforms to the Oracle XML DB configuration XML schema: `http://xmlns.oracle.com/xdb/xdbconfig.xsd`. To configure or reconfigure Oracle XML DB, update file `/xdbconfig.xml`. Its structure is described in the following sections. You need administrator privileges to access file `/xdbconfig.xml`.

See Also: "xdbconfig.xsd: XML Schema for Configuring Oracle XML DB" on page A-16 for a complete listing of the Oracle XML DB configuration XML schema

<xdbconfig> (Top-Level Element)

Element `<xdbconfig>` is the top-level element. Its structure is as follows:

```
<xdbconfig>
    <sysconfig> ... </sysconfig>
    <userconfig> ... </userconfig>
</xdbconfig>
```

Element `<sysconfig>` defines system-specific, built-in parameters. Element `<userconfig>` lets you store new custom parameters.

<sysconfig> (Child of <xdbconfig>)

Element `<sysconfig>` is a child of `<xdbconfig>`. Its structure is as follows:

```
<sysconfig>
    general parameters
    <protocolconfig> ... </protocolconfig>
</sysconfig>
```

Element `<sysconfig>` includes as content several general parameters that apply to all of Oracle XML DB, such as the maximum age of an access control list (ACL) and

whether or not Oracle XML DB is case sensitive. Child `<protocolconfig>` contains protocol-specific parameters.

<userconfig> (Child of <xdbconfig>)

Element `<userconfig>` is a child of `<xdbconfig>`. It contains any parameters that you may want to add.

<protocolconfig> (Child of <sysconfig>)

Element `<protocolconfig>` is a child of `<sysconfig>`. Its structure is as follows:

```
<protocolconfig>
  <common> ... </common>
  <ftpconfig> ... </ftpconfig>
  <httpconfig> ... </httpconfig>
</protocolconfig>
```

Under `<common>`, Oracle Database stores parameters that apply to all protocols, such as MIME-type information. Parameters that are specific to protocols FTP and HTTP(S) are in elements `<ftpconfig>` and `<httpconfig>`, respectively.

> **See Also:** Chapter 28, "Accessing the Repository using Protocols", Table 28–1, Table 28–2, and Table 28–3, for a list of protocol configuration parameters

<httpconfig> (Child of <protocolconfig>)

Element `<httpconfig>` is a child of `<protocolconfig>`. Its structure is as follows:

```
<httpconfig>
  ...
  <webappconfig>
    ...
    <servletconfig>
      ...
      <servlet-list>
        <servlet> ... </servlet>
        ...
      </servlet-list>
    </servletconfig>
  </webappconfig>
  ...
  <plsql> ... </plsql>
</httpconfig>
```

Element `<httpconfig>` has the following child elements, in addition to others:

- `<webappconfig>` – used to configure Web-based applications. This includes Web application-specific parameters, such as icon name, display name for the application, and a list of servlets.

  Element `<servletconfig>` is a child of `<webappconfig>` that is used to define servlets. It has child `<servlet-list>`, which has child `<servlet>` (see "<servlet> (Descendant of <httpconfig>)" on page 34-7).

- `<plsql>`[1] – used to define global configuration parameters when configuring the *embedded PL/SQL gateway*. Each global parameter is defined with a child element of

[1] There are two different `<plsql>` elements that are used to configure the embedded PL/SQL gateway. One, a child of `<httpconfig>`, defines *global parameters*. The other, a child of `<servlet>`, defines *DAD attributes*.

`<plsql>`. The element name is the same as the global parameter name. The element content is the same as the parameter value.

The recommended way to configure the embedded PL/SQL gateway is to use the procedures in PL/SQL package `DBMS_EPG`, *not* to edit file `xdbconfig.xml`.

See Also:

- Chapter 28, "Accessing the Repository using Protocols", Table 28–1, Table 28–2, and Table 28–3, for a list of protocol configuration parameters
- *Oracle Database Advanced Application Developer's Guide*, for complete information on configuring and using the embedded PL/SQL gateway
- *Oracle Fusion Middleware Administrator's Guide for Oracle HTTP Server* for information about mod_plsql and conceptual information about using the PL/SQL gateway
- *Oracle Database PL/SQL Packages and Types Reference*, for information about package `DBMS_EPG`

<servlet> (Descendant of <httpconfig>)

An optional `<plsql>` element, child of `<servlet>`, configures the embedded PL/SQL gateway servlet. However, the *recommended* way to configure the embedded gateway is to use the procedures in PL/SQL package `DBMS_EPG`, *not* to edit file `xdbconfig.xml`.

Element `<plsql>` has a child element for each embedded PL/SQL DAD attribute[2] that is needed to configure the embedded gateway. All such children are optional. The element name is the same as the DAD attribute name. The element content is the same as the DAD-attribute value.

Element `<servlet>` is a descendent of `<httpconfig>` – see "<httpconfig> (Child of <protocolconfig>)" on page 34-6. It is used to configure servlets, including Java servlets and embedded PL/SQL gateway servlets.

See Also:

- Chapter 32, "Writing Oracle XML DB Applications in Java" for information about configuring Java servlets
- *Oracle Database Advanced Application Developer's Guide*, for complete information on configuring and using the embedded PL/SQL gateway
- *Oracle Application Express Application Builder User's Guide*, for information about Oracle Application Express
- *Oracle Database PL/SQL Packages and Types Reference*, for information about package `DBMS_EPG`

Oracle XML DB Configuration File Example

The following is a sample Oracle XML DB configuration file:

[2] DAD is an abbreviation for Database Access Descriptor. DAD attributes are parameters that define such a descriptor.

Example 34–1 Oracle XML DB Configuration File

```
<xdbconfig xmlns="http://xmlns.oracle.com/xdb/xdbconfig.xsd"
                 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                 xsi:schemaLocation="http://xmlns.oracle.com/xdb/xdbconfig.xsd
                                     http://xmlns.oracle.com/xdb/xdbconfig.xsd">
   <sysconfig>
      <acl-max-age>900</acl-max-age>
      <acl-cache-size>32</acl-cache-size>
      <invalid-pathname-chars>,</invalid-pathname-chars>
      <case-sensitive>true</case-sensitive>
      <call-timeout>300</call-timeout>
      <max-link-queue>65536</max-link-queue>
      <max-session-use>100</max-session-use>
      <persistent-sessions>false</persistent-sessions>
      <default-lock-timeout>3600</default-lock-timeout>
      <xdbcore-logfile-path>/sys/log/xdblog.xml</xdbcore-logfile-path>
      <xdbcore-log-level>0</xdbcore-log-level>
      <resource-view-cache-size>1048576</resource-view-cache-size>

      <protocolconfig>
          <common>
             <extension-mappings>
                <mime-mappings>
                  <mime-mapping>
                    <extension>au</extension>
                    <mime-type>audio/basic</mime-type>
                  </mime-mapping>
                  <mime-mapping>
                    <extension>avi</extension>
                    <mime-type>video/x-msvideo</mime-type>
                  </mime-mapping>
                  <mime-mapping>
                    <extension>bin</extension>
                    <mime-type>application/octet-stream</mime-type>
                  </mime-mapping>
                </mime-mappings>

                <lang-mappings>
                  <lang-mapping>
                    <extension>en</extension>
                    <lang>english</lang>
                  </lang-mapping>
                </lang-mappings>

                <charset-mappings>
                </charset-mappings>

                <encoding-mappings>
                  <encoding-mapping>
                    <extension>gzip</extension>
                    <encoding>zip file</encoding>
                  </encoding-mapping>
                  <encoding-mapping>
                    <extension>tar</extension>
                    <encoding>tar file</encoding>
                  </encoding-mapping>
                </encoding-mappings>
             </extension-mappings>

             <session-pool-size>50</session-pool-size>
```

```
      <session-timeout>6000</session-timeout>
    </common>

    <ftpconfig>
      <ftp-port>2100</ftp-port>
      <ftp-listener>local_listener</ftp-listener>
      <ftp-protocol>tcp</ftp-protocol>
      <logfile-path>/sys/log/ftplog.xml</logfile-path>
      <log-level>0</log-level>
      <session-timeout>6000</session-timeout>
      <buffer-size>8192</buffer-size>
    </ftpconfig>

    <httpconfig>
      <http-port>8080</http-port>
      <http-listener>local_listener</http-listener>
      <http-protocol>tcp</http-protocol>
      <max-http-headers>64</max-http-headers>
      <session-timeout>6000</session-timeout>
      <server-name>XDB HTTP Server</server-name>
      <max-header-size>16384</max-header-size>
      <max-request-body>2000000000</max-request-body>
      <logfile-path>/sys/log/httplog.xml</logfile-path>
      <log-level>0</log-level>
      <servlet-realm>Basic realm="XDB"</servlet-realm>
      <webappconfig>
        <welcome-file-list>
          <welcome-file>index.html</welcome-file>
          <welcome-file>index.htm</welcome-file>
        </welcome-file-list>
        <error-pages>
        </error-pages>
        <servletconfig>
          <servlet-mappings>
            <servlet-mapping>
              <servlet-pattern>/oradb/*</servlet-pattern>
              <servlet-name>DBURIServlet</servlet-name>
            </servlet-mapping>
          </servlet-mappings>

          <servlet-list>
            <servlet>
              <servlet-name>DBURIServlet</servlet-name>
              <display-name>DBURI</display-name>
              <servlet-language>C</servlet-language>
              <description>Servlet for accessing DBURIs</description>
              <security-role-ref>
                <role-name>authenticatedUser</role-name>
                <role-link>authenticatedUser</role-link>
              </security-role-ref>
            </servlet>
          </servlet-list>
        </servletconfig>
      </webappconfig>
    </httpconfig>
  </protocolconfig>
 <xdbcore-xobmem-bound>1024</xdbcore-xobmem-bound>
 <xdbcore-loadableunit-size>16</xdbcore-loadableunit-size>
</sysconfig>
</xdbconfig>
```

Oracle XML DB Configuration API

You can access the Oracle XML DB configuration file, `xdbconfig.xml`, the same way you access any other XML schema-based resource in the hierarchy. It can be accessed using FTP, HTTP(S), WebDAV, Oracle Enterprise Manager, or any of the resource and Document Object Model (DOM) APIs for Java, PL/SQL, or C (OCI).

For convenience, you can use PL/SQL package `DBMS_XDB` package for configuration access. It exposes the following functions and procedures:

- `cfg_get` – Returns the configuration information for the current session.
- `cfg_refresh` – Refreshes the session configuration information using the current configuration file. Typical uses of `cfg_refresh` include the following:
  - You have modified the configuration and now want the session to pick up the latest version of the configuration information.
  - It has been a long running session, the configuration has been modified by a concurrent session, and you want the current session to pick up the latest version of the configuration information.
- `cfg_update` – Updates the configuration information, writing the configuration file. A `COMMIT` is performed.

Example 34–2 updates parameters `ftp-port` and `http-port` in the configuration file.

Example 34–2 Updating the Configuration File using CFG_UPDATE and CFG_GET

```
DECLARE
  v_cfg XMLType;
BEGIN
  SELECT updateXML(DBMS_XDB.cfg_get(),
                   '/xdbconfig/descendant::ftp-port/text()',
                   '2121',
                   '/xdbconfig/descendant::http-port/text()',
                   '19090')
    INTO v_cfg FROM DUAL;
  DBMS_XDB.cfg_update(v_cfg);
  COMMIT;
END;
/
```

If you have many parameters to update, then it can be easier to use FTP, HTTP(S), or Oracle Enterprise Manager to update the configuration.

> **See Also:** *Oracle Database PL/SQL Packages and Types Reference*

Configuring Default Namespace to Schema Location Mappings

Oracle XML DB identifies schema-based `XMLType` instances by pre-parsing the input XML document. If the appropriate `xsi:schemaLocation` or `xsi:noNamespaceSchemaLocation` attribute is found, then the specified schema location URL is used to consult the registered schema. If the appropriate `xsi:` attribute is not found, the XML document is considered to be non-schema-based.

Oracle XML DB provides a mechanism to configure default schema location mappings. If the appropriate `xsi:` attribute is not specified in the XML document, the default schema location mappings is used. Element `schemaLocation-mappings` of the Oracle XML DB configuration XML schema, `xdbconfig.xsd`, can be used to specify the mapping between *`(namespace, element)`* pairs and the default schema

location. If the *element* value is empty, the mapping applies to all global elements in the specified namespace. If the *namespace* value is empty, it corresponds to the null namespace.

The definition of the `schemaLocation-mappings` element is as follows:

```
<element name="schemaLocation-mappings"
         type="xdbc:schemaLocation-mapping-type" minOccurs="0"/>

<complexType name="schemaLocation-mapping-type"><sequence>
     <element name="schemaLocation-mapping"
              minOccurs="0" maxOccurs="unbounded">
     <complexType><sequence>
       <element name="namespace" type="string"/>
       <element name="element" type="string"/>
       <element name="schemaURL" type="string"/>
     </sequence></complexType>
     </element></sequence>
</complexType>
```

The schema location used depends on mappings in the Oracle XML DB configuration file for the namespace used and the root document element. For example, assume that the document does not have the appropriate `xsi:` attribute to indicate the schema location. Consider a document root element *R* in namespace *N*. The algorithm for identifying the default schema location is as follows:

1. If the Oracle XML DB configuration file has a mapping for *N* and *R*, the corresponding schema location is used.
2. If the configuration file has a mapping for *N*, but not *R*, the schema location for *N* is used.
3. If the document root *R* does not have any namespace, the schema location for *R* is used.

For example, suppose that your Oracle XML DB configuration file includes the following mapping:

```
<schemaLocation-mappings>
   <schemaLocation-mapping>
      <namespace>http://www.oracle.com/example</namespace>
      <element>root</element>
      <schemaURL>http://www.oracle.com/example/sch.xsd</schemaURL>
   </schemaLocation-mapping>
   <schemaLocation-mapping>
      <namespace>http://www.oracle.com/example2</namespace>
      <element></element>
      <schemaURL>http://www.oracle.com/example2/sch.xsd</schemaURL>
   </schemaLocation-mapping>
   <schemaLocation-mapping>
     <namespace></namespace>
     <element>specialRoot</element>
     <schemaURL>http://www.oracle.com/example3/sch.xsd</schemaURL>
   </schemaLocation-mapping>
</schemaLocation-mappings>
```

The following schema locations are used:

- Namespace = `http://www.oracle.com/example`
  Root Element = `root`
  Schema URL = `http://www.oracle.com/example/sch.xsd`

This mapping is used when the instance document specifies:

```
<root xmlns="http://www.oracle.com/example">
```

- Namespace = `http://www.oracle.com/example2`
  Root Element = `null` (any global element in the namespace)
  Schema URL = `http://www.oracle.com/example2/sch.xsd`

  This mapping is used when the instance document specifies:

  ```
  <root xmlns="http://www.oracle.example2">
  ```

- Namespace = `null` (i.e null namespace)
  Root Element = `specialRoot`
  Schema URL = `http://www.oracle.com/example3/sch.xsd`

  This mapping is used when the instance document specifies:

  ```
  <specialRoot>
  ```

> **Note:** This functionality is available only on the server side, that is, when XML is parsed on the server. If XML is parsed on the client side, the appropriate `xsi:` attribute is still required.

Configuring XML File Extensions

Oracle XML DB Repository treats certain files as XML documents, based on their file extensions. When such files are inserted into the repository, Oracle XML DB pre-parses them to identify the schema location (or uses the default mapping if present) and inserts the document into the appropriate default table.

By default, the following extensions are considered as XML file extensions: **`xml`**, **`xsd`**, **`xsl`**, **`xlt`**. In addition, Oracle XML DB provides a mechanism for applications to specify other file extensions as XML file extensions. The **`xml-extensions`** element is defined in the configuration schema, `http://xmlns.oracle.com/xdb/xdbconfig.xsd`, as follows:

```
<element name="xml-extensions"
         type="xdbc:xml-extension-type" minOccurs="0"/>


<complexType name="xml-extension-type"><sequence>
     <element name="extension" type="xdbc:exttype"
              minOccurs="0" maxOccurs="unbounded">
     </element></sequence>
 </complexType>
```

For example, the following fragment from the Oracle XML DB configuration file, `xdbconfig.xml`, specifies that files with extensions `vsd`, `vml`, and `svgl` should be treated as XML files:

```
<xml-extensions>
  <extension>vsd</extension>
  <extension>vml</extension>
  <extension>svgl</extension>
</xml-extensions>
```

35

Loading XML Data using SQL*Loader

This chapter describes how to load XML data into Oracle XML DB with a focus on SQL*Loader.

This chapter contains these topics:

- Overview of Loading XMLType Data Into Oracle Database
- Loading XMLType Data using SQL*Loader
- Loading Large XML Documents into Oracle Database

See Also: Chapter 3, "Using Oracle XML DB"

Overview of Loading XMLType Data Into Oracle Database

Starting with Oracle9*i* release 1 (9.0.1), the Export-Import utility and SQL*Loader support `XMLType` as a column type. Starting with Oracle Database 10*g*, SQL*Loader also supports loading `XMLType` tables. You can load `XMLType` data with SQL*Loader using either the conventional method or the direct-path method, regardless of how it is stored (structured, unstructured, or binary XML storage).

Note: For *structured storage* of XML data, if the data involves *inheritance* (extension or restriction) of XML Schema types, then SQL*Loader does *not* support direct-path loading.

That is, if an XML schema contains a `complexType` element that extends or restricts another `complexType` element (the base type), then this results in some SQL types being defined in terms of other SQL types. In this case, direct-path loading is not supported for object-relational storage.

See Also: Chapter 36, "Exporting and Importing XMLType Tables" and *Oracle Database Utilities*

Oracle XML DB Repository information is *not* exported when user data is exported. Neither the resources nor any information are exported.

Loading XMLType Data using SQL*Loader

XML columns are columns declared to be of type `XMLType`.

SQL*Loader treats `XMLType` columns and tables like object-relational columns and tables. All methods described in the following sections for loading LOB data from the primary datafile or from a `LOBFILE` value apply also to loading `XMLType` columns and tables when the `XMLType` data is stored as a LOB.

> **See Also:** *Oracle Database Utilities*

> **Note:** You cannot specify a SQL string for LOB fields. This is true even if you specify `LOBFILE_spec`.

`XMLType` data can be present in a control file or in a LOB file. In the former case, the LOB file name is present in the control file.

Because `XMLType` data can be quite large, SQL*Loader can load LOB data from either a primary datafile (in line with the rest of the data) or from LOB files, independent of how the data is stored (the underlying storage can, for example, still be object-relational).

Loading XMLType Data in LOBs using SQL*Loader

To load internal LOBs, Binary Large Objects (BLOBs), Character Large Objects (CLOBs), and National Character Large Object (NCLOBs), or `XMLType` columns and tables from a primary datafile, use the following standard SQL*Loader formats:

- Predetermined size fields
- Delimited fields
- Length-value pair fields

These formats are described in the following sections and in more detail in *Oracle Database Utilities.*

Loading LOB Data in Predetermined Size Fields

This is a very fast and conceptually simple format to load LOBs.

> **Note:** Because the LOBs you are loading might not be of equal size, you can use whitespace to pad the LOB data to make the LOBs all of equal length within a particular data field.

Loading LOB Data in Delimited Fields

This format handles LOBs of different sizes within the same column (datafile field) without problem. However, this added flexibility can affect performance, because SQL*Loader must scan through the data, looking for the delimiter string.

As with single-character delimiters, when you specify string delimiters, you should consider the character set of the datafile. When the character set of the datafile is different than that of the control file, you can specify the delimiters in hexadecimal (that is, hexadecimal string). If the delimiters are specified in hexadecimal notation, then the specification must consist of characters that are valid in the character set of the input datafile. In contrast, if hexadecimal specification is not used, then the delimiter specification is considered to be in the client (that is, the control file) character set. In this case, the delimiter is converted into the datafile character set before SQL*Loader searches for the delimiter in the datafile.

Loading XML Columns Containing LOB Data from LOBFILEs

LOB data can be lengthy enough so that it makes sense to load it from a LOBFILE instead of from a primary datafile. In LOBFILEs, LOB data instances are still considered to be in fields (predetermined size, delimited, length-value), but these fields are not organized into records (the concept of a record does not exist within LOBFILEs). Therefore, the processing overhead of dealing with records is avoided. This type of organization of data is ideal for LOB loading.

There is no requirement that a LOB from a LOBFILE fit in memory. SQL*Loader reads LOBFILEs in 64 KB chunks.

In LOBFILEs the data can be in any of the following types of fields, any of which can be used to load XML columns:

- A single LOB field into which the entire contents of a file can be read
- Predetermined size fields (fixed-length fields)
- Delimited fields (that is, `TERMINATED BY` or `ENCLOSED BY`)

  The clause `PRESERVE BLANKS` is not applicable to fields read from a LOBFILE.
- Length-value pair fields (variable-length fields).

  To load data from this type of field, use the `VARRAY`, `VARCHAR`, or `VARCHAR2` SQL*Loader data types.

Specifying LOBFILEs

You can specify LOBFILEs either statically (you specify the name of the file) or dynamically (you use a FILLER field as the source of the filename). In either case, when the EOF of a LOBFILE is reached, the file is closed and additional attempts to read data from that file produce results equivalent to reading data from an empty field.

You should not specify the same LOBFILE as the source of two different fields. If you do so, then typically, the two fields read the data independently.

Loading XMLType Data Directly from a Control File using SQL*Loader

`XMLType` data can be loaded directly from a control file. SQL*Loader treats `XMLType` data like any scalar type. For example, consider a table containing a `NUMBER` column followed by an `XMLType` column that is stored object-relationally. The control file used for this table can contain the value of the `NUMBER` column followed by the value of the `XMLType` instance.

SQL*Loader accommodates `XMLType` instances that are very large. You also have the option to load such data from a LOB file.

Loading Large XML Documents into Oracle Database

You can use SQL*Loader to load large amounts of XML data into Oracle Database. Follow these steps:

1. List in a data file, say `filelist.dat`, the locations of the XML documents to be loaded.
2. Create a control file, say `load_data.ctl`, with commands that process the files listed in the data file.
3. Invoke the SQL*Loader shell command, `sqlldr`, passing it the name of the control file.

This is illustrated in Example 35–2, Example 35–1, and Example 35–3. File `filelist.dat` lists XML files that contain purchase orders for the year 2002.

Example 35–1 Data File filelist.dat: List of XML Files to Load

```
2002/Jan/AMCEWEN-20021009123335370PDT.xm
2002/Jan/AWALSH-200210091233357OPDT.xml
2002/Jan/CJOHNSON-20021009123335170PDT.xml
2002/Jan/LSMITH-20021009123335500PDT.xml
2002/Jan/PTUCKER-20021009123335430PDT.xml
2002/Jan/SBELL-20021009123335280PDT.xml
2002/Jan/SKING-20021009123335560PDT.xml
2002/Jan/SMCCAIN-20021009123335470PDT.xml
2002/Jan/TFOX-20021009123335520PDT.xml
2002/Jan/VJONES-20021009123335350PDT.xml
2002/Jan/WSMITH-20021009123335450PDT.xml
2002/Feb/AMCEWEN-20021009123335600PDT.xml
2002/Feb/AMCEWEN-20021009123335701PDT.xml
2002/Feb/DAUSTIN-20021009123335811PDT.xml
2002/Feb/EABEL-20021009123335791PDT.xml
2002/Feb/PTUCKER-20021009123335721PDT.xml
2002/Feb/PTUCKER-20021009123335821PDT.xml
2002/Feb/SBELL-20021009123335771PDT.xml
2002/Feb/SMCCAIN-20021009123335681PDT.xml
2002/Feb/WSMITH-20021009123335650PDT.xml
2002/Feb/WSMITH-20021009123335741PDT.xml
2002/Feb/WSMITH-20021009123335751PDT.xml
...
```

Example 35–2 Control File load_datra.ctl, for Loading Purchase-Order XML Documents

```
load data
infile 'filelist.dat'
append
into table PURCHASEORDER
xmltype(XMLDATA)
(
 filename filler char(120),
 XMLDATA  lobfile(filename) terminated by eof
)
```

Example 35–3 Loading XML Data Using Shell Command sqlldr

```
sqlldr load_data.ctl
```

For direct-path loading, use this instead:

```
sqlldr load_data.ctl direct=y
```

See Also:

- Chapter 3, "Using Oracle XML DB", "Loading Large XML Files using SQL*Loader" on page 3-9
- *Oracle Database Utilities* for information about shell command `sqlldr`

36

Exporting and Importing XMLType Tables

This chapter describes how you can export and import `XMLType` tables for use with Oracle XML DB using Oracle Data Pump.

> **Note:** The original Export and Import utilities can be used to migrate data to database releases that are prior to Oracle Database 11*g* Release 2 (11.2).

This chapter discusses the following topics:

- Overview of Oracle Data Pump
- EXPORT/IMPORT Support in Oracle XML DB
- Exporting XML Schema-Based XMLType Tables
- Exporting Hierarchy-Enabled (Repository) Tables
- Exporting and Importing Transportable Tablespaces
- Repository Resources and Foldering Support
- Export/Import Syntax and Examples

Overview of Oracle Data Pump

Oracle Data Pump technology enables high-speed movement of data and metadata from one database to another. Oracle Data Pump has two command-line clients, `expdp` and `impdp`, that invoke Data Pump Export utility and Data Pump Import utility, respectively. The `expdp` and `impdp` clients use procedures provided in PL/SQL package `DBMS_DATAPUMP` to execute export and import commands, passing the parameters entered at the command-line. These parameters enable the exporting and importing of data and metadata for a complete database or subsets of a database.

The Data Pump Export and Import utilities (invoked with commands `expdp` and `impdp`, respectively) have a similar look and feel to the original Export (`exp`) and Import (`imp`) utilities, but they are completely separate.

> **See Also:** *Oracle Database Utilities*, for information about situations in which you should still use the original Export and Import utilities.

Data Pump Export utility (invoked with `expdp`) unloads data and metadata into a set of operating system files called a *dump file set*. The dump file set can be imported only by the Data Pump Import utility (invoked with `impdp`).

EXPORT/IMPORT Support in Oracle XML DB

Oracle XML DB supports export and import of `XMLType` tables and columns that store XML data.

You can export and import this data regardless of the `XMLType` storage format (structured, unstructured, or binary XML). However, Data Pump exports and imports XML data as text or binary XML data only. The underlying object-relational tables and columns used for structured storage of `XMLType` are thus not exported. Instead, they are converted to binary form and then exported as self-describing binary XML data. `XMLType` data stored as `CLOB` instances (unstructured storage) is exported as text.

> **Note:** Oracle Data Pump for Oracle Database 11*g* Release 1 (11.1) does not support the export of XML schemas, XML schema-based `XMLType` columns, or binary XML data to database releases prior to 11.1.

Regardless of the export format, the format of the dump file is either `CLOB` or self-describing binary XML with a token map preamble. How Oracle Data Pump stores this data in the dump file depends on the value of the export parameter, `data_options` (the only valid value for this parameter is `xml_clobs.`) If you specify this value on the export command line, all `XMLType` data is stored in text format in the dump file. If you do not specify the `xml_clobs` parameter in the `expdp` command, then the format of the `XMLType` columns in the table determines the format of the data in the dump file. Table 36–1 shows the format of the `XMLType` columns in the table with the corresponding format of the dump file.

Table 36–1 Format of the XMLType columns in the table with the corresponding format of the dump file

Storage Model of XMLType Columns	Dump File Format of XML Data
Unstructured storage	Text
Structured storage, binary XML storage, or hybrid storage (a mixture of unstructured and structured storage)	Self-describing binary XML

Since `XMLType` data is exported and imported as XML data, the source and target databases can use different `XMLType` storage models for that data. You can export data from a database that stores `XMLType` data one way and import it into a database that stores `XMLType` data a different way.

Exporting XML Schema-Based XMLType Tables

You can export `XMLType` tables, whether they are XML schema-based or not. If a table is XML schema-based, then it depends on the XML schema used to define its data. This XML schema can also have dependencies on SQL object types that are used to store the data, in the case of structured storage. Therefore, exporting a user who has XML schema-based `XMLType` tables also exports the following:

- SQL objects types (if structured storage was used)
- XML schemas
- XML tables

Exporting Hierarchy-Enabled (Repository) Tables

The following guidelines apply to exporting hierarchy-enabled tables, that is, tables that underlie Oracle XML DB Repository data:

- The row-level security (RLS) policies and path-index triggers are not exported for hierarchy-enabled tables. When these tables are imported, they are *not* hierarchy-enabled.
- Hidden columns `ACLOID` and `OWNERID` are *not* exported for these tables. In an imported database the values of these columns could be different, so they should be reinitialized.

> **See Also:** "Repository Resources and Database Table Security" on page 27-18 for information about RLS policies and path-index triggers

Exporting and Importing Transportable Tablespaces

Using the *transportable tablespace* feature, you can move a set of tablespaces from one Oracle database to another, whether it is XML schema-based or non-schema-based. When you export using transportable tablespaces mode, only the metadata for tables (and their dependent objects) within a specified set of tablespaces are unloaded. You can then copy the tablespace data files to another Oracle database and perform a transport tablespace import. This is generally very fast, because it involves only copying the tablespace and re-creating the tablespace metadata.

Use `TRANSPORT_TABLESPACES` parameter in `expdp` to specify a list of tablespace names for which the object metadata is to be exported from the source database to the target database.

You cannot export transportable tablespaces and import them to a database at a lower release level. The target database must be at the same or higher release level as the source database.

When *exporting*, Oracle XML DB Repository hierarchy information is lost (see "Exporting Hierarchy-Enabled (Repository) Tables" on page 36-3). When *importing*, any XML schemas referenced by the data to be imported are also imported.

Repository Resources and Foldering Support

Oracle XML DB supports a *foldering mechanism* that helps store content in the database in hierarchical structures, as opposed to traditional relational database structures. Foldering lets you use path names and URIs to refer to data (repository resources), rather than table names, column names, and so on. This foldering mechanism is not entirely supported using `expdp` or `impdp`.

However, for resources based on a registered XML schema, the `XMLType` tables storing the data can be exported and imported. During export, only the XML data is exported, the relationship in the Oracle XML DB foldering hierarchy is lost.

Full Database Export

Oracle XML DB stores the metadata (and data unrelated to XML Schema) for Oracle XML DB Repository in database schema (user account) `XDB`. Because Oracle Database does not export the repository structure, these metadata tables and structures are not exported during a full database export.

The entire database schema (user) `XDB` is skipped during a full database export, and any database objects owned by user `XDB` are not exported.

Exporting and Importing with Different Character Sets

As with other database objects, XML data is exported in the character set of the exporting server. During import, the data gets converted to the character set of the importing server.

Export/Import Syntax and Examples

Export and import using Oracle Data Pump is described in *Oracle Database Utilities*. This section includes additional guidelines and examples for using commands `expdp` and `impdp` with `XMLType` data. For tables with `XMLType` data stored as `CLOB`, Oracle Data Pump exports and imports the tables in the same way as it would do for any table.

The examples presented here use the command-line commands `expdp` and `impdp`. After submitting such a command with a user name and command parameters, you are prompted for a password. The examples here do not show this prompting.

Performing a Table-Mode Export /Import

An `XMLType` table has a dependency on the XML schema that was used to define it. Similarly, the XML schema has dependencies on the SQL object types created or specified for it. Importing an `XMLType` table requires the existence of the XML schema and the SQL object types. When a `TABLE` mode export is used, only the table related metadata and data are exported. To be able to import this data successfully, the user needs to ensure that both the XML schema and object types have been created.

The examples here assume that you are using a database with the following features:

- A database with a sample schema
- A table `foo` with an `XMLType` column in unstructured (`CLOB`) storage format
- A directory object `dpump_dir`, for which `READ` and `WRITE` privileges have been granted to the sample schema

Example 36–1 shows a table-mode export, specified using the `TABLES` parameter. It exports table `foo` to `foo.dmp` dump file.

Example 36–1 Exporting XMLType Data in TABLE Mode

```
expdp system directory=dpump_dir dumpfile=foo.dmp tables=foo
```

> **Note:** In table mode, if you do not specify a schema prefix in the `expdp` command, the schema of the exporter is used by default.

Example 36–2 shows a table-mode import. It imports table `foo` from the `foo.dmp` dump file.

Example 36–2 Importing XMLType Data in TABLE Mode

```
impdp system tables=foo directory=dpump_dir dumpfile=foo.dmp table_exists_action=append
```

If a table by the name `foo` already exists at the time of this import, then parameter `table_exists_action` appends rows at the end of the existing table. When you use `APPEND`, the data is always loaded into new space. Existing space, even if available, is never reused. For this reason, you might need to compress your data after the load.

See Also: *Oracle Database Utilities*, for more information about Oracle Data Pump and its command-line clients, `expdp` and `impdp`

Performing a Schema-Mode Export/Import

When performing a `Schema` mode export, if you have role `EXP_FULL_DATABASE`, then you can export a database schema, the database schema definition, and the system grants and privileges of that database schema.

The example here assumes that you are using a database with the following features:

- User `x4a` has created a table `po2`.
- User `x4a` has a registered XML schema, `ipo`, which created two ordered collection tables `item_oct2` and `sitem_nt2`.

User `x4a` creates table `po2` as shown in Example 36–3.

Example 36–3 Creating Table po2

```
CREATE TABLE po2 (po XMLType)
  XMLTYPE COLUMN po
  XMLSCHEMA "ipo.xsd"
  ELEMENT "purchaseOrder"
  VARRAY po.XMLDATA."items"."item"
    STORE AS TABLE item_oct2 ((PRIMARY KEY(NESTED_TABLE_ID, SYS_NC_ARRAY_INDEX$)))
      NESTED TABLE po.XMLDATA."shippedItems"."item" STORE AS sitem_nt2;
```

Table `po2` is then populated and exported, as shown in Example 36–4.

Example 36–4 Exporting XMLType Data in SCHEMA Mode

```
expdp x4a directory=tkxm_xmldir dumpfile=x4a.dmp
```

Example 36–4 exports all of the following:

- All data types that were generated during registration of XML schema `ipo`.
- XML schema `ipo`.
- Table `po2` and the ordered collection tables `item_oct2` and `sitem_nt2`, which were generated during registration of XML schema `ipo`.
- All data in all of those tables.

Example 36–5 Importing XMLType Data in SCHEMA Mode

```
impdp x4a directory=tkxm_xmldir dumpfile=x4a.dmp
```

Example 36–5 imports all of the data in `x4a.dmp` to another database, in which the user `x4a` already exists.

Example 36–6 does the same thing as Example 36–5, but it also remaps the database schema from user `x4a` to user `quine`.

Example 36–6 Importing XMLType Data in SCHEMA Mode, Remapping Schema

```
impdp x4a directory=tkxm_xmldir dumpfile=x4a.dmp remap_schema=x4a:quine
```

Example 36–6 imports all of the data in `x4a.dmp` (exported from the database schema of user `x4a`) into database schema `quine`. To remap the database schema, user `x4a` must have been granted role `IMP_FULL_DATABASE` on the local database and role

`EXP_FULL_DATABASE` on the source database. `REMAP_SCHEMA` loads all of the objects from the source schema into the target schema.

37

Exchanging XML Data using Oracle Streams AQ

Oracle Streams Advanced Queuing (AQ) provides database-integrated message-queuing:

- It enables and manages asynchronous communication of two or more applications, using messages
- It supports point-to-point and publish/subscribe communication models

Integration of message queuing with Oracle Database brings the integrity, reliability, recoverability, scalability, performance, and security features of Oracle Database to message queuing. It also facilitates the extraction of intelligence from message flows.

This chapter describes how XML data can be exchanged using AQ. It contains these topics:

- How Do AQ and XML Complement Each Other?
- Oracle Streams and AQ
- XMLType Attributes in Object Types
- Internet Data Access Presentation (iDAP)
- iDAP Architecture
- Guidelines for Using XML and Oracle Streams Advanced Queuing

How Do AQ and XML Complement Each Other?

XML has emerged as a standard format for business communications. XML is being used not only for data communicated between business applications, but also to represent business logic.

In Oracle Database, AQ supports native XML messages. It lets AQ operations be defined using the XML-based Internet-Data-Access-Presentation (iDAP) format. iDAP is an extensible message invocation protocol. It is built on Internet standards, using HTTP(S) and e-mail protocols as the transport mechanism. XML is the data representation language for iDAP.

AQ and XML Message Payloads

Figure 37–1 shows an Oracle database using AQ to communicate with three applications. The message payload is XML data. The general tasks performed by AQ in this scenario are:

- Message flow using subscription rules
- Message management
- Extraction of business intelligence from messages
- Message transformation

XML messages are passed asynchronously among applications using AQ.

- Intra-business. Typical examples include sales order fulfillment and supply-chain management.
- Inter-business. Multiple integration hubs can communicate over the Internet. Examples include travel reservations, coordination between manufacturers and suppliers, transfer of funds between banks, and insurance claims settlements.

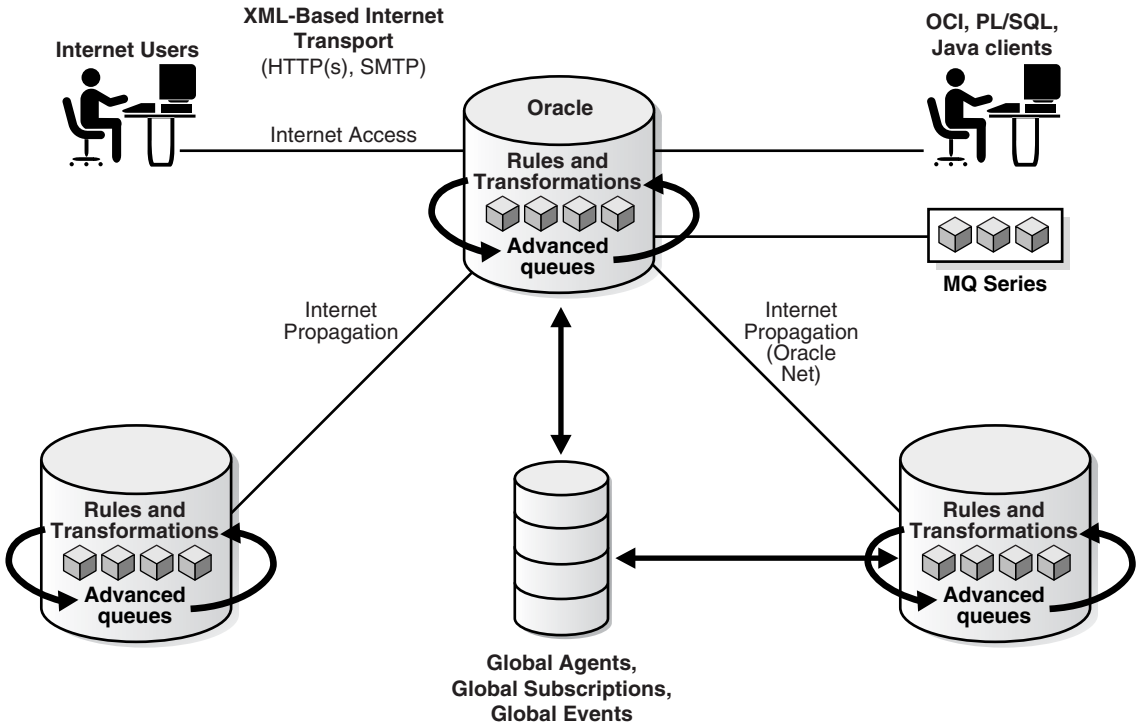
  Oracle uses this approach in its enterprise application integration products. XML messages are sent from applications to an Oracle AQ hub. The hub serves as a message server for any application that wants the message. Through this hub-and-spoke architecture, XML messages can be communicated asynchronously to multiple loosely coupled applications.

Figure 37–1 shows XML payload messages transported using AQ in the following ways:

- A Web-based application uses an AQ operation over an HTTP(S) connection using iDAP
- An application uses AQ to propagate an XML message over a Net* connection
- An application uses AQ to propagate an Internet or XML message directly to the database using HTTP(S) or SMTP

Figure 37–1 also shows that AQ clients can access data using OCI, Java, or PL/SQL.

Figure 37–1 Oracle Streams Advanced Queuing and XML Message Payloads



Advantages of Using AQ

AQ provides flexibility in configuring communication between different applications. It makes an integrated solution easy to manage, easy to configure, and easy to modify, to meet changing business needs. It enables multiple applications to cooperate, coordinate, and synchronize, to carry out complex business transactions.

Message management provided by AQ manages the flow of messages between different applications. AQ can also retain messages for auditing and tracking purposes, and for extracting business intelligence.

AQ provides SQL views to access messages. You can use these views to analyze trends.

Oracle Streams and AQ

Oracle Streams (Streams) enables you to share data and events in a stream. The stream can propagate this information within a database or from one database to another. The stream routes specified information to specified destinations. This provides greater functionality and flexibility than traditional solutions for capturing and managing events, and sharing the events with other databases and applications.

Streams enables you to break the cycle of trading off one solution for another. It enable you to build and operate distributed enterprises and applications, data warehouses, and high availability solutions. You can use all the capabilities of Oracle Streams at the same time.

You can use Streams to:

- *Capture changes at a database.* You can configure a background capture process to capture changes made to tables, database schemas, or the entire database. A capture process captures changes from the redo log and formats each captured

change into a logical change record (LCR). The database where changes are generated in the redo log is called the source database.

- *Enqueue events into a queue.* Two types of events may be staged in a Streams queue: LCRs and user messages. A capture process enqueues LCR events into a queue that you specify. The queue can then share the LCR events within the same database or with other databases. You can also enqueue user events explicitly with a user application. These explicitly enqueued events can be LCRs or user messages.
- *Propagate events from one queue to another.* These queues may be in the same database or in different databases.
- *Dequeue events.* A background apply process can dequeue events. You can also dequeue events explicitly with a user application.
- *Apply events at a database.* You can configure an apply process to apply all of the events in a queue or only the events that you specify. You can also configure an apply process to call your own PL/SQL subprograms to process events.

  The database where LCR events are applied and other types of events are processed is called the destination database. In some configurations, the source database and the destination database may be the same.

Streams Message Queuing

Streams lets user applications:

- Enqueue messages of different types
- Propagate messages are ready for consumption
- Dequeue messages at the destination database

Streams introduces a new type of queue that stages messages of type `SYS.AnyData`. Messages of almost any type can be wrapped in a `SYS.AnyData` wrapper and staged in `SYS.AnyData` queues. Streams interoperates with Advanced Queuing (AQ), which supports all the standard features of message queuing systems, including multiconsumer queues, publishing and subscribing, content-based routing, internet propagation, transformations, and gateways to other messaging subsystems.

> **See Also:** *Oracle Streams Concepts and Administration,* and its Appendix A, "XML Schema for LCRs".

XMLType Attributes in Object Types

You can create queues that use Oracle object types containing `XMLType` attributes. These queues can be used to transmit and store messages that are XML documents. Using `XMLType`, you can do the following:

- Store any type of message in a queue
- Store documents internally as `CLOB` values
- Store more than one type of payload in a queue
- Query `XMLType` columns using SQL/XML functions such as `XMLExists`
- Specify the operators in subscriber rules or dequeue selectors

Internet Data Access Presentation (iDAP)

You can access AQ over the Internet by using SOAP. Internet Data Access Presentation (iDAP) is the SOAP specification for AQ operations. iDAP defines XML message structure for the body of the SOAP request. An iDAP-structured message is transmitted over the Internet using transport protocols such as HTTP(S) and SMTP.

iDAP uses the `text/xml` content type to specify the body of the SOAP request. XML provides the presentation for iDAP request and response messages as follows:

- All request and response tags are scoped in the SOAP namespace.
- AQ operations are scoped in the iDAP namespace.
- The sender includes namespaces in iDAP elements and attributes in the SOAP body.
- The receiver processes iDAP messages that have correct namespaces. For the requests with incorrect namespaces, the receiver returns an invalid request error.
- The SOAP namespace has this value:
  `http://schemas.xmlsoap.org/soap/envelope/`
- The iDAP namespace has this value:
  `http://ns.oracle.com/AQ/schemas/access`

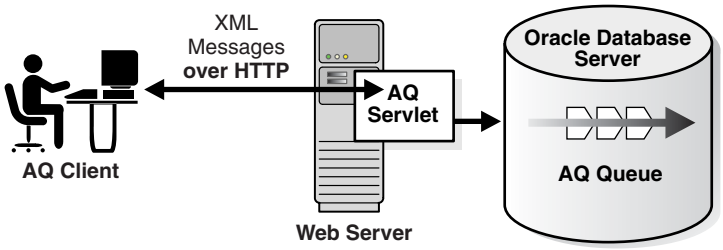
See Also: *Oracle Streams Advanced Queuing User's Guide*

iDAP Architecture

Figure 37–2 shows the following components needed to send HTTP(S) messages:

- A client program that sends XML messages, conforming to iDAP format, to the AQ Servlet. This can be any HTTP client, such as Web browsers.
- The Web server or `ServletRunner` which hosts the AQ servlet that can interpret the incoming XML messages, for example, Apache/Jserv or Tomcat.
- Oracle Server/Database. Oracle Streams AQ servlet connects to Oracle Database to perform operations on your queues.

Figure 37–2 iDAP Architecture for Performing AQ Operations using HTTP(S)



XMLType Queue Payloads

You can create queues with payloads that contain `XMLType` attributes. These can be used for transmitting and storing messages that contain XML documents. By defining Oracle objects with `XMLType` attributes, you can do the following:

- Store more than one type of XML document in the same queue. The documents are stored internally as `CLOB` instances.

- Selectively dequeue messages with `XMLType` attributes using SQL/XML functions such as `XMLExists` and `XMLQuery`.
- Define transformations to convert Oracle objects to `XMLType`.
- Define rule-based subscribers that query message content using SQL/XML functions such as `XMLExists` and `XMLQuery`.

In the BooksOnline application, assume that the Overseas Shipping site represents an order using `SYS.XMLType`. The Order Entry site represents an order as an Oracle object, `ORDER_TYP`.

Example 37–1 creates the queue table and queue for Overseas Shipping.

Example 37–1 Creating a Queue Table and Queue

```
BEGIN
   DBMS_AQADM.create_queue_table(
     queue_table        => 'OS_orders_pr_mqtab',
     comment            => 'Overseas Shipping MultiConsumer Orders queue table',
     multiple_consumers => TRUE,
     queue_payload_type => 'SYS.XMLtype',
     compatible         => '8.1');
END;
/

BEGIN
   DBMS_AQADM.create_queue(queue_name   => 'OS_bookedorders_que',
                           queue_table  => 'OS_orders_pr_mqtab');
END;
/
```

Because the representation of orders at the overseas shipping site is different from the representation of orders at the order-entry site, messages need to be transformed before sending them from the order-entry site to the overseas shipping site. Example 37–2 creates the transformation, and Example 37–3 applies it.

Example 37–2 Creating a Transformation to Convert Message Data to XML

```
CREATE OR REPLACE FUNCTION convert_to_order_xml(input_order ORDER_TYP)
  RETURN XMLType AS
    new_order XMLType;
BEGIN
  SELECT sys_xmlgen(input_order) INTO new_order FROM DUAL;
  RETURN new_order;
END convert_to_order_xml;
/

BEGIN
  SYS.DBMS_TRANSFORM.create_transformation(
    schema =>         'OE',
    name   =>         'OE2XML',
    from_schema =>    'OE',
    from_type =>      'ORDER_TYP',
    to_schema =>      'SYS',
    to_type =>        'XMLTYPE',
    transformation => 'convert_to_order_xml(source.user_data)');
END;
/
```

Example 37–3 Applying a Transformation before Sending Messages Overseas

```
-- Add a rule-based subscriber for overseas shipping to the booked-orders
-- queues with transformation.
DECLARE
   subscriber SYS.AQ$_AGENT;
BEGIN
   subscriber := SYS.AQ$_AGENT('Overseas_Shipping',
                               'OS.OS_bookedorders_que',
                               NULL);
   DBMS_AQADM.add_subscriber(
     queue_name     => 'OS_bookedorders_que',
     subscriber     => subscriber,
     rule           => 'XMLSerialize(CONTENT XMLQuery(''//orderregion''' ||
                       'PASSING tab.user_data RETURNING CONTENT)' ||
                       ' AS VARCHAR2(1000)) = ''INTERNATIONAL''',
     transformation => 'OE.OE2XML');
END;
/
```

For more information about defining transformations that convert the type used by the order entry application to the type used by Overseas Shipping, see *Oracle Streams Advanced Queuing User's Guide*.

Example 37–4 shows how an application that processes orders for customers in another country, in this case Canada, can dequeue messages.

Example 37–4 XMLType and AQ: Dequeuing Messages

```
-- Create procedure to enqueue into single-consumer queues.
CREATE OR REPLACE PROCEDURE get_canada_orders AS
   deq_msgid             RAW(16);
   dopt                  DBMS_AQ.dequeue_options_t;
   mprop                 DBMS_AQ.message_properties_t;
   deq_order_data        SYS.XMLType;
   deq_order_data_text   CLOB;
   no_messages           EXCEPTION;
   PRAGMA EXCEPTION_INIT (no_messages, -25228);
   new_orders            BOOLEAN := TRUE;
BEGIN
   dopt.wait := 1;
   -- Specify dequeue condition to select orders for Canada.
   dopt.deq_condition := 'XMLSerialize(CONTENT ' ||
                         'XMLQuery(''/ORDER_TYP/CUSTOMER/COUNTRY/text()''' ||
                         ' PASSING tab.user_data RETURNING CONTENT)' ||
                         ' AS VARCHAR2(1000))=''CANADA''';
   dopt.consumer_name := 'Overseas_Shipping';
   WHILE (new_orders) LOOP
     BEGIN
       DBMS_AQ.dequeue(queue_name         => 'OS.OS_bookedorders_que',
                       dequeue_options    => dopt,
                       message_properties => mprop,
                       payload            => deq_order_data,
                       msgid              => deq_msgid);
       COMMIT;
       SELECT XMLSerialize(DOCUMENT deq_order_data AS CLOB)
         INTO deq_order_data_text FROM DUAL;
       DBMS_OUTPUT.put_line('Order for Canada - Order: ' || deq_order_data_text);
     EXCEPTION
       WHEN no_messages THEN
         DBMS_OUTPUT.put_line (' ---- NO MORE ORDERS  ---- ');
```

```
        new_orders := FALSE;
    END;
  END LOOP;
END;
/
```

Guidelines for Using XML and Oracle Streams Advanced Queuing

This section describes guidelines for using XML and Oracle Streams Advanced Queuing.

Storing Oracle Streams AQ XML Messages with Many PDFs as One Record?

You can exchange XML documents between businesses using Oracle Streams Advanced Queuing, where each message received or sent includes an XML header, XML attachment (XML data stream), DTDs, and PDF files, and store the data in a database table, such as a `queuetable`. You can enqueue the messages into Oracle queue tables as one record or piece. Or you can enqueue the messages as multiple records, such as one record for XML data streams as `CLOB` type, one record for PDF files as `RAW` type, and so on. You can also then dequeue the messages.

You can achieve this in the following ways:

- By defining an object type with (`CLOB`, `RAW`,...) attributes, and storing it as a single message.
- By using the AQ message grouping feature and storing it in multiple messages. Here the message properties are associated with a group. To use the message grouping feature, all messages must be the same payload type.

To specify the payload, first create an object type, for example:

```
CREATE TYPE mypayload_type as OBJECT (xmlDataStream CLOB, dtd CLOB, pdf BLOB);
```

then store it as a single message.

Adding New Recipients After Messages Are Enqueued

You can use the queue table to support message assignments. For example, when other businesses send messages to a specific company, they do not know who should be assigned to process the messages, but they know the messages are for Human Resources (HR) for example. Hence all messages go to the HR supervisor. At this point, the message is enqueued in the queue table. The HR supervisor is the only recipient of this message, and the entire HR staff have been pre-defined as subscribers for this queue.

You cannot change the recipient list after the message is enqueued. If you do not specify a recipient list then subscribers can subscribe to the queue and dequeue the message. Here, new recipients must be subscribers to the queue. Otherwise, you must dequeue the message and enqueue it again with new recipients.

Enqueuing and Dequeuing XML Messages?

Oracle Streams AQ supports enqueuing and dequeuing objects. These objects can have an attribute of type `XMLType` that contains an XML document, in addition to metadata attributes. Refer to *Oracle Streams Advanced Queuing User's Guide* for specific details and more examples.

Parsing Messages with XML Content from Oracle Streams AQ Queues

You can parse messages with XML content from an Oracle Streams AQ queue and then update tables and fields in an ODS (Operational Data Store).

You can use Oracle XML Parser for Java and Java Stored Procedures together with Oracle Streams AQ to obtain metadata such as AQ enqueue or dequeue times and JMS header information, based on queries that target certain XML data. You can combine this with using Oracle Text XML search.

See Also: Chapter 12, "Full-Text Search Over XML Data".

Preventing the Listener from Stopping Until the XML Document Is Processed

When receiving XML messages from clients as messages you may need to process them as soon as they arrive. But each XML document might take several seconds to process. For PL/SQL, one procedure starts the listener, dequeues the message, and calls another procedure to process the XML document. The listener could be held up until the XML document is processed, and messages would accumulate in the queue.

After receiving a message, you can instead submit a job using PL/SQL package `DBMS_JOB`. The job is invoked asynchronously in a different database session.

You can register a PL/SQL callback, which is invoked asynchronously when a message shows up in a queue. PL/SQL callbacks are part of the Oracle Streams AQ notification framework.

Using HTTPS with AQ

You can use Oracle Streams AQ Internet access to send XML messages to suppliers using HTTPS and receive a response. Using XML, you can enqueue and dequeue messages over HTTP(S) securely and transactionally.

See Also: *Oracle Streams Advanced Queuing User's Guide*

Storing XML in Oracle Streams AQ Message Payloads

You can store XML data in Oracle Streams AQ message payloads natively other than having an ADT as the payload with `SYS.XMLType` as part of the ADT. You can create queues with payloads and attributes as `XMLType`.

Comparing iDAP and SOAP

iDAP is the SOAP specification for AQ operations. iDAP is the XML specification for Oracle Streams AQ operations. SOAP defines a generic mechanism to invoke a service. iDAP defines these mechanisms to perform AQ operations.

iDAP has the following key properties not defined by SOAP:

- Transactional behavior. You can perform AQ operations in a transactional manner. A transaction can span multiple iDAP requests.
- Security. iDAP operations can be carried out only by authorized and authenticated users.

Part VII

Appendixes

Part VII of this manual provides background material as a set of appendixes:

- Appendix A, "Oracle-Supplied XML Schemas and Examples"
- Appendix B, "Oracle XML DB Restrictions"

A

Oracle-Supplied XML Schemas and Examples

This appendix includes the definition and structure of `RESOURCE_VIEW` and `PATH_VIEW` and the Oracle XML DB-supplied XML schemas. It also includes a full listing of the purchase-order XML schemas used in various examples, and the C example for loading XML content into Oracle XML DB.

This appendix contains these topics:

- XDBResource.xsd: XML Schema for Oracle XML DB Resources
- XDBResConfig.xsd: XML Schema for Resource Configuration
- acl.xsd: XML Schema for ACLs
- xdbconfig.xsd: XML Schema for Configuring Oracle XML DB
- xdiff.xsd: XML Schema for Comparing Schemas for In-Place Evolution
- Purchase-Order XML Schemas
- XSL Style Sheet Example, PurchaseOrder.xsl
- Loading XML Data using C (OCI)
- Initializing and Terminating an XML Context (OCI)

XDBResource.xsd: XML Schema for Oracle XML DB Resources

Here is the complete listing for the Oracle XML DB supplied XML schema, `XDBResource.xsd`, which is used to represent Oracle XML DB resources.

XDBResource.xsd

```
<schema xdb:schemaURL="http://xmlns.oracle.com/xdb/XDBResource.xsd"
        targetNamespace="http://xmlns.oracle.com/xdb/XDBResource.xsd"
        version="1.0" xdb:numProps="72" elementFormDefault="qualified"
        xdb:flags="23" xdb:mapStringToNCHAR="false" xdb:mapUnboundedStringToLob="false"
        xdb:storeVarrayAsTable="false" xdb:schemaOwner="XDB">
  <simpleType name="OracleUserName">
    <restriction base="string">
      <minLength value="1" fixed="false"/>
      <maxLength value="4000" fixed="false"/>
    </restriction>
  </simpleType>
  <simpleType name="ResMetaStr">
    <restriction base="string">
      <minLength value="1" fixed="false"/>
```

```
      <maxLength value="128" fixed="false"/>
    </restriction>
  </simpleType>
  <simpleType name="SchElemType">
    <restriction base="string">
      <minLength value="1" fixed="false"/>
      <maxLength value="4000" fixed="false"/>
    </restriction>
  </simpleType>
  <simpleType name="GUID">
    <restriction base="hexBinary">
      <minLength value="8" fixed="false"/>
      <maxLength value="32" fixed="false"/>
    </restriction>
  </simpleType>
  <simpleType name="LocksRaw">
    <restriction base="hexBinary">
      <minLength value="0" fixed="false"/>
      <maxLength value="2000" fixed="false"/>
    </restriction>
  </simpleType>
  <simpleType name="lockModeType">
    <restriction base="string">
      <enumeration value="exclusive" fixed="false"/>
      <enumeration value="shared" fixed="false"/>
    </restriction>
  </simpleType>
  <simpleType name="lockTypeType">
    <restriction base="string">
      <enumeration value="read-write" fixed="false"/>
      <enumeration value="write" fixed="false"/>
      <enumeration value="read" fixed="false"/>
    </restriction>
  </simpleType>
  <simpleType name="lockDepthType">
    <restriction base="string">
      <enumeration value="0" fixed="false"/>
      <enumeration value="infinity" fixed="false"/>
    </restriction>
  </simpleType>
  <complexType name="lockType" abstract="false" mixed="false">
    <sequence minOccurs="1" maxOccurs="1">
      <element xdb:propNumber="768" name="LockOwner" type="string" xdb:memType="1"
        xdb:system="false" xdb:mutable="true" xdb:JavaType="String" xdb:global="false"
        nillable="false" abstract="false" xdb:SQLInline="true" xdb:JavaInline="false"
        xdb:MemInline="true" xdb:maintainDOM="false" xdb:defaultTableSchema="XDB"
        xdb:numCols="1" minOccurs="1" maxOccurs="1"/>
      <element xdb:propNumber="769" name="Mode" type="xdb:lockModeType" xdb:memType="1"
        xdb:system="false" xdb:mutable="true" xdb:JavaType="String" xdb:global="false"
        nillable="false" abstract="false" xdb:SQLInline="true" xdb:JavaInline="false"
        xdb:MemInline="true" xdb:maintainDOM="false" xdb:defaultTableSchema="XDB"
        xdb:numCols="1" minOccurs="1" maxOccurs="1"/>
      <element xdb:propNumber="770" name="Type" type="xdb:lockTypeType" xdb:memType="1"
        xdb:system="false" xdb:mutable="true" xdb:JavaType="String" xdb:global="false"
        nillable="false" abstract="false" xdb:SQLInline="true" xdb:JavaInline="false"
        xdb:MemInline="true" xdb:maintainDOM="false" xdb:defaultTableSchema="XDB"
        xdb:numCols="1" minOccurs="1" maxOccurs="1"/>
      <element xdb:propNumber="771" name="Depth" type="xdb:lockDepthType" xdb:memType="1"
        xdb:system="false" xdb:mutable="true" xdb:JavaType="String" xdb:global="false"
        nillable="false" abstract="false" xdb:SQLInline="true" xdb:JavaInline="false"
```

```
      xdb:MemInline="true" xdb:maintainDOM="false" xdb:defaultTableSchema="XDB"
      xdb:numCols="1" minOccurs="1" maxOccurs="1"/>
    <element xdb:propNumber="772" name="Expiry" type="dateTime" xdb:memType="180"
      xdb:system="false" xdb:mutable="true" xdb:JavaType="TimeStamp" xdb:global="false"
      nillable="false" abstract="false" xdb:SQLInline="true" xdb:JavaInline="false"
      xdb:MemInline="true" xdb:maintainDOM="false" xdb:defaultTableSchema="XDB"
      xdb:numCols="1" minOccurs="0" maxOccurs="1"/>
    <choice minOccurs="0" maxOccurs="unbounded">
      <element xdb:propNumber="773" name="Token" type="string" xdb:memType="1"
      xdb:system="false" xdb:mutable="true" xdb:JavaType="String" xdb:global="false"
      nillable="false" abstract="false" xdb:SQLInline="true" xdb:JavaInline="false"
      xdb:MemInline="true" xdb:maintainDOM="false" xdb:defaultTableSchema="XDB"
      xdb:numCols="1" minOccurs="0" maxOccurs="1"/>
      <element xdb:propNumber="774" name="NodeId" type="string" xdb:memType="1"
      xdb:system="false" xdb:mutable="true" xdb:JavaType="String" xdb:global="false"
      nillable="false" abstract="false" xdb:SQLInline="true" xdb:JavaInline="false"
      xdb:MemInline="true" xdb:maintainDOM="false" xdb:defaultTableSchema="XDB"
      xdb:numCols="1" minOccurs="0" maxOccurs="1"/>
    </choice>
  </sequence>
</complexType>
<complexType name="locksType" abstract="false" mixed="false">
  <sequence minOccurs="1" maxOccurs="1">
    <element xdb:propNumber="767" name="Lock" type="xdb:lockType" xdb:memType="258"
      xdb:system="false" xdb:mutable="true" xdb:JavaType="XMLType" xdb:global="false"
      nillable="false" abstract="false" xdb:SQLInline="true" xdb:JavaInline="false"
      xdb:MemInline="true" xdb:maintainDOM="false" xdb:defaultTableSchema="XDB"
      minOccurs="0" maxOccurs="2147483647"/>
  </sequence>
</complexType>
<complexType name="ResContentsType" abstract="false" mixed="false">
  <sequence minOccurs="1" maxOccurs="1">
    <any xdb:propNumber="736" name="ContentsAny" xdb:memType="258" xdb:system="false"
      xdb:mutable="false" xdb:JavaType="XMLType" minOccurs="0" maxOccurs="1"/>
  </sequence>
</complexType>
<complexType name="ResAclType" abstract="false" mixed="false">
  <sequence minOccurs="1" maxOccurs="1">
    <any xdb:propNumber="737" name="ACLAny" xdb:memType="258" xdb:system="false"
      xdb:mutable="false" xdb:JavaType="XMLType" minOccurs="0" maxOccurs="1"/>
  </sequence>
</complexType>
<complexType name="AttrCopyType" abstract="false" mixed="false">
  <sequence minOccurs="1" maxOccurs="1">
    <any xdb:propNumber="748" name="AttrCopyAny" xdb:memType="258" xdb:system="false"
      xdb:mutable="false" xdb:JavaType="XMLType" minOccurs="0" maxOccurs="65535"/>
  </sequence>
</complexType>
<complexType name="RCListType" abstract="false" mixed="false">
  <sequence minOccurs="1" maxOccurs="1">
    <element xdb:propNumber="755" name="OID" type="hexBinary" xdb:memByteLength="22"
      xdb:memType="23" xdb:system="false" xdb:mutable="false" xdb:SQLName="OID"
      xdb:SQLType="RAW" xdb:JavaType="byteArray" xdb:global="false"
      xdb:SQLCollType="XDB$OID_LIST_T" xdb:SQLCollSchema="XDB" xdb:hidden="false"
      nillable="false" abstract="false" xdb:SQLInline="true" xdb:JavaInline="false"
      xdb:MemInline="true" xdb:maintainDOM="false" xdb:defaultTableSchema="XDB"
      xdb:numCols="1" minOccurs="1" maxOccurs="65535"/>
  </sequence>
</complexType>
<complexType name="ResourceType" abstract="false" mixed="false">
```

```
<sequence minOccurs="1" maxOccurs="1">
  <element xdb:propNumber="709" name="CreationDate" type="dateTime" xdb:memType="180"
    xdb:system="false" xdb:mutable="false" xdb:SQLName="CREATIONDATE"
    xdb:SQLType="TIMESTAMP" xdb:JavaType="TimeStamp" xdb:global="false"
    nillable="false" abstract="false" xdb:SQLInline="true" xdb:JavaInline="false"
    xdb:MemInline="true" xdb:maintainDOM="false" xdb:defaultTableSchema="XDB"
    xdb:numCols="1" minOccurs="1" maxOccurs="1"/>
  <element xdb:propNumber="710" name="ModificationDate" type="dateTime"
    xdb:memType="180" xdb:system="false" xdb:mutable="false"
    xdb:SQLName="MODIFICATIONDATE" xdb:SQLType="TIMESTAMP" xdb:JavaType="TimeStamp"
    xdb:global="false" nillable="false" abstract="false" xdb:SQLInline="true"
    xdb:JavaInline="false" xdb:MemInline="true" xdb:maintainDOM="false"
    xdb:defaultTableSchema="XDB" xdb:numCols="1" minOccurs="1" maxOccurs="1"/>
  <element xdb:propNumber="711" name="Author" type="xdb:ResMetaStr" xdb:memType="1"
    xdb:system="false" xdb:mutable="false" xdb:SQLName="AUTHOR" xdb:SQLType="VARCHAR2"
    xdb:JavaType="String" xdb:global="false" nillable="false" abstract="false"
    xdb:SQLInline="true" xdb:JavaInline="false" xdb:MemInline="false"
    xdb:maintainDOM="false" xdb:defaultTableSchema="XDB" xdb:numCols="1" minOccurs="0"
    maxOccurs="1"/>
  <element xdb:propNumber="712" name="DisplayName" type="xdb:ResMetaStr"
    xdb:memType="1" xdb:system="false" xdb:mutable="false" xdb:SQLName="DISPNAME"
    xdb:SQLType="VARCHAR2" xdb:JavaType="String" xdb:global="false" nillable="false"
    abstract="false" xdb:SQLInline="true" xdb:JavaInline="false" xdb:MemInline="false"
    xdb:maintainDOM="false" xdb:defaultTableSchema="XDB" xdb:numCols="1" minOccurs="1"
    maxOccurs="1"/>
  <element xdb:propNumber="713" name="Comment" type="xdb:ResMetaStr" xdb:memType="1"
    xdb:system="false" xdb:mutable="false" xdb:SQLName="RESCOMMENT"
    xdb:SQLType="VARCHAR2" xdb:JavaType="String" xdb:global="false" nillable="false"
    abstract="false" xdb:SQLInline="true" xdb:JavaInline="false" xdb:MemInline="false"
    xdb:maintainDOM="false" xdb:defaultTableSchema="XDB" xdb:numCols="1" minOccurs="1"
    maxOccurs="1"/>
  <element xdb:propNumber="714" name="Language" type="xdb:ResMetaStr" xdb:memType="1"
    xdb:system="false" xdb:mutable="false" xdb:SQLName="LANGUAGE"
    xdb:SQLType="VARCHAR2" xdb:JavaType="String" default="en" xdb:global="false"
    nillable="false" abstract="false" xdb:SQLInline="true" xdb:JavaInline="false"
    xdb:MemInline="false" xdb:maintainDOM="false" xdb:defaultTableSchema="XDB"
    xdb:numCols="1" minOccurs="1" maxOccurs="1"/>
  <element xdb:propNumber="715" name="CharacterSet" type="xdb:ResMetaStr"
    xdb:memType="1" xdb:system="false" xdb:mutable="false" xdb:SQLName="CHARSET"
    xdb:SQLType="VARCHAR2" xdb:JavaType="String" xdb:global="false" nillable="false"
    abstract="false" xdb:SQLInline="true" xdb:JavaInline="false" xdb:MemInline="false"
    xdb:maintainDOM="false" xdb:defaultTableSchema="XDB" xdb:numCols="1" minOccurs="0"
    maxOccurs="1"/>
  <element xdb:propNumber="716" name="ContentType" type="xdb:ResMetaStr"
    xdb:memType="1" xdb:system="false" xdb:mutable="false" xdb:SQLName="CONTYPE"
    xdb:SQLType="VARCHAR2" xdb:JavaType="String" xdb:global="false" nillable="false"
    abstract="false" xdb:SQLInline="true" xdb:JavaInline="false" xdb:MemInline="false"
    xdb:maintainDOM="false" xdb:defaultTableSchema="XDB" xdb:numCols="1" minOccurs="0"
    maxOccurs="1"/>
  <element xdb:propNumber="717" name="RefCount" type="nonNegativeInteger"
    xdb:memByteLength="4" xdb:memType="68" xdb:system="false" xdb:mutable="true"
    xdb:SQLName="REFCOUNT" xdb:SQLType="RAW" xdb:JavaType="long" xdb:global="false"
    nillable="false" abstract="false" xdb:SQLInline="true" xdb:JavaInline="false"
    xdb:MemInline="true" xdb:maintainDOM="false" xdb:defaultTableSchema="XDB"
    xdb:numCols="1" minOccurs="1" maxOccurs="1"/>
  <element xdb:propNumber="718" name="LockBuf" type="xdb:LocksRaw" xdb:memType="23"
    xdb:system="false" xdb:mutable="true" xdb:SQLName="LOCKS" xdb:SQLType="RAW"
    xdb:JavaType="byteArray" xdb:global="false" nillable="false" abstract="false"
    xdb:SQLInline="true" xdb:JavaInline="false" xdb:MemInline="false"
    xdb:maintainDOM="false" xdb:defaultTableSchema="XDB" xdb:numCols="1" minOccurs="0"
```

```
  maxOccurs="1"/>
<element xdb:propNumber="732" name="ACL" type="xdb:ResAclType" xdb:memType="258"
  xdb:system="false" xdb:mutable="false" xdb:JavaType="XMLType" xdb:global="false"
  xdb:hidden="false" xdb:transient="generated" xdb:baseProp="false" nillable="false"
  abstract="false" xdb:SQLInline="true" xdb:JavaInline="false" xdb:MemInline="false"
  xdb:maintainDOM="false" xdb:defaultTableSchema="XDB"
  xdb:JavaClassname="oracle.xdb.ResAclTypeBean"
  xdb:beanClassname="oracle.xdb.ResAclTypeBean" xdb:numCols="0" minOccurs="0"
  maxOccurs="1"/>
<element xdb:propNumber="719" name="ACLOID" type="hexBinary" xdb:memType="23"
  xdb:system="false" xdb:mutable="false" xdb:SQLName="ACLOID" xdb:SQLType="RAW"
  xdb:JavaType="byteArray" xdb:global="false" xdb:hidden="true" xdb:baseProp="true"
  nillable="false" abstract="false" xdb:SQLInline="true" xdb:JavaInline="false"
  xdb:MemInline="true" xdb:maintainDOM="false" xdb:defaultTableSchema="XDB"
  xdb:numCols="1" minOccurs="1" maxOccurs="1"/>
<element xdb:propNumber="720" name="Owner" type="xdb:OracleUserName" xdb:memType="1"
  xdb:system="false" xdb:mutable="false" xdb:JavaType="String" xdb:global="false"
  xdb:hidden="false" xdb:transient="generated" xdb:baseProp="false" nillable="false"
  abstract="false" xdb:SQLInline="true" xdb:JavaInline="false" xdb:MemInline="false"
  xdb:maintainDOM="false" xdb:defaultTableSchema="XDB" xdb:numCols="0" minOccurs="0"
  maxOccurs="1"/>
<element xdb:propNumber="721" name="OwnerID" type="xdb:GUID" xdb:memType="23"
  xdb:system="false" xdb:mutable="false" xdb:SQLName="OWNERID" xdb:SQLType="RAW"
  xdb:JavaType="byteArray" xdb:global="false" xdb:hidden="true" xdb:baseProp="true"
  nillable="false" abstract="false" xdb:SQLInline="true" xdb:JavaInline="false"
  xdb:MemInline="true" xdb:maintainDOM="false" xdb:defaultTableSchema="XDB"
  xdb:numCols="1" minOccurs="1" maxOccurs="1"/>
<element xdb:propNumber="722" name="Creator" type="xdb:OracleUserName"
  xdb:memType="1" xdb:system="false" xdb:mutable="false" xdb:JavaType="String"
  xdb:global="false" xdb:hidden="false" xdb:transient="generated"
  xdb:baseProp="false" nillable="false" abstract="false" xdb:SQLInline="true"
  xdb:JavaInline="false" xdb:MemInline="false" xdb:maintainDOM="false"
  xdb:defaultTableSchema="XDB" xdb:numCols="0" minOccurs="0" maxOccurs="1"/>
<element xdb:propNumber="723" name="CreatorID" type="xdb:GUID" xdb:memType="23"
  xdb:system="false" xdb:mutable="false" xdb:SQLName="CREATORID" xdb:SQLType="RAW"
  xdb:JavaType="byteArray" xdb:global="false" xdb:hidden="true" xdb:baseProp="true"
  nillable="false" abstract="false" xdb:SQLInline="true" xdb:JavaInline="false"
  xdb:MemInline="true" xdb:maintainDOM="false" xdb:defaultTableSchema="XDB"
  xdb:numCols="1" minOccurs="1" maxOccurs="1"/>
<element xdb:propNumber="724" name="LastModifier" type="xdb:OracleUserName"
  xdb:memType="1" xdb:system="false" xdb:mutable="false" xdb:JavaType="String"
  xdb:global="false" xdb:hidden="false" xdb:transient="generated"
  xdb:baseProp="false" nillable="false" abstract="false" xdb:SQLInline="true"
  xdb:JavaInline="false" xdb:MemInline="false" xdb:maintainDOM="false"
  xdb:defaultTableSchema="XDB" xdb:numCols="0" minOccurs="0" maxOccurs="1"/>
<element xdb:propNumber="725" name="LastModifierID" type="xdb:GUID" xdb:memType="23"
  xdb:system="false" xdb:mutable="false" xdb:SQLName="LASTMODIFIERID"
  xdb:SQLType="RAW" xdb:JavaType="byteArray" xdb:global="false" xdb:hidden="true"
  xdb:baseProp="true" nillable="false" abstract="false" xdb:SQLInline="true"
  xdb:JavaInline="false" xdb:MemInline="true" xdb:maintainDOM="false"
  xdb:defaultTableSchema="XDB" xdb:numCols="1" minOccurs="1" maxOccurs="1"/>
<element xdb:propNumber="726" name="SchemaElement" type="xdb:SchElemType"
  xdb:memType="1" xdb:system="false" xdb:mutable="false" xdb:JavaType="String"
  xdb:global="false" xdb:hidden="false" xdb:transient="generated"
  xdb:baseProp="false" nillable="false" abstract="false" xdb:SQLInline="true"
  xdb:JavaInline="false" xdb:MemInline="false" xdb:maintainDOM="false"
  xdb:defaultTableSchema="XDB" xdb:numCols="0" minOccurs="0" maxOccurs="1"/>
<element xdb:propNumber="727" name="ElNum" type="nonNegativeInteger"
  xdb:memByteLength="4" xdb:memType="3" xdb:system="false" xdb:mutable="false"
  xdb:SQLName="ELNUM" xdb:SQLType="INTEGER" xdb:JavaType="long" xdb:global="false"
```

```
    xdb:hidden="true" xdb:baseProp="true" nillable="false" abstract="false"
    xdb:SQLInline="true" xdb:JavaInline="false" xdb:MemInline="true"
    xdb:maintainDOM="false" xdb:defaultTableSchema="XDB" xdb:numCols="1" minOccurs="1"
    maxOccurs="1"/>
  <element xdb:propNumber="728" name="SchOID" type="hexBinary" xdb:memType="23"
    xdb:system="false" xdb:mutable="false" xdb:SQLName="SCHOID" xdb:SQLType="RAW"
    xdb:JavaType="byteArray" xdb:global="false" xdb:hidden="true" xdb:baseProp="true"
    nillable="false" abstract="false" xdb:SQLInline="true" xdb:JavaInline="false"
    xdb:MemInline="true" xdb:maintainDOM="false" xdb:defaultTableSchema="XDB"
    xdb:numCols="1" minOccurs="1" maxOccurs="1"/>
  <element xdb:propNumber="733" name="Contents" type="xdb:ResContentsType"
    xdb:memType="258" xdb:system="false" xdb:mutable="false" xdb:JavaType="XMLType"
    xdb:global="false" xdb:hidden="false" xdb:transient="manifested"
    xdb:baseProp="false" nillable="false" abstract="false" xdb:SQLInline="true"
    xdb:JavaInline="false" xdb:MemInline="false" xdb:maintainDOM="false"
    xdb:defaultTableSchema="XDB" xdb:JavaClassname="oracle.xdb.ResContentsTypeBean"
    xdb:beanClassname="oracle.xdb.ResContentsTypeBean" xdb:numCols="0" minOccurs="0"
    maxOccurs="1"/>
  <element xdb:propNumber="729" name="XMLRef" type="REF" xdb:memType="110"
    xdb:system="false" xdb:mutable="true" xdb:SQLName="XMLREF" xdb:SQLType="REF"
    xdb:JavaType="Reference" xdb:global="false" xdb:hidden="true" xdb:baseProp="false"
    nillable="false" abstract="false" xdb:SQLInline="true" xdb:JavaInline="false"
    xdb:MemInline="false" xdb:maintainDOM="false" xdb:defaultTableSchema="XDB"
    xdb:numCols="1" minOccurs="0" maxOccurs="1"/>
  <element xdb:propNumber="730" name="XMLLob" type="hexBinary" xdb:memType="113"
    xdb:system="false" xdb:mutable="true" xdb:SQLName="XMLLOB" xdb:SQLType="BLOB"
    xdb:JavaType="String" xdb:global="false" xdb:hidden="true" xdb:baseProp="false"
    nillable="false" abstract="false" xdb:SQLInline="true" xdb:JavaInline="false"
    xdb:MemInline="false" xdb:maintainDOM="false" xdb:defaultTableSchema="XDB"
    xdb:numCols="1" minOccurs="0" maxOccurs="1"/>
  <element xdb:propNumber="731" name="Flags" type="nonNegativeInteger"
    xdb:memByteLength="4" xdb:memType="3" xdb:system="false" xdb:mutable="true"
    xdb:SQLName="FLAGS" xdb:SQLType="RAW" xdb:JavaType="long" xdb:global="false"
    xdb:hidden="true" xdb:baseProp="true" nillable="false" abstract="false"
    xdb:SQLInline="true" xdb:JavaInline="false" xdb:MemInline="true"
    xdb:maintainDOM="false" xdb:defaultTableSchema="XDB" xdb:numCols="1" minOccurs="1"
    maxOccurs="1"/>
  <element xdb:propNumber="740" name="VCRUID" type="xdb:GUID" xdb:memType="23"
    xdb:system="false" xdb:mutable="false" xdb:SQLName="VCRUID" xdb:SQLType="RAW"
    xdb:JavaType="byteArray" xdb:global="false" nillable="false" abstract="false"
    xdb:SQLInline="true" xdb:JavaInline="false" xdb:MemInline="false"
    xdb:maintainDOM="false" xdb:defaultTableSchema="XDB" xdb:numCols="1" minOccurs="1"
    maxOccurs="1"/>
  <element xdb:propNumber="741" name="Parents" type="hexBinary" xdb:memType="23"
    xdb:system="false" xdb:mutable="false" xdb:SQLName="PARENTS" xdb:SQLType="RAW"
    xdb:JavaType="Reference" xdb:global="false"
    xdb:SQLCollType="XDB$PREDECESSOR_LIST_T" xdb:SQLCollSchema="XDB" nillable="false"
    abstract="false" xdb:SQLInline="true" xdb:JavaInline="false" xdb:MemInline="false"
    xdb:maintainDOM="false" xdb:defaultTableSchema="XDB" xdb:numCols="0" minOccurs="0"
    maxOccurs="1000"/>
  <element xdb:propNumber="745" name="SBResExtra" type="REF" xdb:memType="110"
    xdb:system="false" xdb:mutable="true" xdb:SQLName="SBRESEXTRA" xdb:SQLType="REF"
    xdb:JavaType="Reference" xdb:global="false"
    xdb:SQLCollType="XDB$XMLTYPE_REF_LIST_T" xdb:SQLCollSchema="XDB" xdb:hidden="true"
    xdb:baseProp="false" nillable="false" abstract="false" xdb:SQLInline="true"
    xdb:JavaInline="false" xdb:MemInline="false" xdb:maintainDOM="false"
    xdb:defaultTableSchema="XDB" xdb:numCols="1" minOccurs="0"
    maxOccurs="2147483647"/>
  <element xdb:propNumber="746" name="Snapshot" type="hexBinary" xdb:memType="23"
    xdb:system="false" xdb:mutable="true" xdb:SQLName="SNAPSHOT" xdb:SQLType="RAW"
```

```
  xdb:JavaType="byteArray" xdb:global="false" xdb:hidden="true" xdb:baseProp="true"
  nillable="false" abstract="false" xdb:SQLInline="true" xdb:JavaInline="false"
  xdb:MemInline="true" xdb:maintainDOM="false" xdb:defaultTableSchema="XDB"
  xdb:numCols="1" minOccurs="1" maxOccurs="1"/>
<element xdb:propNumber="747" name="AttrCopy" type="xdb:AttrCopyType"
  xdb:memType="258" xdb:system="false" xdb:mutable="true" xdb:SQLName="ATTRCOPY"
  xdb:SQLType="BLOB" xdb:JavaType="XMLType" xdb:global="false" xdb:hidden="true"
  xdb:baseProp="true" nillable="false" abstract="false" xdb:SQLInline="true"
  xdb:JavaInline="false" xdb:MemInline="false" xdb:maintainDOM="false"
  xdb:defaultTableSchema="XDB" xdb:numCols="0" minOccurs="0" maxOccurs="1"/>
<element xdb:propNumber="749" name="CtsCopy" type="hexBinary" xdb:memType="113"
  xdb:system="false" xdb:mutable="true" xdb:SQLName="CTSCOPY" xdb:SQLType="BLOB"
  xdb:JavaType="String" xdb:global="false" xdb:hidden="true" xdb:baseProp="false"
  nillable="false" abstract="false" xdb:SQLInline="true" xdb:JavaInline="false"
  xdb:MemInline="false" xdb:maintainDOM="false" xdb:defaultTableSchema="XDB"
  xdb:numCols="1" minOccurs="0" maxOccurs="1"/>
<element xdb:propNumber="750" name="NodeNum" type="hexBinary" xdb:memType="23"
  xdb:system="false" xdb:mutable="true" xdb:SQLName="NODENUM" xdb:SQLType="RAW"
  xdb:JavaType="byteArray" xdb:global="false" xdb:hidden="true" xdb:baseProp="true"
  nillable="false" abstract="false" xdb:SQLInline="true" xdb:JavaInline="false"
  xdb:MemInline="true" xdb:maintainDOM="false" xdb:defaultTableSchema="XDB"
  xdb:numCols="1" minOccurs="1" maxOccurs="1"/>
<element xdb:propNumber="751" name="ContentSize" type="integer"
  xdb:memByteLength="8" xdb:memType="3" xdb:system="false" xdb:mutable="false"
  xdb:JavaType="long" xdb:global="false" xdb:hidden="true" xdb:transient="generated"
  xdb:baseProp="false" nillable="false" abstract="false" xdb:SQLInline="true"
  xdb:JavaInline="false" xdb:MemInline="true" xdb:maintainDOM="false"
  xdb:defaultTableSchema="XDB" xdb:numCols="1" minOccurs="0" maxOccurs="1"/>
<element xdb:propNumber="752" name="SizeOnDisk" type="nonNegativeInteger"
  xdb:memByteLength="8" xdb:memType="3" xdb:system="false" xdb:mutable="false"
  xdb:SQLName="SIZEONDISK" xdb:SQLType="INTEGER" xdb:JavaType="long"
  xdb:global="false" xdb:hidden="true" xdb:baseProp="true" nillable="false"
  abstract="false" xdb:SQLInline="true" xdb:JavaInline="false" xdb:MemInline="true"
  xdb:maintainDOM="false" xdb:defaultTableSchema="XDB" xdb:numCols="1" minOccurs="0"
  maxOccurs="1"/>
<element xdb:propNumber="754" name="RCList" type="xdb:RCListType" xdb:memType="258"
  xdb:system="false" xdb:mutable="false" xdb:SQLName="RCLIST"
  xdb:SQLType="XDB$RCLIST_T" xdb:SQLSchema="XDB" xdb:JavaType="XMLType"
  xdb:global="true" xdb:hidden="true" xdb:baseProp="true" nillable="false"
  abstract="false" xdb:SQLInline="true" xdb:JavaInline="false" xdb:MemInline="false"
  xdb:maintainDOM="false" xdb:defaultTable="00" xdb:defaultTableSchema="XDB"
  xdb:JavaClassname="oracle.xdb.RCListBean"
  xdb:beanClassname="oracle.xdb.RCListBean" xdb:numCols="1" minOccurs="0"
  maxOccurs="1"/>
<element xdb:propNumber="762" name="Branch" type="string" xdb:memType="1"
  xdb:system="false" xdb:mutable="false" xdb:SQLName="BRANCH" xdb:SQLType="VARCHAR2"
  xdb:JavaType="String" xdb:global="false" xdb:hidden="false"
  xdb:transient="generated" xdb:baseProp="false" nillable="false" abstract="false"
  xdb:SQLInline="true" xdb:JavaInline="false" xdb:MemInline="false"
  xdb:maintainDOM="false" xdb:defaultTableSchema="XDB" xdb:numCols="1" minOccurs="0"
  maxOccurs="1"/>
<element xdb:propNumber="763" name="CheckedOutBy" type="xdb:OracleUserName"
  xdb:memType="1" xdb:system="false" xdb:mutable="false" xdb:JavaType="String"
  xdb:global="false" xdb:hidden="false" xdb:transient="generated"
  xdb:baseProp="false" nillable="false" abstract="false" xdb:SQLInline="true"
  xdb:JavaInline="false" xdb:MemInline="false" xdb:maintainDOM="false"
  xdb:defaultTableSchema="XDB" xdb:numCols="0" minOccurs="0" maxOccurs="1"/>
<element xdb:propNumber="764" name="CheckedOutByID" type="xdb:GUID" xdb:memType="23"
  xdb:system="false" xdb:mutable="false" xdb:SQLName="CHECKEDOUTBYID"
  xdb:SQLType="RAW" xdb:JavaType="byteArray" xdb:global="false" xdb:hidden="true"
```

```
      xdb:baseProp="true" nillable="false" abstract="false" xdb:SQLInline="true"
      xdb:JavaInline="false" xdb:MemInline="true" xdb:maintainDOM="false"
      xdb:defaultTableSchema="XDB" xdb:numCols="1" minOccurs="0" maxOccurs="1"/>
    <element xdb:propNumber="765" name="BaseVersion" type="hexBinary" xdb:memType="23"
      xdb:system="false" xdb:mutable="false" xdb:SQLName="BASEVERSION" xdb:SQLType="RAW"
      xdb:JavaType="byteArray" xdb:global="false" xdb:hidden="false"
      xdb:baseProp="false" nillable="false" abstract="false" xdb:SQLInline="true"
      xdb:JavaInline="false" xdb:MemInline="true" xdb:maintainDOM="false"
      xdb:defaultTableSchema="XDB" xdb:numCols="1" minOccurs="0" maxOccurs="1"/>
    <element xdb:propNumber="766" name="Locks" type="xdb:locksType" xdb:memType="258"
      xdb:system="false" xdb:mutable="true" xdb:JavaType="XMLType" xdb:global="false"
      xdb:hidden="true" xdb:transient="generated" nillable="false" abstract="false"
      xdb:SQLInline="true" xdb:JavaInline="false" xdb:MemInline="false"
      xdb:maintainDOM="false" xdb:defaultTableSchema="XDB" xdb:numCols="1" minOccurs="0"
      maxOccurs="1"/>
    <any xdb:propNumber="735" name="ResExtra" xdb:memType="258" xdb:system="false"
      xdb:mutable="false" xdb:SQLName="RESEXTRA" xdb:SQLType="CLOB"
      xdb:JavaType="XMLType" namespace="##other" minOccurs="0" maxOccurs="65535"/>
  </sequence>
  <attribute xdb:propNumber="705" name="Hidden" type="boolean" xdb:memByteLength="1"
      xdb:memType="252" xdb:system="false" xdb:mutable="false" xdb:JavaType="boolean"
      default="false" xdb:hidden="false" xdb:transient="generated"
      xdb:baseProp="false"/>
  <attribute xdb:propNumber="706" name="Invalid" type="boolean" xdb:memByteLength="1"
      xdb:memType="252" xdb:system="false" xdb:mutable="false" xdb:JavaType="boolean"
      default="false" xdb:hidden="false" xdb:transient="generated"
      xdb:baseProp="false"/>
  <attribute xdb:propNumber="707" name="VersionID" type="integer" xdb:memByteLength="4"
      xdb:memType="3" xdb:system="false" xdb:mutable="false" xdb:SQLName="VERSIONID"
      xdb:SQLType="INTEGER" xdb:JavaType="long"/>
  <attribute xdb:propNumber="708" name="ActivityID" type="integer" xdb:memByteLength="4"
      xdb:memType="3" xdb:system="false" xdb:mutable="false" xdb:SQLName="ACTIVITYID"
      xdb:SQLType="INTEGER" xdb:JavaType="long"/>
  <attribute xdb:propNumber="738" name="Container" type="boolean" xdb:memByteLength="1"
      xdb:memType="252" xdb:system="false" xdb:mutable="false" xdb:JavaType="boolean"
      default="false" xdb:hidden="false" xdb:transient="generated"
      xdb:baseProp="false"/>
  <attribute xdb:propNumber="739" name="CustomRslv" type="boolean" xdb:memByteLength="1"
      xdb:memType="252" xdb:system="false" xdb:mutable="false" xdb:JavaType="boolean"
      default="false" xdb:hidden="false" xdb:transient="generated"
      xdb:baseProp="false"/>
  <attribute xdb:propNumber="742" name="VersionHistory" type="boolean"
      xdb:memByteLength="1" xdb:memType="252" xdb:system="false" xdb:mutable="false"
      xdb:JavaType="boolean" default="false" xdb:hidden="false"
      xdb:transient="generated" xdb:baseProp="false"/>
  <attribute xdb:propNumber="743" name="StickyRef" type="boolean" xdb:memByteLength="1"
      xdb:memType="252" xdb:system="false" xdb:mutable="false" xdb:JavaType="boolean"
      default="false" xdb:hidden="false" xdb:transient="generated"
      xdb:baseProp="false"/>
  <attribute xdb:propNumber="744" name="HierSchmResource" type="boolean"
      xdb:memByteLength="1" xdb:memType="252" xdb:system="false" xdb:mutable="false"
      xdb:JavaType="boolean" default="false" xdb:hidden="true" xdb:transient="generated"
      xdb:baseProp="false"/>
  <attribute xdb:propNumber="753" name="SizeAccurate" type="boolean"
      xdb:memByteLength="1" xdb:memType="252" xdb:system="false" xdb:mutable="false"
      xdb:JavaType="boolean" default="false" xdb:hidden="true" xdb:transient="generated"
      xdb:baseProp="false"/>
  <attribute xdb:propNumber="756" name="IsVersionable" type="boolean"
      xdb:memByteLength="1" xdb:memType="252" xdb:system="false" xdb:mutable="false"
      xdb:JavaType="boolean" default="false" xdb:hidden="true" xdb:transient="generated"
```

```
        xdb:baseProp="false"/>
    <attribute xdb:propNumber="757" name="IsCheckedOut" type="boolean"
        xdb:memByteLength="1" xdb:memType="252" xdb:system="false" xdb:mutable="false"
        xdb:JavaType="boolean" default="false" xdb:hidden="true" xdb:transient="generated"
        xdb:baseProp="false"/>
    <attribute xdb:propNumber="758" name="IsVersion" type="boolean" xdb:memByteLength="1"
        xdb:memType="252" xdb:system="false" xdb:mutable="false" xdb:JavaType="boolean"
        default="false" xdb:hidden="true" xdb:transient="generated" xdb:baseProp="false"/>
    <attribute xdb:propNumber="759" name="IsVCR" type="boolean" xdb:memByteLength="1"
        xdb:memType="252" xdb:system="false" xdb:mutable="false" xdb:JavaType="boolean"
        default="false" xdb:hidden="true" xdb:transient="generated" xdb:baseProp="false"/>
    <attribute xdb:propNumber="760" name="IsVersionHistory" type="boolean"
        xdb:memByteLength="1" xdb:memType="252" xdb:system="false" xdb:mutable="false"
        xdb:JavaType="boolean" default="false" xdb:hidden="true" xdb:transient="generated"
        xdb:baseProp="false"/>
    <attribute xdb:propNumber="761" name="IsWorkspace" type="boolean"
        xdb:memByteLength="1" xdb:memType="252" xdb:system="false" xdb:mutable="false"
        xdb:JavaType="boolean" default="false" xdb:hidden="true" xdb:transient="generated"
        xdb:baseProp="false"/>
    <attribute xdb:propNumber="776" name="HasUnresolvedLinks" type="boolean"
        xdb:memByteLength="1" xdb:memType="252" xdb:system="false" xdb:mutable="false"
        xdb:JavaType="boolean" default="false" xdb:hidden="false"
        xdb:transient="generated" xdb:baseProp="false"/>
  </complexType>
  <element xdb:propNumber="734" name="Resource" type="xdb:ResourceType" xdb:memType="258"
        xdb:system="false" xdb:mutable="false" xdb:SQLName="RESOURCE"
        xdb:SQLType="XDB$RESOURCE_T" xdb:SQLSchema="XDB" xdb:JavaType="XMLType"
        xdb:global="true" nillable="false" abstract="false" xdb:SQLInline="false"
        xdb:JavaInline="false" xdb:MemInline="false" xdb:maintainDOM="false"
        xdb:defaultTable="XDB$RESOURCE" xdb:defaultTableSchema="XDB"
        xdb:JavaClassname="oracle.xdb.ResourceBean"
        xdb:beanClassname="oracle.xdb.ResourceBean" xdb:numCols="33" minOccurs="1"
        maxOccurs="1"/>
  <element xdb:propNumber="775" name="Locks" type="xdb:locksType" xdb:memType="258"
        xdb:system="false" xdb:mutable="true" xdb:JavaType="XMLType" xdb:global="false"
        xdb:hidden="false" nillable="false" abstract="false" xdb:SQLInline="true"
        xdb:JavaInline="false" xdb:MemInline="false" xdb:maintainDOM="false"
        xdb:defaultTableSchema="XDB" xdb:numCols="1" minOccurs="0" maxOccurs="1"/>
</schema>
```

XDBResConfig.xsd: XML Schema for Resource Configuration

This section presents the Oracle XML DB supplied XML schema used to configure repository resources. This is accessible in Oracle XML DB Repository at path `/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/XDBResConfig.xsd`.

XDBResConfig.xsd

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:xdb="http://xmlns.oracle.com/xdb"
        xmlns:rescfg="http://xmlns.oracle.com/xdb/XDBResConfig.xsd"
        targetNamespace="http://xmlns.oracle.com/xdb/XDBResConfig.xsd"
        elementFormDefault="qualified" version="1.0">
  <annotation>
    <documentation>
      This XML schema declares the schema of an XDB resource configuration,
      which includes default ACL, event listeners and user configuration.
      It lists all XDB repository events that will be supported.
```

```
    Future extension can be added to support user-defined events and
    XML events.
  </documentation>
</annotation>
<simpleType name="language">
  <restriction base="string">
    <enumeration value="Java"/>
    <enumeration value="C"/>
    <enumeration value="PL/SQL"/>
  </restriction>
</simpleType>
<complexType name="existsNode">
  <all>
    <element name="XPath" type="string"/>
    <element name="namespace" type="string" minOccurs="0"/>
  </all>
</complexType>
<!-- listener pre-condition element  -->
<complexType name="condition">
  <all>
    <element name="existsNode" type="rescfg:existsNode" minOccurs="0"/>
  </all>
</complexType>
<complexType name="events">
  <all>
    <element name="Render" type="string" minOccurs="0"/>
    <element name="Pre-Create" type="string" minOccurs="0"/>
    <element name="Post-Create" type="string" minOccurs="0"/>
    <element name="Pre-Delete" type="string" minOccurs="0"/>
    <element name="Post-Delete" type="string" minOccurs="0"/>
    <element name="Pre-Update" type="string" minOccurs="0"/>
    <element name="Post-Update" type="string" minOccurs="0"/>
    <element name="Pre-Lock" type="string" minOccurs="0"/>
    <element name="Post-Lock" type="string" minOccurs="0"/>
    <element name="Pre-Unlock" type="string" minOccurs="0"/>
    <element name="Post-Unlock" type="string" minOccurs="0"/>
    <element name="Pre-LinkIn" type="string" minOccurs="0"/>
    <element name="Post-LinkIn" type="string" minOccurs="0"/>
    <element name="Pre-LinkTo" type="string" minOccurs="0"/>
    <element name="Post-LinkTo" type="string" minOccurs="0"/>
    <element name="Pre-UnlinkIn" type="string" minOccurs="0"/>
    <element name="Post-UnlinkIn" type="string" minOccurs="0"/>
    <element name="Pre-UnlinkFrom" type="string" minOccurs="0"/>
    <element name="Post-UnlinkFrom" type="string" minOccurs="0"/>
    <element name="Pre-CheckIn" type="string" minOccurs="0"/>
    <element name="Post-CheckIn" type="string" minOccurs="0"/>
    <element name="Pre-CheckOut" type="string" minOccurs="0"/>
    <element name="Post-CheckOut" type="string" minOccurs="0"/>
    <element name="Pre-UncheckOut" type="string" minOccurs="0"/>
    <element name="Post-UncheckOut" type="string" minOccurs="0"/>
    <element name="Pre-VersionControl" type="string" minOccurs="0"/>
    <element name="Post-VersionControl" type="string" minOccurs="0"/>
    <element name="Pre-Open" type="string" minOccurs="0"/>
    <element name="Post-Open" type="string" minOccurs="0"/>
    <element name="Pre-InconsistentUpdate" type="string" minOccurs="0"/>
    <element name="Post-InconsistentUpdate" type="string" minOccurs="0"/>
  </all>
</complexType>
<!-- event listener element  -->
<complexType name="event-listener">
```

```
  <all>
    <element name="description" type="string" minOccurs="0"/>
    <element name="schema" type="string" minOccurs="0"/>
    <element name="source" type="string"/>
    <element name="language" type="rescfg:language" minOccurs="0"/>
    <element name="pre-condition" type="rescfg:condition" minOccurs="0"/>
    <element name="events" type="rescfg:events" minOccurs="0"/>
  </all>
</complexType>
<complexType name="event-listeners">
  <sequence>
    <element name="listener" type="rescfg:event-listener" maxOccurs="unbounded"/>
  </sequence>
  <attribute name="default-schema" type="string" use="optional"/>
  <attribute name="default-language" type="rescfg:language" use="optional"/>
  <attribute name="set-invoker" type="boolean" default="false"/>
</complexType>
<complexType name="defaultPath">
  <all>
    <element name="pre-condition" type="rescfg:condition" minOccurs="0"/>
    <element name="path" type="string" minOccurs="0"/>
    <element name="resolvedpath" type="string"/>
    <element name="oid" type="hexBinary"/>
  </all>
</complexType>
<complexType name="defaultACL">
  <sequence>
    <element name="ACL" type="rescfg:defaultPath" maxOccurs="unbounded"/>
  </sequence>
</complexType>
<complexType name="defaultConfig">
  <sequence>
    <element name="configuration" type="rescfg:defaultPath" maxOccurs="unbounded"/>
  </sequence>
</complexType>
<simpleType name="link-type">
  <restriction base="string">
    <enumeration value="None"/>
    <enumeration value="Hard"/>
    <enumeration value="Weak"/>
    <enumeration value="Symbolic"/>
  </restriction>
</simpleType>
<simpleType name="path-format">
  <restriction base="string">
    <enumeration value="OID"/>
    <enumeration value="Named"/>
  </restriction>
</simpleType>
<simpleType name="link-metadata">
  <restriction base="string">
    <enumeration value="None"/>
    <enumeration value="Attributes"/>
    <enumeration value="All"/>
  </restriction>
</simpleType>
<simpleType name="unresolved-link">
  <restriction base="string">
    <enumeration value="Error"/>
    <enumeration value="SymLink"/>
```

```
    <enumeration value="Skip"/>
  </restriction>
</simpleType>
<simpleType name="conflict-rule">
  <restriction base="string">
    <enumeration value="Error"/>
    <enumeration value="Overwrite"/>
    <enumeration value="Syspath"/>
  </restriction>
</simpleType>
<simpleType name="section-type">
  <restriction base="string">
    <enumeration value="None"/>
    <enumeration value="Fragment"/>
    <enumeration value="Document"/>
  </restriction>
</simpleType>
<!-- XLinkConfig complex type -->
<complexType name="xlink-config">
  <sequence>
    <element name="LinkType" type="rescfg:link-type"/>
    <element name="PathFormat" type="rescfg:path-format" default="OID" minOccurs="0"/>
    <element name="LinkMetadata" type="rescfg:link-metadata" default="None" minOccurs="0"/>
    <element name="pre-condition" type="rescfg:condition" minOccurs="0"/>
  </sequence>
  <attribute name="UnresolvedLink" type="rescfg:unresolved-link" default="Error"/>
</complexType>
<!-- XIncludeConfig element -->
<complexType name="xinclude-config">
  <sequence>
    <element name="LinkType" type="rescfg:link-type"/>
    <element name="PathFormat" type="rescfg:path-format" default="OID" minOccurs="0"/>
    <element name="ConflictRule" type="rescfg:conflict-rule" default="Error" minOccurs="0"/>
  </sequence>
  <attribute name="UnresolvedLink" type="rescfg:unresolved-link" default="Error"/>
</complexType>
<!-- SectionConfig element -->
<complexType name="section-config">
  <sequence>
    <element name="Section" maxOccurs="unbounded">
      <complexType>
        <sequence>
          <element name="sectionPath" type="string"/>
          <element name="documentPath" type="string" minOccurs="0"/>
          <element name="namespace" type="string" minOccurs="0"/>
        </sequence>
        <attribute name="type" type="rescfg:section-type" default="None"/>
      </complexType>
    </element>
  </sequence>
</complexType>
<!-- ContentFormat element -->
<simpleType name="content-format">
  <restriction base="string">
    <enumeration value="text"/>
    <enumeration value="binary"/>
  </restriction>
</simpleType>
<!-- resource configuration element  -->
<complexType name="ResConfig">
```

```
    <all>
      <element name="defaultChildConfig" type="rescfg:defaultConfig" minOccurs="0"/>
      <element name="defaultChildACL" type="rescfg:defaultACL" minOccurs="0"/>
      <element name="event-listeners" type="rescfg:event-listeners" minOccurs="0"/>
      <element name="XLinkConfig" type="rescfg:xlink-config" minOccurs="0"/>
      <element name="XIncludeConfig" type="rescfg:xinclude-config" minOccurs="0"/>
      <element name="SectionConfig" type="rescfg:section-config" minOccurs="0"/>
      <element name="ContentFormat" type="rescfg:content-format" minOccurs="0"/>
      <!-- application data -->
      <element name="applicationData" minOccurs="0">
        <complexType>
          <sequence>
            <any namespace="##other" processContents="lax" maxOccurs="unbounded"/>
          </sequence>
        </complexType>
      </element>
    </all>
    <attribute name="enable" type="boolean" default="true"/>
    <attribute name="copy-on-inconsistent-update" type="boolean" use="optional"/>
  </complexType>
  <element name="ResConfig" type="rescfg:ResConfig" xdb:defaultTable="XDB$RESCONFIG"/>
</schema>
```

acl.xsd: XML Schema for ACLs

This section presents the Oracle Database supplied XML schema used to represent access control lists (ACLs).

acl.xsd

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://xmlns.oracle.com/xdb/acl.xsd" version="1.0"
        xmlns:xdb="http://xmlns.oracle.com/xdb"
        xmlns:xdbacl="http://xmlns.oracle.com/xdb/acl.xsd"
        elementFormDefault="qualified">
  <annotation>
    <documentation>
       This XML schema describes the structure of XDB ACL documents.

       Note : The "systemPrivileges" element below lists all supported
         system privileges and their aggregations.
         See dav.xsd for description of DAV privileges
       Note : The elements and attributes marked "hidden" are for
         internal use only.
    </documentation>
    <appinfo>
      <xdb:systemPrivileges>
        <xdbacl:all>
          <xdbacl:read-properties/>
          <xdbacl:read-contents/>
          <xdbacl:read-acl/>
          <xdbacl:update/>
          <xdbacl:link/>
          <xdbacl:unlink/>
          <xdbacl:unlink-from/>
          <xdbacl:write-acl-ref/>
          <xdbacl:update-acl/>
          <xdbacl:link-to/>
          <xdbacl:resolve/>
```

```
        <xdbacl:write-config/>
      </xdbacl:all>
    </xdb:systemPrivileges>
  </appinfo>
</annotation>
<!-- privilegeNameType (this is an emptycontent type) -->
<complexType name = "privilegeNameType"/>
<!-- privilegeName element
     All system and user privileges are in the substitutionGroup
     of this element.
  -->
<element name = "privilegeName" type="xdbacl:privilegeNameType"
         xdb:defaultTable=""/>
<!-- all system privileges in the XDB ACL namespace -->
<element name = "read-properties" type="xdbacl:privilegeNameType"
         substitutionGroup="xdbacl:privilegeName" xdb:defaultTable=""/>
<element name = "read-contents" type="xdbacl:privilegeNameType"
         substitutionGroup="xdbacl:privilegeName" xdb:defaultTable=""/>
<element name = "read-acl" type="xdbacl:privilegeNameType"
         substitutionGroup="xdbacl:privilegeName" xdb:defaultTable=""/>
<element name = "update" type="xdbacl:privilegeNameType"
         substitutionGroup="xdbacl:privilegeName" xdb:defaultTable=""/>
<element name = "link" type="xdbacl:privilegeNameType"
         substitutionGroup="xdbacl:privilegeName" xdb:defaultTable=""/>
<element name = "unlink" type="xdbacl:privilegeNameType"
         substitutionGroup="xdbacl:privilegeName" xdb:defaultTable=""/>
<element name = "unlink-from" type="xdbacl:privilegeNameType"
         substitutionGroup="xdbacl:privilegeName" xdb:defaultTable=""/>
<element name = "write-acl-ref" type="xdbacl:privilegeNameType"
         substitutionGroup="xdbacl:privilegeName" xdb:defaultTable=""/>
<element name = "update-acl" type="xdbacl:privilegeNameType"
         substitutionGroup="xdbacl:privilegeName" xdb:defaultTable=""/>
<element name = "link-to" type="xdbacl:privilegeNameType"
         substitutionGroup="xdbacl:privilegeName" xdb:defaultTable=""/>
<element name = "resolve" type="xdbacl:privilegeNameType"
         substitutionGroup="xdbacl:privilegeName" xdb:defaultTable=""/>
<element name = "all" type="xdbacl:privilegeNameType"
         substitutionGroup="xdbacl:privilegeName" xdb:defaultTable=""/>
<!-- privilege element -->
<element name = "privilege" xdb:defaultTable="">
  <complexType>
    <sequence>
      <any maxOccurs="unbounded" processContents="lax"/>
    </sequence>
  </complexType>
</element>
<!-- ace element -->
<element name = "ace" xdb:defaultTable="">
  <complexType>
    <sequence>
      <element name = "grant" type = "boolean"/>
      <choice>
        <element name="invert" xdb:transient="generated">
          <complexType>
            <sequence>
              <element name="principal" type="string"
                       xdb:transient="generated" />
            </sequence>
          </complexType>
        </element>
```

```
        <element name="principal" type="string" xdb:transient="generated"/>
      </choice>
      <element ref="xdbacl:privilege" minOccurs="1"/>
      <!-- "any" contain all app info for an ACE e.g.reason for creation -->
      <any minOccurs="0" maxOccurs="unbounded" namespace="##other"
           processContents="lax"/>
      <!-- HIDDEN ELEMENTS -->
      <choice minOccurs="0">
        <element name = "principalID" type = "hexBinary"
                 xdb:baseProp="true" xdb:hidden="true"/>
        <element name = "principalString" type = "string"
                 xdb:baseProp="true" xdb:hidden="true"/>
      </choice>
      <element name = "flags" type = "unsignedInt" minOccurs="0"
                xdb:baseProp="true" xdb:hidden="true"/>
    </sequence>
    <attribute name = "collection" type = "boolean"
                xdb:transient="generated" use="optional"/>
    <attribute name = "principalFormat"
                xdb:transient="generated" use="optional">
      <simpleType>
        <restriction base="string">
          <enumeration value="ShortName"/>
          <enumeration value="DistinguishedName"/>
          <enumeration value="GUID"/>
          <enumeration value="XSName"/>
          <enumeration value="ApplicationName"/>
        </restriction>
      </simpleType>
    </attribute>
    <attribute name = "start_date" type = "dateTime" use = "optional"/>
    <attribute name = "end_date" type = "dateTime" use = "optional"/>
  </complexType>
</element>
<!-- acl element -->
<complexType name="inheritanceType">
  <attribute name="type" type="string" use="required"/>
  <attribute name="href" type="string" use="required"/>
</complexType>
<complexType name="aclType">
  <sequence>
    <element name = "schemaURL" type = "string" minOccurs="0"
             xdb:transient="generated"/>
    <element name = "elementName" type = "string" minOccurs="0"
             xdb:transient="generated"/>
    <element name = "security-class" type = "QName" minOccurs="0"/>
    <choice minOccurs="0">
      <element name="extends-from" type="xdbacl:inheritanceType"/>
      <element name="constrained-with" type="xdbacl:inheritanceType"/>
    </choice>
    <element ref = "xdbacl:ace" minOccurs="0" maxOccurs = "unbounded"/>
    <!-- this "any" contains all application specific info for an ACL,
         e.g., reason for creation  -->
    <any minOccurs="0" maxOccurs="unbounded" namespace="##other"
         processContents="lax"/>
    <!-- HIDDEN ELEMENTS -->
    <element name = "schemaOID" type = "hexBinary" minOccurs="0"
             xdb:baseProp="true" xdb:hidden="true"/>
    <element name = "elementNum" type = "unsignedInt" minOccurs="0"
             xdb:baseProp="true" xdb:hidden="true"/>
```

```
      </sequence>
      <attribute name = "shared" type = "boolean" default="true"/>
      <attribute name = "description" type = "string"/>
    </complexType>
    <complexType name="rule-based-acl">
      <complexContent>
        <extension base="xdbacl:aclType">
          <sequence>
            <element name = "param" minOccurs="0" maxOccurs="unbounded">
              <complexType>
                <simpleContent>
                  <extension base="string">
                    <attribute name = "name" type = "string" use = "required"/>
                  </extension>
                </simpleContent>
              </complexType>
            </element>
          </sequence>
        </extension>
      </complexContent>
    </complexType>
    <element name = "acl" type="xdbacl:aclType" xdb:defaultTable = "XDB$ACL"/>
    <element name = "write-config" type="xdbacl:privilegeNameType"
             substitutionGroup="xdbacl:privilegeName" xdb:defaultTable=""/>
  </schema>
```

xdbconfig.xsd: XML Schema for Configuring Oracle XML DB

File `xdbconfig.xsd` contains the Oracle XML DB supplied XML schema used to configure Oracle XML DB.

Note: The value of attribute `value` of element `pattern` has been split here for documentation purposes. In reality, the value is not split (no newline characters), but is one long string.

xdbconfig.xsd

```
<schema targetNamespace="http://xmlns.oracle.com/xdb/xdbconfig.xsd"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:xdbc="http://xmlns.oracle.com/xdb/xdbconfig.xsd"
        xmlns:xdb="http://xmlns.oracle.com/xdb"
        version="1.0" elementFormDefault="qualified">
  <element name="xdbconfig" xdb:defaultTable="XDB$CONFIG">
    <complexType>
      <sequence>

        <!-- predefined XDB properties - these should NOT be changed -->
        <element name="sysconfig">
          <complexType>
            <sequence>

              <!-- generic XDB properties -->
              <element name="acl-max-age" type="unsignedInt" default="15"/>
              <element name="acl-cache-size" type="unsignedInt" default="32"/>
              <element name="invalid-pathname-chars" type="string" default=""/>
              <element name="case-sensitive" type="boolean" default="true"/>
              <element name="call-timeout"   type="unsignedInt" default="300"/>
              <element name="max-link-queue" type="unsignedInt" default="65536"/>
```

```
<element name="max-session-use" type="unsignedInt" default="100"/>
<element name="persistent-sessions" type="boolean" default="false"/>
<element name="default-lock-timeout" type="unsignedInt"
         default="3600"/>
<element name="xdbcore-logfile-path" type="string"
         default="/sys/log/xdblog.xml"/>
<element name="xdbcore-log-level" type="unsignedInt"
         default="0"/>
<element name="resource-view-cache-size" type="unsignedInt"
         default="1048576"/>
<element name="case-sensitive-index-clause" type="string"
         minOccurs="0"/>

<!-- protocol specific properties -->
<element name="protocolconfig">
  <complexType>
    <sequence>

      <!-- these apply to all protocols -->
      <element name="common">
        <complexType>
          <sequence>
            <element name="extension-mappings">
              <complexType>
                <sequence>
                  <element name="mime-mappings"
                           type="xdbc:mime-mapping-type"/>
                  <element name="lang-mappings"
                           type="xdbc:lang-mapping-type"/>
                  <element name="charset-mappings"
                           type="xdbc:charset-mapping-type"/>
                  <element name="encoding-mappings"
                           type="xdbc:encoding-mapping-type"/>
                  <element name="xml-extensions"
                           type="xdbc:xml-extension-type"
                           minOccurs="0"/>
                </sequence>
              </complexType>
            </element>
            <element name="session-pool-size" type="unsignedInt"
                     default="50"/>
            <element name="session-timeout" type="unsignedInt"
                     default="6000"/>
          </sequence>
        </complexType>
      </element>

      <!-- FTP specific -->
      <element name="ftpconfig">
        <complexType>
          <sequence>
            <element name="ftp-port" type="unsignedShort"
                     default="2100"/>
            <element name="ftp-listener" type="string"/>
            <element name="ftp-protocol" type="string"/>
            <element name="logfile-path" type="string"
                     default="/sys/log/ftplog.xml"/>
            <element name="log-level" type="unsignedInt"
                     default="0"/>
            <element name="session-timeout"  type="unsignedInt"
```

```
                default="6000"/>
        <element name="buffer-size" default="8192">
          <simpleType>
            <restriction base="unsignedInt">
              <minInclusive value="1024"/>       <!-- 1KB -->
              <maxInclusive value="1048496"/>    <!-- 1MB -->
            </restriction>
          </simpleType>
        </element>
        <element name="ftp-welcome-message" type="string"
                minOccurs="0" maxOccurs="1"/>
      </sequence>
    </complexType>
  </element>

  <!-- HTTP specific -->
  <element name="httpconfig">
    <complexType>
      <sequence>
        <element name="http-port" type="unsignedShort"
                default="8080"/>
        <element name="http-listener" type="string"/>
        <element name="http-protocol" type="string"/>
        <element name="max-http-headers" type="unsignedInt"
                default="64"/>
        <element name="max-header-size" type="unsignedInt"
                default="4096"/>
        <element name="max-request-body" type="unsignedInt"
                default="2000000000" minOccurs="1"/>
        <element name="session-timeout"  type="unsignedInt"
                default="6000"/>
        <element name="server-name" type="string"/>
        <element name="logfile-path" type="string"
                default="/sys/log/httplog.xml"/>
        <element name="log-level" type="unsignedInt"
                default="0"/>
        <element name="servlet-realm" type="string"
                minOccurs="0"/>
        <element name="webappconfig">
          <complexType>
            <sequence>
              <element name="welcome-file-list"
                       type="xdbc:welcome-file-type"/>
              <element name="error-pages"
                       type="xdbc:error-page-type"/>
              <element name="servletconfig"
                       type="xdbc:servlet-config-type"/>
            </sequence>
          </complexType>
        </element>
        <element name="default-url-charset" type="string"
                minOccurs="0"/>
        <element name="http2-port" type="unsignedShort"
                minOccurs="0"/>
        <element name="http2-protocol" type="string"
                default="tcp" minOccurs="0"/>
        <element name="plsql" minOccurs="0">
          <complexType>
            <sequence>
              <element name="log-level"
```

```
                    type="unsignedInt" minOccurs="0"/>
            <element name="max-parameters"
                    type="unsignedInt" minOccurs="0"/>
          </sequence>
        </complexType>
      </element>
      <element name="allow-repository-anonymous-access"
               minOccurs="0" default="false" type="boolean"/>
      <element name="authentication" minOccurs="0"
               maxOccurs="1">
        <complexType>
          <sequence>
            <element name="allow-mechanism" minOccurs="1"
                    maxOccurs="unbounded">
              <simpleType>
                <restriction base="string">
                  <enumeration value="digest"/>
                  <enumeration value="basic" />
                  <enumeration value="custom"/>
                </restriction>
              </simpleType>
            </element>
            <element name="digest-auth" minOccurs="0"
                    maxOccurs="1">
              <complexType>
                <sequence>
                  <element name="nonce-timeout"
                           type="unsignedInt"
                           minOccurs="1" maxOccurs="1"
                           default="300"/>
                </sequence>
              </complexType>
            </element>
          </sequence>
        </complexType>
      </element>
      <element name="http-host" type="string" minOccurs="0"/>
      <element name="http2-host" type="string" minOccurs="0"/>
      <element name="custom-authentication"
               type="xdbc:custom-authentication-type"
               minOccurs="0"/>
      <element name="realm" type="string" minOccurs="0"/>
      <element name="respond-with-server-info" type="boolean"
               default="true" minOccurs="0"/>
      <element name="expire" type="xdbc:expire-type"
               minOccurs="0"/>
    </sequence>
  </complexType>
</element>
<element name="nfsconfig" minOccurs="0">
  <complexType>
    <sequence>
      <element name="nfs-port" type="unsignedShort"
               default="2049"/>
      <element name="nfs-listener" type="string"/>
      <element name="nfs-protocol" type="string"/>
      <element name="logfile-path" type="string"
               default="/sys/log/nfslog.xml"/>
      <element name="log-level" type="unsignedInt"
               default="0"/>
```

```
                <element name="nfs-exports"
                         type="xdbc:nfs-exports-type"/>
              </sequence>
            </complexType>
          </element>
        </sequence>
      </complexType>
    </element>
    <element name="schemaLocation-mappings"
             type="xdbc:schemaLocation-mapping-type" minOccurs="0"/>
    <element name="xdbcore-xobmem-bound" type="unsignedInt"
             default="1024" minOccurs="0"/>
    <element name="xdbcore-loadableunit-size" type="unsignedInt"
             default="16" minOccurs="0"/>
    <element name="folder-hard-links" type="boolean" default="false"
             minOccurs="0"/>
    <element name="non-folder-hard-links" type="boolean" default="true"
             minOccurs="0"/>
    <element name="copy-on-inconsistent-update" type="boolean"
             default="false" minOccurs="0"/>
    <element name="rollback-on-sync-error" type="boolean"
             default="false" minOccurs="0"/>
    <element name="acl-evaluation-method" default="deny-trumps-grant"
             minOccurs="0">
      <simpleType>
        <restriction base="string">
          <enumeration value="deny-trumps-grant"/>
          <enumeration value="ace-order"/>
        </restriction>
      </simpleType>
    </element>
    <element name="default-workspace" type="string" minOccurs="0"/>
    <element name="num_job_queue_processes" type="unsignedInt"
             minOccurs="0"/>
    <element name="allow-authentication-trust" type="boolean"
             default="false" minOccurs="0"/>
    <element name="custom-authentication-trust"
             type="xdbc:custom-authentication-trust-type"
             minOccurs="0"/>
    <element name="default-type-mappings" minOccurs="0">
      <simpleType>
        <restriction base="string">
          <enumeration value="pre-11.2"/>
          <enumeration value="post-11.2"/>
        </restriction>
      </simpleType>
    </element>
    <element name="localApplicationGroupStore" type="boolean"
             default="true" minOccurs="0"/>
  </sequence>
</complexType>
</element>

<!-- users can add any properties they want here -->
<element name="userconfig" minOccurs="0">
  <complexType>
    <sequence>
      <any maxOccurs="unbounded" namespace="##other"/>
    </sequence>
  </complexType>
```

```
      </element>
    </sequence>
  </complexType>
</element>
<complexType name="welcome-file-type">
  <sequence>
    <element name="welcome-file" minOccurs="0" maxOccurs="unbounded">
      <simpleType>
        <restriction base="string">
          <pattern value="[^/]*"/>
        </restriction>
      </simpleType>
    </element>
  </sequence>
</complexType>

<!-- customized error pages -->
<complexType name="error-page-type">
  <sequence>
    <element name="error-page" minOccurs="0" maxOccurs="unbounded">
      <complexType>
        <sequence>
          <choice>
            <element name="error-code">
              <simpleType>
                <restriction base="positiveInteger">
                  <minInclusive value="100"/>
                  <maxInclusive value="999"/>
                </restriction>
              </simpleType>
            </element>

            <!-- Fully qualified classname of a Java exception type -->
            <element name="exception-type" type="string"/>
            <element name="OracleError">
              <complexType>
                <sequence>
                   <element name="facility" type="string" default="ORA"/>
                   <element name="errnum" type="unsignedInt"/>
                </sequence>
              </complexType>
            </element>
          </choice>
          <element name="location" type="anyURI"/>
        </sequence>
      </complexType>
    </element>
  </sequence>
</complexType>

<!-- parameter for a servlet: name, value pair and a description  -->
<complexType name="param">
  <sequence>
    <element name="param-name" type="string"/>
    <element name="param-value" type="string"/>
    <element name="description" type="string"/>
  </sequence>
</complexType>
<complexType name="servlet-config-type">
  <sequence>
```

```
<element name="servlet-mappings">
  <complexType>
    <sequence>
      <element name="servlet-mapping" minOccurs="0"
               maxOccurs="unbounded">
        <complexType>
          <sequence>
            <element name="servlet-pattern" type="string"/>
            <element name="servlet-name" type="string"/>
          </sequence>
        </complexType>
      </element>
    </sequence>
  </complexType>
</element>
<element name="servlet-list">
  <complexType>
    <sequence>
      <element name="servlet" minOccurs="0" maxOccurs="unbounded">
        <complexType>
          <sequence>
            <element name="servlet-name" type="string"/>
            <element name="servlet-language">
              <simpleType>
                <restriction base="string">
                  <enumeration value="C"/>
                  <enumeration value="Java"/>
                  <enumeration value="PL/SQL"/>
                </restriction>
              </simpleType>
            </element>
            <element name="icon" type="string" minOccurs="0"/>
            <element name="display-name" type="string"/>
            <element name="description" type="string" minOccurs="0"/>
            <choice>
              <element name="servlet-class" type="string" minOccurs="0"/>
              <element name="jsp-file" type="string" minOccurs="0"/>
              <element name="plsql" type="xdbc:plsql-servlet-config"
                       minOccurs="0"/>
            </choice>
            <element name="servlet-schema" type="string" minOccurs="0"/>
            <element name="init-param" minOccurs="0"
                     maxOccurs="unbounded" type="xdbc:param"/>
            <element name="load-on-startup" type="string" minOccurs="0"/>
            <element name="security-role-ref" minOccurs="0"
                     maxOccurs="unbounded">
              <complexType>
                <sequence>
                  <element name="description" type="string" minOccurs="0"/>
                  <element name="role-name" type="string"/>
                  <element name="role-link" type="string"/>
                </sequence>
              </complexType>
            </element>
          </sequence>
        </complexType>
      </element>
    </sequence>
  </complexType>
</element>
```

```
  </sequence>
</complexType>
<complexType name="lang-mapping-type">
  <sequence>
    <element name="lang-mapping" minOccurs="0" maxOccurs="unbounded">
      <complexType>
        <sequence>
          <element name="extension" type="xdbc:exttype"/>
          <element name="lang" type="string"/>
        </sequence>
      </complexType>
    </element>
  </sequence>
</complexType>
<complexType name="charset-mapping-type">
  <sequence>
    <element name="charset-mapping" minOccurs="0" maxOccurs="unbounded">
      <complexType>
        <sequence>
          <element name="extension" type="xdbc:exttype"/>
          <element name="charset" type="string"/>
        </sequence>
      </complexType>
    </element>
  </sequence>
</complexType>
<complexType name="encoding-mapping-type">
  <sequence>
    <element name="encoding-mapping" minOccurs="0" maxOccurs="unbounded">
      <complexType>
        <sequence>
          <element name="extension" type="xdbc:exttype"/>
          <element name="encoding" type="string"/>
        </sequence>
      </complexType>
    </element>
  </sequence>
</complexType>
<complexType name="mime-mapping-type">
  <sequence>
    <element name="mime-mapping" minOccurs="0" maxOccurs="unbounded">
      <complexType>
        <sequence>
          <element name="extension" type="xdbc:exttype"/>
          <element name="mime-type" type="string"/>
        </sequence>
      </complexType>
    </element>
  </sequence>
</complexType>
<complexType name="xml-extension-type">
  <sequence>
    <element name="extension" type="xdbc:exttype"
             minOccurs="0" maxOccurs="unbounded">
    </element>
  </sequence>
</complexType>
<complexType name="schemaLocation-mapping-type">
  <sequence>
    <element name="schemaLocation-mapping"
```

```
            minOccurs="0" maxOccurs="unbounded">
      <complexType>
        <sequence>
          <element name="namespace" type="string"/>
          <element name="element" type="string"/>
          <element name="schemaURL" type="string"/>
        </sequence>
      </complexType>
    </element>
  </sequence>
</complexType>
<complexType name="plsql-servlet-config">
  <sequence>
    <element name="database-username" type="string" minOccurs="0"/>
    <element name="authentication-mode" minOccurs="0">
      <simpleType>
        <restriction base="string">
          <enumeration value="Basic"/>
          <enumeration value="SingleSignOn"/>
          <enumeration value="GlobalOwa"/>
          <enumeration value="CustomOwa"/>
          <enumeration value="PerPackageOwa"/>
        </restriction>
      </simpleType>
    </element>
    <element name="session-cookie-name" type="string" minOccurs="0"/>
    <element name="session-state-management" minOccurs="0">
      <simpleType>
        <restriction base="string">
          <enumeration value="StatelessWithResetPackageState"/>
          <enumeration value="StatelessWithFastResetPackageState"/>
          <enumeration value="StatelessWithPreservePackageState"/>
        </restriction>
      </simpleType>
    </element>
    <element name="max-requests-per-session" type="unsignedInt" minOccurs="0"/>
    <element name="default-page" type="string" minOccurs="0"/>
    <element name="document-table-name" type="string" minOccurs="0"/>
    <element name="document-path" type="string" minOccurs="0"/>
    <element name="document-procedure" type="string" minOccurs="0"/>
    <element name="upload-as-long-raw" type="string" minOccurs="0"
            maxOccurs="unbounded"/>
    <element name="path-alias" type="string" minOccurs="0"/>
    <element name="path-alias-procedure" type="string" minOccurs="0"/>
    <element name="exclusion-list" type="string" minOccurs="0"
            maxOccurs="unbounded"/>
    <element name="cgi-environment-list" type="string" minOccurs="0"
            maxOccurs="unbounded"/>
    <element name="compatibility-mode" type="unsignedInt" minOccurs="0"/>
    <element name="nls-language" type="string" minOccurs="0"/>
    <element name="fetch-buffer-size" type="unsignedInt" minOccurs="0"/>
    <element name="error-style" minOccurs="0">
      <simpleType>
        <restriction base="string">
          <enumeration value="ApacheStyle"/>
          <enumeration value="ModplsqlStyle"/>
          <enumeration value="DebugStyle"/>
        </restriction>
      </simpleType>
    </element>
```

```
    <element name="transfer-mode" minOccurs="0">
      <simpleType>
        <restriction base="string">
          <enumeration value="Char"/>
          <enumeration value="Raw"/>
        </restriction>
      </simpleType>
    </element>
    <element name="before-procedure" type="string" minOccurs="0"/>
    <element name="after-procedure" type="string" minOccurs="0"/>
    <element name="bind-bucket-lengths" type="unsignedInt" minOccurs="0"
             maxOccurs="unbounded"/>
    <element name="bind-bucket-widths" type="unsignedInt" minOccurs="0"
             maxOccurs="unbounded"/>
    <element name="always-describe-procedure" minOccurs="0">
      <simpleType>
        <restriction base="string">
          <enumeration value="On"/>
          <enumeration value="Off"/>
        </restriction>
      </simpleType>
    </element>
    <element name="info-logging" minOccurs="0">
      <simpleType>
        <restriction base="string">
          <enumeration value="InfoDebug"/>
        </restriction>
      </simpleType>
    </element>
    <element name="owa-debug-enable" minOccurs="0">
      <simpleType>
        <restriction base="string">
          <enumeration value="On"/>
          <enumeration value="Off"/>
        </restriction>
      </simpleType>
    </element>
    <element name="request-validation-function" type="string" minOccurs="0"/>
    <element name="input-filter-enable" minOccurs="0">
      <simpleType>
        <restriction base="string">
          <enumeration value="On"/>
          <enumeration value="Off"/>
        </restriction>
      </simpleType>
    </element>
    <element name="database-edition" type="string" minOccurs="0"/>
  </sequence>
</complexType>
<simpleType name="exttype">
  <restriction base="string">
    <pattern value="[^\*\./]*"/>
  </restriction>
</simpleType>
<simpleType name="ipaddress">
  <restriction base="string">
    <maxLength value="40" />
  </restriction>
</simpleType>
<complexType name="nfs-exports-type">
```

```
    <sequence>
      <element name="nfs-export" minOccurs="1" maxOccurs="unbounded">
        <complexType>
          <sequence>
            <element name="nfs-clientgroup">
              <complexType>
                <sequence>
                  <element name="nfs-client" minOccurs="1" maxOccurs="unbounded">
                    <complexType>
                      <sequence>
                        <choice>
                          <element name="nfs-client-subnet"
                                   type="xdbc:ipaddress"/>
                          <element name="nfs-client-dnsname" type="string"/>
                          <element name="nfs-client-address"
                                   type="xdbc:ipaddress"/>
                        </choice>
                        <element name="nfs-client-netmask"
                                 type="xdbc:ipaddress"/>
                      </sequence>
                    </complexType>
                  </element>
                </sequence>
              </complexType>
            </element>
            <element name="nfs-export-paths">
              <complexType>
                <sequence>
                  <element name="nfs-export-path" minOccurs="1"
                           maxOccurs="unbounded">
                    <complexType>
                      <sequence>
                        <element name="path" type="string"/>
                        <element name="mode">
                          <simpleType>
                            <restriction base="string">
                              <enumeration value="read-write"/>
                              <enumeration value="read-only"/>
                            </restriction>
                          </simpleType>
                        </element>
                      </sequence>
                    </complexType>
                  </element>
                </sequence>
              </complexType>
            </element>
          </sequence>
        </complexType>
      </element>
    </sequence>
  </complexType>
  <complexType name="custom-authentication-type">
    <sequence>
      <element name="custom-authentication-mappings">
        <complexType>
          <sequence>
            <element name="custom-authentication-mapping" minOccurs="0"
                     maxOccurs="unbounded">
              <complexType>
```

```
            <sequence>
              <element name="authentication-pattern" type="string"/>
              <element name="authentication-name" type="string"/>
              <element name="authentication-trust-name" type="string"
                       minOccurs="0"/>
              <element name="user-prefix" type="string" minOccurs="0"/>
              <element name="on-deny" minOccurs="0">
                <simpleType><restriction base="string">
                  <enumeration value="next-custom"/>
                  <enumeration value="basic"/>
                </restriction></simpleType>
              </element>
            </sequence>
          </complexType>
        </element>
      </sequence>
    </complexType>
  </element>
  <element name="custom-authentication-list">
    <complexType>
      <sequence>
        <element name="authentication" minOccurs="0" maxOccurs="unbounded">
          <complexType>
            <sequence>
              <element name="authentication-name" type="string"/>
              <element name="authentication-description" type="string"
                       minOccurs="0"/>
              <element name="authentication-implement-schema" type="string"/>
              <element name="authentication-implement-method" type="string"/>
              <element name="authentication-implement-language">
                <simpleType>
                  <restriction base="string">
                    <enumeration value="PL/SQL"/>
                  </restriction>
                </simpleType>
              </element>
            </sequence>
          </complexType>
        </element>
      </sequence>
    </complexType>
  </element>
  <element name="custom-authentication-trust"
           type="xdbc:custom-authentication-trust-type" minOccurs="0"/>
</sequence>
</complexType>
<complexType name="custom-authentication-trust-type">
  <sequence>
    <element name="trust-scheme"  minOccurs="0" maxOccurs="unbounded">
      <complexType>
        <sequence>
          <element name="trust-scheme-name" type="string"/>
          <element name="requireParsingSchema" type="boolean" default="true"
                   minOccurs="0"/>
          <element name="allowRegistration" type="boolean" default="true"
                   minOccurs="0"/>
          <element name="trust-scheme-description" type="string" minOccurs="0"/>
          <element name="trusted-session-user" type="string"  minOccurs="1"
                   maxOccurs="unbounded"/>
          <element name="trusted-parsing-schema" type="string" minOccurs="0"
```

```
                    maxOccurs="unbounded"/>
          </sequence>
       </complexType>
      </element>
    </sequence>
  </complexType>
  <complexType name="expire-type">
    <sequence>
      <element name="expire-mapping" minOccurs="0" maxOccurs="unbounded">
        <complexType>
          <sequence>
            <element name="expire-pattern" type="string"/>
            <element name="expire-default">
              <simpleType>
                <restriction base="string">
                  <pattern value=
"(now|modification)(\s(plus))?(\s(([1]\s(year))|
([0-9]*\s(years))))?(\s(([1]\s(month))|([0-9]*\s(months))))?(\s(([1]\s(week))|
([0-9]*\s(weeks))))?(\s(([1]\s(day))|([0-9]*\s(days))))?(\s(([1]\s(hour))|
([0-9]*\s(hours))))?(\s(([1]\s(minute))|
([0-9]*\s(minutes))))?(\s(([1]\s(second))|([0-9]*\s(seconds))))?"/>[1]
                </restriction>
              </simpleType>
            </element>
          </sequence>
        </complexType>
      </element>
    </sequence>
  </complexType>
</schema>
```

xdiff.xsd: XML Schema for Comparing Schemas for In-Place Evolution

`xdiff.xsd`, is the Oracle XML DB-supplied XML schema to which the document specified as the `diffXML` parameter to procedure `DBMS_XMLSCHEMA.inPlaceEvolve` must conform.

xdiff.xsd

```
<schema targetNamespace="http://xmlns.oracle.com/xdb/xdiff.xsd"
    xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:xd="http://xmlns.oracle.com/xdb/xdiff.xsd"
    version="1.0" elementFormDefault="qualified"
    attributeFormDefault="qualified">
    <annotation>
        <documentation xml:lang="en">
         Defines the structure of XML documents that capture the difference
         between two XML documents. Changes that are not supported by Oracle
         XmlDiff may not be expressible in this schema.

        'oracle-xmldiff' PI:

        We use 'oracle-xmldiff' PI to describe certain aspects of the diff.
        This should be the first element of top level xdiff element.

        operations-in-docorder:
```

[1] The value of attribute `value` has been split here for documentation purposes. In reality, the value is one long string, with no line breaks.

```
    Can be either 'true' or 'false'.
    If true, the operations in the diff document refer to the
    elements of the input doc in the same order as document order. Output of
    global algorithm meets this requirement while local does not.

    output-model: output models for representing the diff. Can be either
    'Snapshot' or 'Current'.

    Snapshot model:
    Each operation uses Xpaths as if no operations
    have been applied to the input document. (like UNIX diff)
    Default and works for both Xmldiff and XmlPatch.
    For XmlPatch to handle this model, "operations-in-docorder" must be
    true and the Xpaths must be simple. (see XmlDif C API documentation).

    Current model :
    Each operation uses Xpaths as if all operations till the previous one
    have been applied to the input document. Not implemented for Xmldiff.
    Works with XmlPatch.

    <!-- Example:
        <?oracle-xmldiff operations-in-docorder="true" output-model=
        "snapshot" diff-algorithm="global"?>
    -->
    </documentation>
</annotation>
<!-- Enumerate the supported node types -->
<simpleType name="xdiff-nodetype">
    <restriction base="string">
        <enumeration value="element"/>
        <enumeration value="attribute"/>
        <enumeration value="text"/>
        <enumeration value="cdata"/>
        <enumeration value="entity-reference"/>
        <enumeration value="entity"/>
        <enumeration value="processing-instruction"/>
        <enumeration value="notation"/>
        <enumeration value="comment"/>
     </restriction>
</simpleType>

<element name="xdiff">
    <complexType>
        <choice minOccurs="0" maxOccurs="unbounded">
            <element name="append-node">
                <complexType>
                    <sequence>
                        <element name="content" type="anyType"/>
                    </sequence>
                    <attribute name="node-type" type="xd:xdiff-nodetype"/>
                    <attribute name="xpath" type="string"/>
                    <attribute name="parent-xpath" type="string"/>
                    <attribute name="attr-local" type="string"/>
                    <attribute name="attr-uri" type="string"/>
                </complexType>
            </element>

            <element name="insert-node-before">
                <complexType>
                    <sequence>
```

```
                        <element name="content" type="anyType"/>
                    </sequence>
                    <attribute name="xpath" type="string"/>
                    <attribute name="node-type" type="xd:xdiff-nodetype"/>

                </complexType>
            </element>

            <element name="delete-node">
                <complexType>
                    <attribute name="node-type" type="xd:xdiff-nodetype"/>
                    <attribute name="xpath" type="string"/>
                    <attribute name="parent-xpath" type="string"/>
                    <attribute name="attr-local" type="string"/>
                    <attribute name="attr-uri" type="string"/>
                </complexType>
            </element>
             <element name="update-node">
                <complexType>
                    <sequence>
                        <element name="content" type="anyType"/>
                    </sequence>
                    <attribute name="node-type" type="xd:xdiff-nodetype"/>
                    <attribute name="parent-xpath" type="string"/>
                    <attribute name="xpath" type="string"/>
                    <attribute name="attr-local" type="string"/>
                    <attribute name="attr-uri" type="string"/>
                </complexType>
            </element>
            <element name="rename-node">
                <complexType>
                    <sequence>
                        <element name="content" type="anyType"/>
                    </sequence>
                    <attribute name="xpath" type="string"/>
                    <attribute name="node-type" type="xd:xdiff-nodetype"/>
                </complexType>
            </element>
        </choice>
     <attribute name="xdiff-version" type="string"/>
    </complexType>
  </element>
</schema>
```

Purchase-Order XML Schemas

This section contains the complete listings of the annotated purchase-order XML schemas used in various examples, particularly in Chapter 3. Example A–2 represents a modified version of Example A–1. The modification is used in Chapter 10 to illustrate XML schema evolution.

Example A–1 is the complete listing of the annotated XML schema presented in Example 3–10 on page 3-20.

Example A–1 Annotated Purchase-Order XML Schema, purchaseOrder.xsd

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:xdb="http://xmlns.oracle.com/xdb"
           version="1.0"
           xdb:storeVarrayAsTable="true">
```

```
<xs:element name="PurchaseOrder" type="PurchaseOrderType" xdb:defaultTable="PURCHASEORDER"/>
<xs:complexType name="PurchaseOrderType" xdb:SQLType="PURCHASEORDER_T">
  <xs:sequence>
    <xs:element name="Reference" type="ReferenceType" minOccurs="1" xdb:SQLName="REFERENCE"/>
    <xs:element name="Actions" type="ActionsType" xdb:SQLName="ACTIONS"/>
    <xs:element name="Reject" type="RejectionType" minOccurs="0" xdb:SQLName="REJECTION"/>
    <xs:element name="Requestor" type="RequestorType" xdb:SQLName="REQUESTOR"/>
    <xs:element name="User" type="UserType" minOccurs="1" xdb:SQLName="USERID"/>
    <xs:element name="CostCenter" type="CostCenterType" xdb:SQLName="COST_CENTER"/>
    <xs:element name="ShippingInstructions" type="ShippingInstructionsType"
                xdb:SQLName="SHIPPING_INSTRUCTIONS"/>
    <xs:element name="SpecialInstructions" type="SpecialInstructionsType"
                xdb:SQLName="SPECIAL_INSTRUCTIONS"/>
    <xs:element name="LineItems" type="LineItemsType" xdb:SQLName="LINEITEMS"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="LineItemsType" xdb:SQLType="LINEITEMS_T">
  <xs:sequence>
    <xs:element name="LineItem" type="LineItemType" maxOccurs="unbounded"
                xdb:SQLName="LINEITEM" xdb:SQLCollType="LINEITEM_V"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="LineItemType" xdb:SQLType="LINEITEM_T">
  <xs:sequence>
    <xs:element name="Description" type="DescriptionType"
                xdb:SQLName="DESCRIPTION"/>
    <xs:element name="Part" type="PartType" xdb:SQLName="PART"/>
  </xs:sequence>
  <xs:attribute name="ItemNumber" type="xs:integer" xdb:SQLName="ITEMNUMBER"
                xdb:SQLType="NUMBER"/>
</xs:complexType>
<xs:complexType name="PartType" xdb:SQLType="PART_T">
  <xs:attribute name="Id" xdb:SQLName="PART_NUMBER" xdb:SQLType="VARCHAR2">
    <xs:simpleType>
      <xs:restriction base="xs:string">
        <xs:minLength value="10"/>
        <xs:maxLength value="14"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:attribute>
  <xs:attribute name="Quantity" type="moneyType" xdb:SQLName="QUANTITY"/>
  <xs:attribute name="UnitPrice" type="quantityType" xdb:SQLName="UNITPRICE"/>
</xs:complexType>
<xs:simpleType name="ReferenceType">
  <xs:restriction base="xs:string">
    <xs:minLength value="18"/>
    <xs:maxLength value="30"/>
  </xs:restriction>
</xs:simpleType>
<xs:complexType name="ActionsType" xdb:SQLType="ACTIONS_T">
  <xs:sequence>
    <xs:element name="Action" maxOccurs="4" xdb:SQLName="ACTION" xdb:SQLCollType="ACTION_V">
      <xs:complexType xdb:SQLType="actioN_t">
        <xs:sequence>
          <xs:element name="User" type="UserType" xdb:SQLName="ACTIONED_BY"/>
          <xs:element name="Date" type="DateType" minOccurs="0" xdb:SQLName="DATE_ACTIONED"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="RejectionType" xdb:SQLType="REJECTION_T">
  <xs:all>
    <xs:element name="User" type="UserType" minOccurs="0" xdb:SQLName="REJECTED_BY"/>
    <xs:element name="Date" type="DateType" minOccurs="0" xdb:SQLName="DATE_REJECTED"/>
    <xs:element name="Comments" type="CommentsType" minOccurs="0" xdb:SQLName="REASON_REJECTED"/>
```

```
  </xs:all>
</xs:complexType>
<xs:complexType name="ShippingInstructionsType" xdb:SQLType="SHIPPING_INSTRUCTIONS_T">
  <xs:sequence>
    <xs:element name="name" type="NameType" minOccurs="0" xdb:SQLName="SHIP_TO_NAME"/>
    <xs:element name="address" type="AddressType" minOccurs="0" xdb:SQLName="SHIP_TO_ADDRESS"/>
    <xs:element name="telephone" type="TelephoneType" minOccurs="0" xdb:SQLName="SHIP_TO_PHONE"/>
  </xs:sequence>
</xs:complexType>
<xs:simpleType name="moneyType">
  <xs:restriction base="xs:decimal">
    <xs:fractionDigits value="2"/>
    <xs:totalDigits value="12"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="quantityType">
  <xs:restriction base="xs:decimal">
    <xs:fractionDigits value="4"/>
    <xs:totalDigits value="8"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="UserType">
  <xs:restriction base="xs:string">
    <xs:minLength value="0"/>
    <xs:maxLength value="10"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="RequestorType">
  <xs:restriction base="xs:string">
    <xs:minLength value="0"/>
    <xs:maxLength value="128"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="CostCenterType">
  <xs:restriction base="xs:string">
    <xs:minLength value="1"/>
    <xs:maxLength value="4"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="VendorType">
  <xs:restriction base="xs:string">
    <xs:minLength value="0"/>
    <xs:maxLength value="20"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="PurchaseOrderNumberType">
  <xs:restriction base="xs:integer"/>
</xs:simpleType>
<xs:simpleType name="SpecialInstructionsType">
  <xs:restriction base="xs:string">
    <xs:minLength value="0"/>
    <xs:maxLength value="2048"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="NameType">
  <xs:restriction base="xs:string">
    <xs:minLength value="1"/>
    <xs:maxLength value="20"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="AddressType">
  <xs:restriction base="xs:string">
    <xs:minLength value="1"/>
    <xs:maxLength value="256"/>
  </xs:restriction>
</xs:simpleType>
```

```
  <xs:simpleType name="TelephoneType">
    <xs:restriction base="xs:string">
      <xs:minLength value="1"/>
      <xs:maxLength value="24"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="DateType">
    <xs:restriction base="xs:date"/>
  </xs:simpleType>
  <xs:simpleType name="CommentsType">
    <xs:restriction base="xs:string">
      <xs:minLength value="1"/>
      <xs:maxLength value="2048"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="DescriptionType">
    <xs:restriction base="xs:string">
      <xs:minLength value="1"/>
      <xs:maxLength value="256"/>
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

Example A–2 is the complete listing of the revised annotated XML schema presented in Example 10–1 on page 10-2. Text that is in **bold face** is additional or significantly different from that in the schema of Example A–1.

Example A–2 Revised Purchase-Order XML Schema

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:xdb="http://xmlns.oracle.com/xdb"
           version="1.0">
  <xs:element
    name="PurchaseOrder" type="PurchaseOrderType"
    xdb:defaultTable="PURCHASEORDER"
    xdb:columnProps=
      "CONSTRAINT purchaseorder_pkey PRIMARY KEY (XMLDATA.reference),
       CONSTRAINT valid_email_address FOREIGN KEY (XMLDATA.userid)
         REFERENCES hr.employees (EMAIL)"
    xdb:tableProps=
      "VARRAY XMLDATA.ACTIONS.ACTION STORE AS TABLE ACTION_TABLE
        ((CONSTRAINT action_pkey PRIMARY KEY (NESTED_TABLE_ID, SYS_NC_ARRAY_INDEX$)))
       VARRAY XMLDATA.LINEITEMS.LINEITEM STORE AS TABLE LINEITEM_TABLE
        ((constraint LINEITEM_PKEY primary key (NESTED_TABLE_ID, SYS_NC_ARRAY_INDEX$)))
       lob (XMLDATA.NOTES) STORE AS (ENABLE STORAGE IN ROW STORAGE(INITIAL 4K NEXT 32K))"/>
  <xs:complexType name="PurchaseOrderType" xdb:SQLType="PURCHASEORDER_T">
    <xs:sequence>
      <xs:element name="Actions" type="ActionsType" xdb:SQLName="ACTIONS"/>
      <xs:element name="Reject" type="RejectionType" minOccurs="0" xdb:SQLName="REJECTION"/>
      <xs:element name="Requestor" type="RequestorType" xdb:SQLName="REQUESTOR"/>
      <xs:element name="User" type="UserType" xdb:SQLName="USERID"/>
      <xs:element name="CostCenter" type="CostCenterType" xdb:SQLName="COST_CENTER"/>
      <xs:element name="BillingAddress" type="AddressType" minOccurs="0"
                  xdb:SQLName="BILLING_ADDRESS"/>
      <xs:element name="ShippingInstructions" type="ShippingInstructionsType"
                  xdb:SQLName="SHIPPING_INSTRUCTIONS"/>
      <xs:element name="SpecialInstructions" type="SpecialInstructionsType"
                  xdb:SQLName="SPECIAL_INSTRUCTIONS"/>
      <xs:element name="LineItems" type="LineItemsType" xdb:SQLName="LINEITEMS"/>
      <xs:element name="Notes" type="NotesType" minOccurs="0" xdb:SQLType="CLOB"
                  xdb:SQLName="NOTES"/>
    </xs:sequence>
    <xs:attribute name="Reference" type="ReferenceType" use="required" xdb:SQLName="REFERENCE"/>
    <xs:attribute name="DateCreated" type="xs:dateTime" use="required"
                  xdb:SQLType="TIMESTAMP WITH TIME ZONE"/>
  </xs:complexType>
```

```
<xs:complexType name="LineItemsType" xdb:SQLType="LINEITEMS_T">
  <xs:sequence>
    <xs:element name="LineItem" type="LineItemType" maxOccurs="unbounded" xdb:SQLName="LINEITEM"
                xdb:SQLCollType="LINEITEM_V"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="LineItemType" xdb:SQLType="LINEITEM_T">
  <xs:sequence>
    <xs:element name="Part" type="PartType" xdb:SQLName="PART"/>
    <xs:element name="Quantity" type="quantityType"/>
  </xs:sequence>
  <xs:attribute name="ItemNumber" type="xs:integer" xdb:SQLName="ITEMNUMBER"
                xdb:SQLType="NUMBER"/>
</xs:complexType>
<xs:complexType name="PartType" xdb:SQLType="PART_T">
  <xs:simpleContent>
    <xs:extension base="UPCCodeType">
      <xs:attribute name="Description" type="DescriptionType" use="required"
                    xdb:SQLName="DESCRIPTION"/>
      <xs:attribute name="UnitCost" type="moneyType" use="required"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
<xs:simpleType name="ReferenceType">
  <xs:restriction base="xs:string">
    <xs:minLength value="18"/>
    <xs:maxLength value="30"/>
  </xs:restriction>
</xs:simpleType>
<xs:complexType name="ActionsType" xdb:SQLType="ACTIONS_T">
  <xs:sequence>
    <xs:element name="Action" maxOccurs="4" xdb:SQLName="ACTION" xdb:SQLCollType="ACTION_V">
      <xs:complexType xdb:SQLType="ACTION_T">
        <xs:sequence>
          <xs:element name="User" type="UserType" xdb:SQLName="ACTIONED_BY"/>
          <xs:element name="Date" type="DateType" minOccurs="0" xdb:SQLName="DATE_ACTIONED"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="RejectionType" xdb:SQLType="REJECTION_T">
  <xs:all>
    <xs:element name="User" type="UserType" minOccurs="0" xdb:SQLName="REJECTED_BY"/>
    <xs:element name="Date" type="DateType" minOccurs="0" xdb:SQLName="DATE_REJECTED"/>
    <xs:element name="Comments" type="CommentsType" minOccurs="0" xdb:SQLName="REASON_REJECTED"/>
  </xs:all>
</xs:complexType>
<xs:complexType name="ShippingInstructionsType" xdb:SQLType="SHIPPING_INSTRUCTIONS_T">
  <xs:sequence>
    <xs:element name="name" type="NameType" minOccurs="0" xdb:SQLName="SHIP_TO_NAME"/>
    <xs:choice>
      <xs:element name="address" type="AddressType" minOccurs="0"/>
      <xs:element name="fullAddress" type="FullAddressType" minOccurs="0"
                  xdb:SQLName="SHIP_TO_ADDRESS"/>
    </xs:choice>
    <xs:element name="telephone" type="TelephoneType" minOccurs="0" xdb:SQLName="SHIP_TO_PHONE"/>
  </xs:sequence>
</xs:complexType>
<xs:simpleType name="moneyType">
  <xs:restriction base="xs:decimal">
    <xs:fractionDigits value="2"/>
    <xs:totalDigits value="12"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="quantityType">
```

```
  <xs:restriction base="xs:decimal">
    <xs:fractionDigits value="4"/>
    <xs:totalDigits value="8"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="UserType">
  <xs:restriction base="xs:string">
    <xs:minLength value="0"/>
    <xs:maxLength value="10"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="RequestorType">
  <xs:restriction base="xs:string">
    <xs:minLength value="0"/>
    <xs:maxLength value="128"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="CostCenterType">
  <xs:restriction base="xs:string">
    <xs:minLength value="1"/>
    <xs:maxLength value="4"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="VendorType">
  <xs:restriction base="xs:string">
    <xs:minLength value="0"/>
    <xs:maxLength value="20"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="PurchaseOrderNumberType">
  <xs:restriction base="xs:integer"/>
</xs:simpleType>
<xs:simpleType name="SpecialInstructionsType">
  <xs:restriction base="xs:string">
    <xs:minLength value="0"/>
    <xs:maxLength value="2048"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="NameType">
  <xs:restriction base="xs:string">
    <xs:minLength value="1"/>
    <xs:maxLength value="20"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="FullAddressType">
  <xs:restriction base="xs:string">
    <xs:minLength value="1"/>
    <xs:maxLength value="256"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="TelephoneType">
  <xs:restriction base="xs:string">
    <xs:minLength value="1"/>
    <xs:maxLength value="24"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="DateType">
        <xs:restriction base="xs:date"/>
</xs:simpleType>
<xs:simpleType name="CommentsType">
  <xs:restriction base="xs:string">
    <xs:minLength value="1"/>
    <xs:maxLength value="2048"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="DescriptionType">
```

```
    <xs:restriction base="xs:string">
      <xs:minLength value="1"/>
      <xs:maxLength value="256"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:complexType name="AddressType" xdb:SQLType="ADDRESS_T">
    <xs:sequence>
      <xs:element name="StreetLine1" type="StreetType"/>
      <xs:element name="StreetLine2" type="StreetType" minOccurs="0"/>
      <xs:element name="City" type="CityType"/>
      <xs:choice>
        <xs:sequence>
          <xs:element name="State" type="StateType"/>
          <xs:element name="ZipCode" type="ZipCodeType"/>
        </xs:sequence>
        <xs:sequence>
          <xs:element name="Province" type="ProvinceType"/>
          <xs:element name="PostCode" type="PostCodeType"/>
        </xs:sequence>
        <xs:sequence>
          <xs:element name="County" type="CountyType"/>
          <xs:element name="Postcode" type="PostCodeType"/>
        </xs:sequence>
      </xs:choice>
      <xs:element name="Country" type="CountryType"/>
    </xs:sequence>
  </xs:complexType>
  <xs:simpleType name="StreetType">
    <xs:restriction base="xs:string">
      <xs:minLength value="1"/>
      <xs:maxLength value="128"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="CityType">
    <xs:restriction base="xs:string">
      <xs:minLength value="1"/>
      <xs:maxLength value="64"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="StateType">
    <xs:restriction base="xs:string">
      <xs:minLength value="2"/>
      <xs:maxLength value="2"/>
      <xs:enumeration value="AK"/>
      <xs:enumeration value="AL"/>
      <xs:enumeration value="AR"/>
      <xs:enumeration value="AS"/>
      <xs:enumeration value="AZ"/>
      <xs:enumeration value="CA"/>
      <xs:enumeration value="CO"/>
      <xs:enumeration value="CT"/>
      <xs:enumeration value="DC"/>
      <xs:enumeration value="DE"/>
      <xs:enumeration value="FL"/>
      <xs:enumeration value="FM"/>
      <xs:enumeration value="GA"/>
      <xs:enumeration value="GU"/>
      <xs:enumeration value="HI"/>
      <xs:enumeration value="IA"/>
      <xs:enumeration value="ID"/>
      <xs:enumeration value="IL"/>
      <xs:enumeration value="IN"/>
      <xs:enumeration value="KS"/>
      <xs:enumeration value="KY"/>
      <xs:enumeration value="LA"/>
      <xs:enumeration value="MA"/>
```

```
      <xs:enumeration value="MD"/>
      <xs:enumeration value="ME"/>
      <xs:enumeration value="MH"/>
      <xs:enumeration value="MI"/>
      <xs:enumeration value="MN"/>
      <xs:enumeration value="MO"/>
      <xs:enumeration value="MP"/>
      <xs:enumeration value="MQ"/>
      <xs:enumeration value="MS"/>
      <xs:enumeration value="MT"/>
      <xs:enumeration value="NC"/>
      <xs:enumeration value="ND"/>
      <xs:enumeration value="NE"/>
      <xs:enumeration value="NH"/>
      <xs:enumeration value="NJ"/>
      <xs:enumeration value="NM"/>
      <xs:enumeration value="NV"/>
      <xs:enumeration value="NY"/>
      <xs:enumeration value="OH"/>
      <xs:enumeration value="OK"/>
      <xs:enumeration value="OR"/>
      <xs:enumeration value="PA"/>
      <xs:enumeration value="PR"/>
      <xs:enumeration value="PW"/>
      <xs:enumeration value="RI"/>
      <xs:enumeration value="SC"/>
      <xs:enumeration value="SD"/>
      <xs:enumeration value="TN"/>
      <xs:enumeration value="TX"/>
      <xs:enumeration value="UM"/>
      <xs:enumeration value="UT"/>
      <xs:enumeration value="VA"/>
      <xs:enumeration value="VI"/>
      <xs:enumeration value="VT"/>
      <xs:enumeration value="WA"/>
      <xs:enumeration value="WI"/>
      <xs:enumeration value="WV"/>
      <xs:enumeration value="WY"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="ZipCodeType">
    <xs:restriction base="xs:string">
      <xs:pattern value="\d{5}"/>
      <xs:pattern value="\d{5}-\d{4}"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="CountryType">
    <xs:restriction base="xs:string">
      <xs:minLength value="1"/>
      <xs:maxLength value="64"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="CountyType">
    <xs:restriction base="xs:string">
      <xs:minLength value="1"/>
      <xs:maxLength value="32"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="PostCodeType">
    <xs:restriction base="xs:string">
      <xs:minLength value="1"/>
      <xs:maxLength value="12"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="ProvinceType">
    <xs:restriction base="xs:string">
```

```
      <xs:minLength value="2"/>
      <xs:maxLength value="2"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="NotesType">
    <xs:restriction base="xs:string">
      <xs:maxLength value="32767"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="UPCCodeType">
    <xs:restriction base="xs:string">
      <xs:minLength value="11"/>
      <xs:maxLength value="14"/>
      <xs:pattern value="\d{11}"/>
      <xs:pattern value="\d{12}"/>
      <xs:pattern value="\d{13}"/>
      <xs:pattern value="\d{14}"/>
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

XSL Style Sheet Example, PurchaseOrder.xsl

Example A–3 shows XSLT style sheet `PurchaseOrder.xsl`. The example style sheet is used in examples in Chapter 3, "Using Oracle XML DB".

Example A–3 PurchaseOrder.xsl Style Sheet

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:xdb="http://xmlns.oracle.com/xdb"
                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <xsl:template match="/">
    <html>
      <head/>
      <body bgcolor="#003333" text="#FFFFCC" link="#FFCC00"
            vlink="#66CC99" alink="#669999">
        <FONT FACE="Arial, Helvetica, sans-serif">
          <xsl:for-each select="PurchaseOrder"/>
          <xsl:for-each select="PurchaseOrder">
            <center>
              <span style="font-family:Arial; font-weight:bold">
                <FONT COLOR="#FF0000">
                  <B>Purchase Order </B>
                </FONT>
              </span>
            </center>
            <br/>
            <center>
              <xsl:for-each select="Reference">
                <span style="font-family:Arial; font-weight:bold">
                  <xsl:apply-templates/>
                </span>
              </xsl:for-each>
            </center>
          </xsl:for-each>
          <P>
            <xsl:for-each select="PurchaseOrder">
              <br/>
            </xsl:for-each>
```

```
  <P/>
  <P>
    <xsl:for-each select="PurchaseOrder">
      <br/>
    </xsl:for-each>
  </P>
</P>
<xsl:for-each select="PurchaseOrder"/>
<xsl:for-each select="PurchaseOrder">
  <table border="0" width="100%" BGCOLOR="#000000">
    <tbody>
      <tr>
        <td WIDTH="296">
          <P>
            <B>
              <FONT SIZE="+1" COLOR="#FF0000"
               FACE="Arial, Helvetica, sans-serif">Internal
              </FONT>
            </B>
          </P>
          <table border="0" width="98%" BGCOLOR="#000099">
            <tbody>
              <tr>
                <td WIDTH="49%">
                  <B>
                    <FONT COLOR="#FFFF00">Actions</FONT>
                  </B>
                </td>
                <td WIDTH="51%">
                  <xsl:for-each select="Actions">
                    <xsl:for-each select="Action">
                      <table border="1" WIDTH="143">
                        <xsl:if test="position()=1">
                          <thead>
                            <tr>
                              <td HEIGHT="21">
                                <FONT
                                 COLOR="#FFFF00">User</FONT>
                              </td>
                              <td HEIGHT="21">
                                <FONT
                                 COLOR="#FFFF00">Date</FONT>
                              </td>
                            </tr>
                          </thead>
                        </xsl:if>
                        <tbody>
                          <tr>
                            <td>
                              <xsl:for-each select="User">
                                <xsl:apply-templates/>
                              </xsl:for-each>
                            </td>
                            <td>
                              <xsl:for-each select="Date">
                                <xsl:apply-templates/>
                              </xsl:for-each>
                            </td>
                          </tr>
                        </tbody>
```

```
                  </table>
                </xsl:for-each>
              </xsl:for-each>
            </td>
          </tr>
          <tr>
            <td WIDTH="49%">
              <B>
                <FONT COLOR="#FFFF00">Requestor</FONT>
              </B>
            </td>
            <td WIDTH="51%">
              <xsl:for-each select="Requestor">
                <xsl:apply-templates/>
              </xsl:for-each>
            </td>
          </tr>
          <tr>
            <td WIDTH="49%">
              <B>
                <FONT COLOR="#FFFF00">User</FONT>
              </B>
            </td>
            <td WIDTH="51%">
              <xsl:for-each select="User">
                <xsl:apply-templates/>
              </xsl:for-each>
            </td>
          </tr>
          <tr>
            <td WIDTH="49%">
              <B>
                <FONT COLOR="#FFFF00">Cost Center</FONT>
              </B>
            </td>
            <td WIDTH="51%">
              <xsl:for-each select="CostCenter">
                <xsl:apply-templates/>
              </xsl:for-each>
            </td>
          </tr>
        </tbody>
      </table>
    </td>
    <td width="93"/>
    <td valign="top" WIDTH="340">
      <B>
        <FONT COLOR="#FF0000">
          <FONT SIZE="+1">Ship To</FONT>
        </FONT>
      </B>
      <xsl:for-each select="ShippingInstructions">
        <xsl:if test="position()=1"/>
      </xsl:for-each>
      <xsl:for-each select="ShippingInstructions">
        <xsl:if test="position()=1">
          <table border="0" BGCOLOR="#999900">
            <tbody>
              <tr>
                <td WIDTH="126" HEIGHT="24">
```

```
                    <B>Name</B>
                  </td>
                  <xsl:for-each
                       select="../ShippingInstructions">
                    <td WIDTH="218" HEIGHT="24">
                      <xsl:for-each select="name">
                        <xsl:apply-templates/>
                      </xsl:for-each>
                    </td>
                  </xsl:for-each>
                </tr>
                <tr>
                  <td WIDTH="126" HEIGHT="34">
                    <B>Address</B>
                  </td>
                  <xsl:for-each
                       select="../ShippingInstructions">
                    <td WIDTH="218" HEIGHT="34">
                      <xsl:for-each select="address">
                        <span style="white-space:pre">
                          <xsl:apply-templates/>
                        </span>
                      </xsl:for-each>
                    </td>
                  </xsl:for-each>
                </tr>
                <tr>
                  <td WIDTH="126" HEIGHT="32">
                    <B>Telephone</B>
                  </td>
                  <xsl:for-each
                       select="../ShippingInstructions">
                    <td WIDTH="218" HEIGHT="32">
                      <xsl:for-each select="telephone">
                        <xsl:apply-templates/>
                      </xsl:for-each>
                    </td>
                  </xsl:for-each>
                </tr>
              </tbody>
            </table>
          </xsl:if>
        </xsl:for-each>
      </td>
    </tr>
  </tbody>
</table>
<br/>
<B>
  <FONT COLOR="#FF0000" SIZE="+1">Items:</FONT>
</B>
<br/>
<br/>
<table border="0">
  <xsl:for-each select="LineItems">
    <xsl:for-each select="LineItem">
      <xsl:if test="position()=1">
        <thead>
          <tr bgcolor="#C0C0C0">
            <td>
```

```
            <FONT COLOR="#FF0000">
              <B>ItemNumber</B>
            </FONT>
          </td>
          <td>
            <FONT COLOR="#FF0000">
              <B>Description</B>
            </FONT>
          </td>
          <td>
            <FONT COLOR="#FF0000">
              <B>PartId</B>
            </FONT>
          </td>
          <td>
            <FONT COLOR="#FF0000">
              <B>Quantity</B>
            </FONT>
          </td>
          <td>
            <FONT COLOR="#FF0000">
              <B>Unit Price</B>
            </FONT>
          </td>
          <td>
            <FONT COLOR="#FF0000">
              <B>Total Price</B>
            </FONT>
          </td>
        </tr>
      </thead>
    </xsl:if>
    <tbody>
      <tr bgcolor="#DADADA">
        <td>
          <FONT COLOR="#000000">
            <xsl:for-each select="@ItemNumber">
              <xsl:value-of select="."/>
            </xsl:for-each>
          </FONT>
        </td>
        <td>
          <FONT COLOR="#000000">
            <xsl:for-each select="Description">
              <xsl:apply-templates/>
            </xsl:for-each>
          </FONT>
        </td>
        <td>
          <FONT COLOR="#000000">
            <xsl:for-each select="Part">
              <xsl:for-each select="@Id">
                <xsl:value-of select="."/>
              </xsl:for-each>
            </xsl:for-each>
          </FONT>
        </td>
        <td>
          <FONT COLOR="#000000">
            <xsl:for-each select="Part">
```

```
                                <xsl:for-each select="@Quantity">
                                  <xsl:value-of select="."/>
                                </xsl:for-each>
                              </xsl:for-each>
                            </FONT>
                          </td>
                          <td>
                            <FONT COLOR="#000000">
                              <xsl:for-each select="Part">
                                <xsl:for-each select="@UnitPrice">
                                  <xsl:value-of select="."/>
                                </xsl:for-each>
                              </xsl:for-each>
                            </FONT>
                          </td>
                          <td>
                            <FONT FACE="Arial, Helvetica, sans-serif"
                                  COLOR="#000000">
                              <xsl:for-each select="Part">
                                <xsl:value-of select="@Quantity*@UnitPrice"/>
                              </xsl:for-each>
                            </FONT>
                          </td>
                        </tr>
                      </tbody>
                    </xsl:for-each>
                  </xsl:for-each>
                </table>
              </xsl:for-each>
            </FONT>
          </body>
        </html>
      </xsl:template>
    </xsl:stylesheet>
```

Loading XML Data using C (OCI)

Example A–4 is a C program that inserts XML data into an `XMLType` table. It is partly listed in Chapter 3, "Using Oracle XML DB", "Loading XML Content using C" on page 3-6.

Example A–4 Inserting XML Data into an XMLType Table using C

```
#include "stdio.h"
#include <xml.h>
#include <stdlib.h>
#include <string.h>
#include <ocixmldb.h>
OCIEnv *envhp;
OCIError *errhp;
OCISvcCtx *svchp;
OCIStmt *stmthp;
OCIServer *srvhp;
OCIDuration dur;
OCISession *sesshp;
oratext *username = "QUINE";
oratext *password = "************";      /* Replace with real password */
oratext *filename = "AMCEWEN-20021009123336171PDT.xml";
oratext *schemaloc = "http://localhost:8080/source/schemas/poSource/xsd/purchaseOrder.xsd";
```

```
/* Execute a SQL statement that binds XML data */
sword exec_bind_xml(OCISvcCtx *svchp, OCIError *errhp, OCIStmt *stmthp,
                    void *xml,        OCIType *xmltdo, OraText *sqlstmt)
{
  OCIBind *bndhp1 = (OCIBind *) 0;
  sword  status = 0;
  OCIInd ind = OCI_IND_NOTNULL;
  OCIInd *indp = &ind;
  if(status = OCIStmtPrepare(stmthp, errhp, (OraText *)sqlstmt,
                             (ub4)strlen((const char *)sqlstmt),
                             (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT))
    return OCI_ERROR;
  if(status = OCIBindByPos(stmthp, &bndhp1, errhp, (ub4) 1, (dvoid *) 0,
                           (sb4) 0, SQLT_NTY, (dvoid *) 0, (ub2 *)0,
                           (ub2 *)0, (ub4) 0, (ub4 *) 0, (ub4) OCI_DEFAULT))
    return OCI_ERROR;
  if(status = OCIBindObject(bndhp1, errhp, (CONST OCIType *) xmltdo,
                            (dvoid **) &xml, (ub4 *) 0,
                            (dvoid **) &indp, (ub4 *) 0))
    return OCI_ERROR;
  if(status = OCIStmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
                             (CONST OCISnapshot*) 0, (OCISnapshot*) 0,
                             (ub4) OCI_DEFAULT))
    return OCI_ERROR;
  return OCI_SUCCESS;
}

/* Initialize OCI handles, and connect  */
sword init_oci_connect()
{
  sword status;
  if (OCIEnvCreate((OCIEnv **) &(envhp), (ub4) OCI_OBJECT,
                   (dvoid *) 0, (dvoid * (*)(dvoid *,size_t)) 0,
                   (dvoid * (*)(dvoid *, dvoid *, size_t)) 0,
                   (void (*)(dvoid *, dvoid *)) 0, (size_t) 0, (dvoid **) 0))
    {
      printf("FAILED: OCIEnvCreate()\n");
      return OCI_ERROR;
    }
  /* Allocate error handle */
  if (OCIHandleAlloc((dvoid *) envhp, (dvoid **) &(errhp),
                     (ub4) OCI_HTYPE_ERROR, (size_t) 0, (dvoid **) 0))
    {
      printf("FAILED: OCIHandleAlloc() on errhp\n");
      return OCI_ERROR;
    }
  /* Allocate server handle */
  if (status = OCIHandleAlloc((dvoid *) envhp, (dvoid **) &srvhp,
                              (ub4) OCI_HTYPE_SERVER, (size_t) 0, (dvoid **) 0))
    {
      printf("FAILED: OCIHandleAlloc() on srvhp\n");
      return OCI_ERROR;
    }
  /* Allocate service context handle */
  if (status = OCIHandleAlloc((dvoid *) envhp,
                              (dvoid **) &(svchp), (ub4) OCI_HTYPE_SVCCTX,
                              (size_t) 0, (dvoid **) 0))
    {
      printf("FAILED: OCIHandleAlloc() on svchp\n");
      return OCI_ERROR;
    }
  /* Allocate session handle */
  if (status = OCIHandleAlloc((dvoid *) envhp, (dvoid **) &sesshp ,
                              (ub4) OCI_HTYPE_SESSION, (size_t) 0, (dvoid **) 0))
    {
```

```
      printf("FAILED: OCIHandleAlloc() on sesshp\n");
      return OCI_ERROR;
    }
  /* Allocate statement handle */
  if (OCIHandleAlloc((dvoid *)envhp, (dvoid **) &stmthp,
                     (ub4)OCI_HTYPE_STMT, (CONST size_t) 0, (dvoid **) 0))
    {
      printf("FAILED: OCIHandleAlloc() on stmthp\n");
      return status;
    }
  if (status = OCIServerAttach((OCIServer *) srvhp, (OCIError *) errhp,
                               (CONST oratext *)"", 0, (ub4) OCI_DEFAULT))
    {
      printf("FAILED: OCIServerAttach() on srvhp\n");
      return OCI_ERROR;
    }
  /* Set server attribute to service context */
  if (status = OCIAttrSet((dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX,
                          (dvoid *) srvhp, (ub4) 0, (ub4) OCI_ATTR_SERVER,
                          (OCIError *) errhp))
    {
      printf("FAILED: OCIAttrSet() on svchp\n");
      return OCI_ERROR;
    }
  /* Set user attribute to session */
  if (status = OCIAttrSet((dvoid *)sesshp, (ub4) OCI_HTYPE_SESSION,
                          (dvoid *)username,
                          (ub4) strlen((const char *)username),
                          (ub4) OCI_ATTR_USERNAME, (OCIError *) errhp))
    {
      printf("FAILED: OCIAttrSet() on authp for user\n");
      return OCI_ERROR;
    }
  /* Set password attribute to session */
  if (status = OCIAttrSet((dvoid *) sesshp, (ub4) OCI_HTYPE_SESSION,
                          (dvoid *)password,
                          (ub4) strlen((const char *)password),
                          (ub4) OCI_ATTR_PASSWORD, (OCIError *) errhp))
    {
      printf("FAILED: OCIAttrSet() on authp for password\n");
      return OCI_ERROR;
    }
  /* Begin a session  */
  if (status = OCISessionBegin((OCISvcCtx *) svchp,
                               (OCIError *) errhp,
                               (OCISession *) sesshp, (ub4) OCI_CRED_RDBMS,
                               (ub4) OCI_STMT_CACHE))
    {
      printf("FAILED: OCISessionBegin(). Make sure database is up and the username/password is valid. \n");
      return OCI_ERROR;
    }
  /* Set session attribute to service context */
  if (status = OCIAttrSet((dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX,
                          (dvoid *)sesshp, (ub4) 0, (ub4) OCI_ATTR_SESSION,
                          (OCIError *) errhp))
    {
      printf("FAILED: OCIAttrSet() on svchp\n");
      return OCI_ERROR;
    }
}

/* Free OCI handles, and disconnect                        */
void free_oci()
{
  sword  status = 0;
```

```
  /* End the session */
  if (status = OCISessionEnd((OCISvcCtx *)svchp, (OCIError *)errhp,
                             (OCISession *)sesshp, (ub4) OCI_DEFAULT))
    {
      if (envhp)
        OCIHandleFree((dvoid *)envhp, OCI_HTYPE_ENV);
      return;
    }
  /* Detach from the server */
  if (status = OCIServerDetach((OCIServer *)srvhp, (OCIError *)errhp,
                               (ub4)OCI_DEFAULT))
    {
      if (envhp)
        OCIHandleFree((dvoid *)envhp, OCI_HTYPE_ENV);
      return;
    }
  /* Free the handles */
  if (stmthp) OCIHandleFree((dvoid *)stmthp, (ub4) OCI_HTYPE_STMT);
  if (sesshp) OCIHandleFree((dvoid *)sesshp, (ub4) OCI_HTYPE_SESSION);
  if (svchp) OCIHandleFree((dvoid *)svchp, (ub4) OCI_HTYPE_SVCCTX);
  if (srvhp) OCIHandleFree((dvoid *)srvhp, (ub4) OCI_HTYPE_SERVER);
  if (errhp) OCIHandleFree((dvoid *)errhp, (ub4) OCI_HTYPE_ERROR);
  if (envhp) OCIHandleFree((dvoid *)envhp, (ub4) OCI_HTYPE_ENV);
  return;
}

void main()
{
  OCIType *xmltdo;
  xmldocnode  *doc;
  ocixmldbparam params[1];
  xmlerr       err;
  xmlctx  *xctx;
  oratext *ins_stmt;
  sword    status;
  xmlnode *root;
  oratext buf[10000];

  /* Initialize envhp, svchp, errhp, dur, stmthp */
  init_oci_connect();

  /* Get an XML context */
  params[0].name_ocixmldbparam = XCTXINIT_OCIDUR;
  params[0].value_ocixmldbparam = &dur;
  xctx = OCIXmlDbInitXmlCtx(envhp, svchp, errhp, params, 1);
  if (!(doc = XmlLoadDom(xctx, &err, "file", filename,
                         "schema_location", schemaloc, NULL)))
    {
      printf("Parse failed.\n");
      return;
    }
  else
    printf("Parse succeeded.\n");
  root = XmlDomGetDocElem(xctx, doc);
  printf("The xml document is :\n");
  XmlSaveDom(xctx, &err, (xmlnode *)doc, "buffer", buf, "buffer_length", 10000, NULL);
  printf("%s\n", buf);

  /* Insert the document into my_table */
  ins_stmt = (oratext *)"insert into purchaseorder values (:1)";
  status = OCITypeByName(envhp, errhp, svchp, (const text *) "SYS",
                         (ub4) strlen((const char *)"SYS"), (const text *) "XMLTYPE",
                         (ub4) strlen((const char *)"XMLTYPE"), (CONST text *) 0,
                         (ub4) 0, OCI_DURATION_SESSION, OCI_TYPEGET_HEADER,
                         (OCIType **) &xmltdo);
  if (status == OCI_SUCCESS)
```

```
    {
      status = exec_bind_xml(svchp, errhp, stmthp, (void *)doc,
                             xmltdo, ins_stmt);
    }
  if (status == OCI_SUCCESS)
    printf ("Insert successful\n");
  else
    printf ("Insert failed\n");

  /* Free XML instances */
  if (doc) XmlFreeDocument((xmlctx *)xctx, (xmldocnode *)doc);

  /* Free XML CTX */
  OCIXmlDbFreeXmlCtx(xctx);
  free_oci();
}
```

Initializing and Terminating an XML Context (OCI)

Example A–5 shows how to use OCI functions `OCIXmlDbInitXmlCtx()` and `OCIXmlDbFreeXmlCtx()` to initialize and terminate the XML context. It constructs an XML document using the C DOM API and saves it to the database.

Example A–5 is partially listed in Chapter 16, "Using the C API for XML", "Initializing and Terminating an XML Context" on page 16-3. It assumes that the following SQL code has first been executed to create table `my_table` in database schema `CAPIUSER`:

```
CONNECT CAPIUSER
Enter password: <password>

Connected.

CREATE TABLE my_table OF XMLType;
```

Example A–5 Using OCIXmlDbInitXmlCtx() and OCIXmlDbFreeXmlCtx()

```
#ifndef S_ORACLE
#include <s.h>
#endif
#ifndef ORATYPES_ORACLE
#include <oratypes.h>
#endif
#ifndef XML_ORACLE
#include <xml.h>
#endif
#ifndef OCIXML_ORACLE
#include <ocixml.h>
#endif
#ifndef OCI_ORACLE
#include <oci.h>
#endif
#include <string.h>

typedef struct test_ctx {
        OCIEnv *envhp;
        OCIError *errhp;
        OCISvcCtx *svchp;
        OCIStmt *stmthp;
        OCIServer *srvhp;
        OCIDuration dur;
        OCISession *sesshp;
        oratext *username;
```

```
        oratext *password;
} test_ctx;

/* Helper function 1: execute a sql statement which binds xml data */
STATICF sword exec_bind_xml(OCISvcCtx *svchp,
                    OCIError *errhp,
                    OCIStmt *stmthp,
                    void *xml,
                    OCIType *xmltdo,
                    OraText *sqlstmt);

/* Helper function 2: Initialize OCI handles and connect */
STATICF sword init_oci_handles(test_ctx *ctx);

/* Helper function 3: Free OCI handles and disconnect */
STATICF sword free_oci_handles(test_ctx *ctx);

void main()
{
  test_ctx temp_ctx;
  test_ctx *ctx = &temp_ctx;
  OCIType *xmltdo = (OCIType *) 0;
  xmldocnode *doc = (xmldocnode *)0;
  ocixmldbparam params[1];
  xmlnode *quux, *foo, *foo_data, *top;
  xmlerr err;
  sword status = 0;
  xmlctx *xctx;

  oratext ins_stmt[] = "insert into my_table values (:1)";
  oratext tlpxml_test_sch[] = "<TOP/>";
  ctx->username = (oratext *)"CAPIUSER";
  ctx->password = (oratext *)"*************";   /* Replace with real password */

  /* Initialize envhp, svchp, errhp, dur, stmthp */
  init_oci_handles(ctx);

  /* Get an xml context */
  params[0].name_ocixmldbparam = XCTXINIT_OCIDUR;
  params[0].value_ocixmldbparam = &ctx->dur;
  xctx = OCIXmlDbInitXmlCtx(ctx->envhp, ctx->svchp, ctx->errhp, params, 1);

  /* Start processing - first, check that this DOM supports XML 1.0 */
  printf("\n\nSupports XML 1.0? : %s\n",
         XmlHasFeature(xctx, (oratext *) "xml", (oratext *) "1.0") ?
         "YES" : "NO");

  /* Parse a document */
  if (!(doc = XmlLoadDom(xctx, &err, "buffer", tlpxml_test_sch,
                         "buffer_length", sizeof(tlpxml_test_sch)-1,
                         "validate", TRUE, NULL)))
  {
    printf("Parse failed, code %d\n", err);
  }
  else
  {
    /* Get the document element */
    top = (xmlnode *)XmlDomGetDocElem(xctx, doc);

    /* Print out the top element */
```

```
    printf("\n\nOriginal top element is :\n");
    XmlSaveDom(xctx, &err, top, "stdio", stdout, NULL);

    /* Print out the document-note that the changes are reflected here */
    printf("\n\nOriginal document is :\n");
    XmlSaveDom(xctx, &err, (xmlnode *)doc, "stdio", stdout, NULL);

    /* Create some elements and add them to the document */
    quux = (xmlnode *) XmlDomCreateElem(xctx ,doc, (oratext *) "QUUX");
    foo = (xmlnode *) XmlDomCreateElem(xctx, doc, (oratext *) "FOO");
    foo_data = (xmlnode *) XmlDomCreateText(xctx, doc, (oratext *) "data");
    foo_data = XmlDomAppendChild(xctx, (xmlnode *) foo, (xmlnode *) foo_data);
    foo = XmlDomAppendChild(xctx, quux, foo);
    quux = XmlDomAppendChild(xctx, top, quux);

    /* Print out the top element */
    printf("\n\nNow the top element is :\n");
    XmlSaveDom(xctx, &err, top, "stdio", stdout, NULL);

    /* Print out the document. Note that the changes are reflected here */
    printf("\n\nNow the document is :\n");
    XmlSaveDom(xctx, &err, (xmlnode *)doc, "stdio", stdout, NULL);

    /* Insert the document into my_table */
    status = OCITypeByName(ctx->envhp, ctx->errhp, ctx->svchp,
                           (const text *) "SYS", (ub4) strlen((char *)"SYS"),
                           (const text *) "XMLTYPE",
                           (ub4) strlen((char *)"XMLTYPE"), (CONST text *) 0,
                           (ub4) 0, OCI_DURATION_SESSION, OCI_TYPEGET_HEADER,
                           (OCIType **) &xmltdo);
    if (status == OCI_SUCCESS)
    {
      exec_bind_xml(ctx->svchp, ctx->errhp, ctx->stmthp, (void *)doc, xmltdo,
                    ins_stmt);
    }
  }
  /* Free xml ctx */
  OCIXmlDbFreeXmlCtx(xctx);

  /* Free envhp, svchp, errhp, stmthp */
  free_oci_handles(ctx);
}

/* Helper function 1: execute a SQL statement that binds xml data */
STATICF sword exec_bind_xml(OCISvcCtx *svchp,
                    OCIError *errhp,
                    OCIStmt *stmthp,
                    void *xml,
                    OCIType *xmltdo,
                    OraText *sqlstmt)
{
  OCIBind *bndhp1 = (OCIBind *) 0;
  sword status = 0;
  OCIInd ind = OCI_IND_NOTNULL;
  OCIInd *indp = &ind;
  if(status = OCIStmtPrepare(stmthp, errhp, (OraText *)sqlstmt,
                             (ub4)strlen((char *)sqlstmt),
                             (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT)) {
    printf("Failed OCIStmtPrepare\n");
    return OCI_ERROR;
```

```
  }
  if(status = OCIBindByPos(stmthp, &bndhp1, errhp, (ub4) 1, (dvoid *) 0,
                           (sb4) 0, SQLT_NTY, (dvoid *) 0, (ub2 *)0,
                           (ub2 *)0, (ub4) 0, (ub4 *) 0, (ub4) OCI_DEFAULT)) {
    printf("Failed OCIBindByPos\n");
    return OCI_ERROR;
  }
  if(status = OCIBindObject(bndhp1, errhp, (CONST OCIType *) xmltdo, (dvoid **)
                            &xml,
                            (ub4 *) 0, (dvoid **) &indp, (ub4 *) 0)) {
    printf("Failed OCIBindObject\n");
    return OCI_ERROR;
  }
  if(status = OCIStmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
                             (CONST OCISnapshot*) 0, (OCISnapshot*) 0,
                             (ub4) OCI_DEFAULT)) {
    printf("Failed OCIStmtExecute\n");
    return OCI_ERROR;
  }
  return OCI_SUCCESS;
}

/* Helper function 2: Initialize OCI handles and connect */
STATICF sword init_oci_handles(test_ctx *ctx)
{
  sword status;
  ctx->dur = OCI_DURATION_SESSION;
  if (OCIEnvCreate((OCIEnv **) &(ctx->envhp), (ub4) OCI_OBJECT,
                   (dvoid *) 0, (dvoid * (*)(dvoid *,size_t)) 0,
                   (dvoid * (*)(dvoid *, dvoid *, size_t)) 0,
                   (void (*)(dvoid *, dvoid *)) 0, (size_t) 0, (dvoid **) 0))
  {
    printf("FAILED: OCIEnvCreate()\n");
    return OCI_ERROR;
  }
  /* Allocate error handle */
  if (OCIHandleAlloc((dvoid *) ctx->envhp, (dvoid **) &(ctx->errhp),
                     (ub4) OCI_HTYPE_ERROR, (size_t) 0, (dvoid **) 0))
  {
    printf("FAILED: OCIHandleAlloc() on errhp\n");
    return OCI_ERROR;
  }
  /* Allocate server handle */
  if (status = OCIHandleAlloc((dvoid *) ctx->envhp, (dvoid **) &ctx->srvhp,
                           (ub4) OCI_HTYPE_SERVER, (size_t) 0, (dvoid **) 0))
  {
    printf("FAILED: OCIHandleAlloc() on srvhp\n");
    return OCI_ERROR;
  }
  /* Allocate service context handle */
  if (status = OCIHandleAlloc((dvoid *) ctx->envhp,
                              (dvoid **) &(ctx->svchp), (ub4) OCI_HTYPE_SVCCTX,
                              (size_t) 0, (dvoid **) 0))
  {
    printf("FAILED: OCIHandleAlloc() on svchp\n");
    return OCI_ERROR;
  }
  /* Allocate session handle */
  if (status = OCIHandleAlloc((dvoid *) ctx->envhp, (dvoid **) &ctx->sesshp ,
                           (ub4) OCI_HTYPE_SESSION, (size_t) 0, (dvoid **) 0))
```

```
{
  printf("FAILED: OCIHandleAlloc() on sesshp\n");
  return OCI_ERROR;
}
/* Allocate statement handle */
if (OCIHandleAlloc((dvoid *)ctx->envhp, (dvoid **) &ctx->stmthp,
                   (ub4)OCI_HTYPE_STMT, (CONST size_t) 0, (dvoid **) 0))
{
  printf("FAILED: OCIHandleAlloc() on stmthp\n");
  return status;
}
if (status = OCIServerAttach((OCIServer *) ctx->srvhp, (OCIError *) ctx->errhp,
                             (CONST oratext *)"", 0, (ub4) OCI_DEFAULT))
{
  printf("FAILED: OCIServerAttach() on srvhp\n");
  return OCI_ERROR;
}
/* Set server attribute to service context */
if (status = OCIAttrSet((dvoid *) ctx->svchp, (ub4) OCI_HTYPE_SVCCTX,
                        (dvoid *) ctx->srvhp, (ub4) 0, (ub4) OCI_ATTR_SERVER,
                        (OCIError *) ctx->errhp))
{
  printf("FAILED: OCIAttrSet() on svchp\n");
  return OCI_ERROR;
}
/* Set user attribute to session */
if (status = OCIAttrSet((dvoid *)ctx->sesshp, (ub4) OCI_HTYPE_SESSION,
                        (dvoid *)ctx->username,
                        (ub4) strlen((char *)ctx->username),
                        (ub4) OCI_ATTR_USERNAME, (OCIError *) ctx->errhp))
{
  printf("FAILED: OCIAttrSet() on authp for user\n");
  return OCI_ERROR;
}
/* Set password attribute to session */
if (status = OCIAttrSet((dvoid *) ctx->sesshp, (ub4) OCI_HTYPE_SESSION,
                        (dvoid *)ctx->password,
                        (ub4) strlen((char *)ctx->password),
                        (ub4) OCI_ATTR_PASSWORD, (OCIError *) ctx->errhp))
{
  printf("FAILED: OCIAttrSet() on authp for password\n");
  return OCI_ERROR;
}
/* Begin a session  */
if (status = OCISessionBegin((OCISvcCtx *) ctx->svchp,
                             (OCIError *) ctx->errhp,
                             (OCISession *) ctx->sesshp, (ub4) OCI_CRED_RDBMS,
                             (ub4) OCI_STMT_CACHE))
{
  printf("FAILED: OCISessionBegin(). Make sure database is up and the \
          username/password is valid. \n");
  return OCI_ERROR;
}
/* Set session attribute to service context */
if (status = OCIAttrSet((dvoid *) ctx->svchp, (ub4) OCI_HTYPE_SVCCTX,
                        (dvoid *)ctx->sesshp, (ub4) 0, (ub4) OCI_ATTR_SESSION,
                        (OCIError *) ctx->errhp))
{
  printf("FAILED: OCIAttrSet() on svchp\n");
  return OCI_ERROR;
```

```
  }
  return status;
}

/* Helper function 3: Free OCI handles and disconnect */
STATICF sword free_oci_handles(test_ctx *ctx)
{
  sword  status = 0;
  /* End the session */
  if (status = OCISessionEnd((OCISvcCtx *)ctx->svchp, (OCIError *)ctx->errhp,
                             (OCISession *)ctx->sesshp, (ub4) OCI_DEFAULT))
  {
    if (ctx->envhp)
      OCIHandleFree((dvoid *)ctx->envhp, OCI_HTYPE_ENV);
    return status;
  }
  /* Detach from the server */
  if (status = OCIServerDetach((OCIServer *)ctx->srvhp, (OCIError *)ctx->errhp,
      (ub4)OCI_DEFAULT))
  {
    if (ctx->envhp)
      OCIHandleFree((dvoid *)ctx->envhp, OCI_HTYPE_ENV);
    return status;
  }
  /* Free the handles */
  if (ctx->stmthp) OCIHandleFree((dvoid *)ctx->stmthp, (ub4) OCI_HTYPE_STMT);
  if (ctx->sesshp) OCIHandleFree((dvoid *)ctx->sesshp, (ub4) OCI_HTYPE_SESSION);
  if (ctx->svchp)  OCIHandleFree((dvoid *)ctx->svchp,  (ub4) OCI_HTYPE_SVCCTX);
  if (ctx->srvhp)  OCIHandleFree((dvoid *)ctx->srvhp,  (ub4) OCI_HTYPE_SERVER);
  if (ctx->errhp)  OCIHandleFree((dvoid *)ctx->errhp,  (ub4) OCI_HTYPE_ERROR);
  if (ctx->envhp)  OCIHandleFree((dvoid *)ctx->envhp,  (ub4) OCI_HTYPE_ENV);
  return status;
}
```

B

Oracle XML DB Restrictions

This appendix describes the restrictions associated with Oracle XML DB.

- *Thin JDBC Driver Not Supported by Some XMLType Functions* – `XMLType` methods `createXML()` with a stream argument, `extract()`, `transform()`, and `existsNode()` work only with the OCI driver. Not all `oracle.xdb.XMLType` functions are supported by the thin JDBC driver. If you do not use `oracle.xdb.XMLType` classes and the OCI driver, you can lose performance benefits.
- *NCHAR, NVARCHAR, and NCLOB Not Supported* – Oracle XML DB does not support the use of SQL data types `NCHAR`, `NVARCHAR`, and `NCLOB` for any of the following:
  - Mapping XML elements or attributes to these data types using the `SQLType` annotation in an XML schema
  - Generating XML data from these data types using SQL/XML functions `XMLElement`, `XMLAttributes`, and `XMLForest`
  - Within SQL/XML functions `XMLQuery` and `XMLTable`, using XQuery functions `ora:view` (deprecated), `fn:doc`, and `fn:collection` on tables that contain columns with these data types

  To handle, store, or generate XML data that contains multibyte characters, Oracle strongly recommends that you use AL32UTF8 as the database character set.
- *XML Identifier Length Limit* – Oracle XML DB supports only XML identifiers that are 4000 characters long or shorter.
- *Repository File Size Limit* – The maximum size of a file in Oracle XML DB Repository is 4 gigabytes. This implies the following limits for different kinds of file data:
  - 4 gigabytes for any LOB, which means 2 gigacharacters for a `CLOB` stored in the database character set AL32UTF8.
  - 4 gigabytes for binary XML encoded data, which typically means more than 4 gigabytes of external XML data before encoding.
  - Indeterminate for XML data stored object-relationally.
- *Repository-Wide Resource Configuration File Limit* – You cannot create more than 125 resource configuration files for repository-wide configuration.
- *Recursive Folder Deletion* – You cannot delete more than 50 levels of nested folders using the option for recursive deletion.
- *No Hybrid Columnar Compression* – Hybrid columnar compression, which is available only for Oracle Exadata Storage Server Software, cannot be used with an

XMLType column that is stored object-relationally or with an XMLType table (no matter which storage model is used). XMLType supports only basic compression and compression for OLTP.

- *No Column-Level Encryption for XMLType* – Column-level encryption is not supported for XMLType. Tablespace-level encryption is supported for all XMLType storage models.
- *No Parallel DML for XMLType* – DML operations on XMLType data are always performed in serial. Parallel DML is not supported for XMLType. (Parallel query and DDL are supported for XMLType.)
- *No XMLType Access over Database Links* – Access to remote XMLType tables or columns is not supported.
- *Oracle JVM Needed for Some Features* – In general, the behavior of Oracle XML DB does not depend on whether or not you have Oracle JVM (Java Virtual Machine) installed. However, if you use Java servlets or PL/SQL package DBMS_XMLSAVE or DBMS_XMLQUERY then you must install Oracle JVM.
- *Editioning Views Not Compatible with XMLType* – Editioning views are not compatible with XMLType data that is stored object-relationally. They cannot be enabled in database schemas that contain persisted object types.

See Also:

- "Oracle XML DB Support for XQuery" on page 5-42 for information about Oracle XML DB support for XQuery
- *Oracle Exadata Storage Server Software User's Guide* for information about Oracle Exadata Storage Server

Index

A

access control entry (ACE)
  definition, 27-3
access control list (ACL)
  child
    definition, 27-9
  constraining inheritance
    definition, 27-9
  definition, 27-4
  extending inheritance
    definition, 27-9
  overview, 1-7
  parent
    definition, 27-9
  system, 27-7
account XDB, 2-1, 3-68, 3-73, 25-20, 26-2, 27-7, 36-3
ACE
  See access control entry
ACL
  See access control list
ACLOID resource property
  definition, 27-7
administering Oracle XML DB, 34-1
Advanced Queuing (AQ)
  IDAP, 37-5
  messaging scenarios, 37-1
  point-to-point support, 37-1
  publish/subscribe support, 37-1
  XMLType queue payloads, 37-5
aggregate privilege
  definition, 27-3
aggregating generated XML data
  XSQL Servlet and SQL function XMLAgg, 18-54
annotations
  XML schema, 3-19, 7-34
    querying to obtain, 7-43
anonymous user, access to repository, 28-6
appendChildXML SQL function, 4-30
atomic privilege
  definition, 27-3
attribute XML Schema data types
  mapping to SQL, 7-47
attributes
  Container, 21-6
  csx:encodingType, 7-39, 7-41
  xdb:columnProps, 7-39
  xdb:defaultSchema, 7-9
  xdb:defaultTable, 7-35, 7-39
  xdb:defaultTableSchema, 7-39
  xdb:maintainDOM, 7-16, 7-35, 7-39, 7-41
  xdb:maintainOrder, 7-40
  xdb:mapUnboundedStringToLob, 7-41
  xdb:maxOccurs, 7-40
  xdb:SQLCollSchema, 7-40
  xdb:SQLCollType, 7-35, 7-40
  xdb:SQLInline, 7-40, 9-4, 9-5
  xdb:SQLName, 7-35, 7-40
  xdb:SQLSchema, 7-40, 7-41
  xdb:SQLType, 7-35, 7-40, 7-41, 9-14
  xdb:srclang, 7-40
  xdb:storeVarrayAsTable, 7-35, 7-41
  xdb:tableProps, 7-40
  xdb:translate, 7-40
  xsi:noNamespaceSchemaLocation, 7-4, 19-8
authenticatedUser role
  DBuri servlet security, 20-28

B

backward-compatible XML schema evolution
  definition, 10-15
binary XML
  definition, 1-15
  encoding types, 7-53
  Schema data types
    mapping to SQL, 7-49
bootstrap ACL
  definition, 27-7
B-tree index, 1-16

C

C API for XML, 16-1
CharacterData interface, 13-9
CheckIn event, repository, 30-4
CheckOut event, repository, 30-4
child ACL
  definition, 27-9
circular dependencies
  XML schemas, 9-18
CLASSPATH Java variable

setting, 31-7
CLOB storage of XML data
definition, 1-15
closeContext PL/SQL procedure, 14-2
collection
in out-of-line tables, 9-8
loading and retrieving large documents, 9-28
XML
definition, 7-32
collection attribute (SQL), 7-52
columnProps attribute, 7-39
complex XLink link
definition, 23-2
See also extended XLink link
complexType
handling cycles, 9-19
handling inheritance, 9-15
mapping
any and anyAttribute declarations, 9-18
fragments to LOBs, 9-14
XML Schema data types to SQL, 7-52
Oracle XML DB restrictions and extensions, 9-15
component of a resource path name
definition, 21-3
compound XML document, 23-3
definition, 23-1
compression
for online transaction processing (OLTP)
using CREATE TABLE, 7-32
using XML schema annotations, 7-43
XMLType support, 1-19, B-1
configuring Oracle XML DB
protocol server, 28-3
repository, 22-1
servlets, 32-3
using DBMS_XDB API, 34-10
using Oracle Enterprise Manager, 34-4
xdbconfig.xml configuration file, 34-5
constraining inheritance, ACLs
definition, 27-9
constraints on XMLType data, 7-34
repetitive XML elements, 9-3
contains SQL function, 6-5
contains XPath function (Oracle), 12-17
content of a resource
definition, 21-4
Content Repository API for Java
See JCR
content-management application
definition, 23-1
Contents element, 21-5
copy-based XML schema evolution, 10-2
copyEvolve PL/SQL procedure, 10-1
COST_XML_QUERY_REWRITE optimizer hint, 5-31
cost-based XML query rewrite
definition, 5-31
Create event, repository, 30-3
CREATE TABLE statement
encoding options for binary XML, 7-31
logging, 7-9
storage options, 7-29
XMLType, 3-3
storage as CLOB, 7-32
CREATE TYPE statement
logging, 7-9
createResource PL/SQL function, 3-9
createXML() XMLType method, 13-2, 15-2
creating an XMLType table, 3-3
for nested collections, 3-29
storage options, 7-29
XML schema-based, 3-28, 7-27
csx:encodingType attribute, 7-39, 7-41
CTXCAT index, 12-13
CTXRULE index, 12-13
cyclical dependencies
XML schemas, 9-18

D

database role
definition, 27-2
database user
definition, 27-2
data-centric use of XML data
definition, 1-15
date and time XML Schema data types
mapping to SQL, 7-50
DAV: WebDAV namespace, 27-5
DBMS_METADATA PL/SQL package, 20-3
reference documentation, 1-5
DBMS_XDB PL/SQL package, 26-1
reference documentation, 1-5
DBMS_XDB_ADMIN PL/SQL package
reference documentation, 1-5
DBMS_XDB_VERSION PL/SQL package, 24-1
reference documentation, 1-5
DBMS_XDBT PL/SQL package, 1-5
DBMS_XDBZ PL/SQL package, 29-13
disable_hierarchy procedure, 27-19
enable_hierarchy procedure, 27-19
is_hierarchy_enabled function, 29-13
purgeLDAPCache procedure, 27-18
reference documentation, 1-5
DBMS_XEVENT PL/SQL package
reference documentation, 1-5
DBMS_XMLDOM PL/SQL package, 13-3
examples, 13-10
reference documentation, 1-5
DBMS_XMLGEN PL/SQL package, 18-22
reference documentation, 1-5
DBMS_XMLINDEX PL/SQL package
modifyParameter procedure, 6-41
reference documentation, 1-5
registerParameter procedure, 6-41
DBMS_XMLPARSER PL/SQL package, 13-19
reference documentation, 1-5
DBMS_XMLQUERY PL/SQL package, 1-6
Oracle Java Virtual Machine dependency, B-2
DBMS_XMLSAVE PL/SQL package, 14-1

Oracle Java Virtual Machine dependency, B-2
reference documentation, 1-6
DBMS_XMLSCHEMA PL/SQL package, 7-6
copyEvolve procedure, 10-1
deleteSchema procedure, 7-12
generateSchema and generateSchemas functions, 9-1
inPlaceEvolve procedure, 10-1
mapping types, 7-45, 7-53
purgeSchema procedure, 7-13
reference documentation, 1-6
registerSchema procedure, 7-7, 9-28
enableHierarchy parameter, 29-3
DBMS_XMLSTORE PL/SQL package, 14-1
reference documentation, 1-6
DBMS_XSLPROCESSOR PL/SQL package, 13-20
reference documentation, 1-6
DBUri
definition, 20-2
generating using sys_DburiGen SQL function, 20-22
identifying a row, 20-17
identifying a target column, 20-17
retrieving column text value, 20-18
retrieving the whole table, 20-16
security, 20-28
servlet, installation, 20-27
DBUri-refs, 20-12
HTTP access, 20-26
DBUriServlet
definition, 20-26
DBURIType
definition, 20-2
debugging
XML schema registration, 7-9
default tables
creating during XML schema registration, 7-10
defaultSchema attribute, 7-9
defaultTable attribute, 7-35, 7-39
defaultTableSchema attribute, 7-39
Delete event, repository, 30-3
deleteSchema PL/SQL procedure, 7-12
deleteXML PL/SQL function, 14-1
deleteXML SQL function, 4-32
deleting
resource, 25-14
XML schema using DBMS_XMLSCHEMA, 7-12
depth SQL function, 25-8
derived XML Schema data types
mapping to SQL, 7-50
directory
See folder
disable_hierarchy PL/SQL procedure, 27-19
document (DOM)
definition, 13-8
document link
definition, 21-7, 23-2
obtaining information about, 23-7
document location hint
definition, 3-24
Document Object Model
See DOM
Document Type Definition
See DTD
document view serialization, JCR
definition, 31-5
DOCUMENT_LINKS public view, 23-7
document-centric use of XML data
definition, 1-16
document-correlated recursive query
definition, 9-25
DOM
definition, 13-1
difference from SAX, 13-4
document
definition, 13-8
fidelity, 7-16
for XML schema mapping, 13-7
SYS_XDBPD$ object attribute, 7-16
using SQL function updateXML, 4-20
Java API for XMLType, 15-1
NamedNodeMap object, 13-9
NodeList object, 13-9
overview, 13-3
PL/SQL API for XMLType, 13-3
DOM fidelity
definition, 7-16
DTD
definition, 7-6
support in Oracle XML DB, 7-6
use with Oracle XML DB, 7-5
dynamic type-checking
XQuery language, 5-37

E

effective text value of a node
definition, 6-16
element XML Schema data types
mapping to SQL, 7-47
elements
Contents, Resource index, 21-5
XDBBinary, 21-11
enable_hierarchy PL/SQL procedure, 27-19
enableHierarchy parameter, DBMS_XMLSCHEMA.registerSchema, 29-3
encodingType attribute, 7-39, 7-41
Enterprise Manager
administering Oracle XML DB, 34-4
entities, XML
using a DTD with binary XML storage, 7-5
equals_path SQL function, 5-36, 25-7
equipartitioning of XMLType tables
definition, 9-10
event
repository, 30-1
configuring, 30-8
predefined, 30-3
event handler, repository
definition, 30-2

event listener, repository
definition, 30-2
evolution, XML schema, 10-1
existsNode Oracle SQL function (deprecated), l
extended XLink link
definition, 23-2
extending inheritance, ACLs
definition, 27-9
extract Oracle SQL function (deprecated), xlix
extracting data from XML, 4-7
extractValue Oracle SQL function (deprecated), xlix

F

fidelity
DOM, 7-16
for XML schema mapping, 13-7
SYS_XDBPD$ object attribute, 7-16
using SQL function updateXML, 4-20
FLWOR XQuery expression, 5-4
fn:available
support, 5-44
fn:collection
avoiding to improve performance, 5-36
support, 5-43
fn:doc
avoiding to improve performance, 5-36
support, 5-43
fn:id
support, 5-43
fn:idref
support, 5-43
fn:matches
support, 5-43
fn:replace
support, 5-43
fn:tokenize
support, 5-43
folder
definition, 21-3
folder link
definition, 21-7
folder sys, repository, 21-2
foldering, 21-1
folder-restricted query
definition, 3-82
fragment, XML
definition, 3-46
SQL operations on, 3-46
fragments, XML
mapping to LOBs, 9-14
freeDocument PL/SQL procedure, 13-10
freeing a DOMdocument instance, 13-10
FROM list order
XMLTable PASSING clause, 5-25
FTP
configuration parameters, 28-4
creating default tables, 7-10
protocol server, features, 28-8
fully qualified XML schema URLs, 9-13
functional evaluation
definition, 3-58
function-based index, 6-5
functions
PL/SQL
createResource, 3-9
deleteXML, 14-1
generateSchema and generateSchemas, 9-1
insertXML, 14-1
is_hierarchy_enabled, 29-13
updateXML, 14-1
SQL
appendChildXML, 4-30
contains, 6-5
deleteXML, 4-32
depth, 25-8
equals_path, 5-36, 25-7
existsNode (deprecated), l
extract (deprecated), xlix
extractValue (deprecated), xlix
insertChildXML, 4-23
insertChildXMLafter, 4-26
insertChildXMLbefore, 4-25
insertXMLafter, 4-29
insertXMLbefore, 4-27
path, 25-7
sys_DburiGen, 20-22
sys_XMLAgg, 18-54
sys_XMLGen, 18-46
under_path, 25-5
updateXML, 4-14
updating XML data, 4-10
XMLAgg, 18-12
XMLAttributes, 18-3
XMLCast, 4-4
XMLCDATA, 18-21
XMLColAttVal, 18-20
XMLComment, 18-16
XMLConcat, 18-11
XMLElement, 18-3
XMLExists, 4-3
XMLForest, 18-10
XMLIsValid, 3-33, 11-7
XMLParse, 18-18
XMLPI, 18-15
XMLQuery, 5-5, 5-6
XMLRoot, 18-19
XMLSequence (deprecated), l
XMLSerialize, 18-17
XMLTable, 5-5, 5-7
XMLtransform, 11-2
XPath
ora:instanceof (deprecated), l
ora:instanceof-only (deprecated), xlviii, l

G

gatherRepositoryStats PL/SQL procedure, 26-2
generateSchema and generateSchemas PL/SQL functions, 9-1

generating XML, 18-1
DBMS_XMLGEN PL/SQL package, 18-22
DBMS_XMLSCHEMA PL/SQL package, 9-1
generateSchema and generateSchemas PL/SQL functions, 9-1
SQL functions, 18-1
sys_XMLAgg SQL function, 18-54
sys_XMLGen SQL function, 18-46
XML schemas, 9-1
XML SQL Utility (XSU), 18-57
XMLAgg SQL function, 18-12
XMLAttributes SQL function, 18-3
XMLCDATA SQL function, 18-21
XMLColAttVal SQL function, 18-20
XMLComment SQL function, 18-16
XMLConcat SQL function, 18-11
XMLElement SQL function, 18-3
XMLForest SQL function, 18-10
XMLParse SQL function, 18-18
XMLPI SQL function, 18-15
XMLRoot SQL function, 18-19
XMLSerialize SQL function, 18-17
XSQL Pages Publishing Framework, 18-54
getBLOBVal() Oracle XMLType method (deprecated), 1
getCLOB() XMLType method, 15-12
getCLOBVal() Oracle XMLType method (deprecated), 1
getNamespace() Oracle XMLType method (deprecated), 1
getNumberVal() XMLType method, 4-2
getObject() XMLType method, 15-3
getOPAQUE() XMLType method, 15-3
getRootElement() Oracle XMLType method (deprecated), 1
getSchemaURL() XMLType method, 7-14
getStringVal() Oracle XMLType method (deprecated), 1
getting JCR repository objects, 31-7
global XML schema
definition, 7-15
using fully qualified URL to override, 9-13
grants declared in an ACL
definition, 27-9
grants defined for an ACL
definition, 27-9
group in an XMLIndex structured component
definition, 6-23

H

hard link
definition, 21-7
JCR, 31-6
hierarchical repository index, 3-86
hierarchy-enabled table
definition, 27-19
HTTP
access for DBUri-refs, 20-26
accessing Java servlet or XMLType, 32-2
accessing repository resources, 21-11
configuration parameters, WebDAV, 28-5
creating default tables, 7-10
improved performance, 28-2
Oracle XML DB servlets, 32-6
protocol server, features, 28-14
requests, 32-6
servlets, 32-2
URIFACTORY, 20-29
using UriRefs to store pointers, 20-3
HTTPUri
definition, 20-2
HTTPURIType
definition, 20-2
hybrid columnar compression, B-1
hybrid storage
definition, 6-4
hybrid storage of XML data
definition, 1-15

I

IDAP
architecture, 37-5
transmitted over Internet, 37-5
IMPORT/EXPORT
in XML DB, 36-3
index
hierarchical repository, 3-86
indexing
CTXCAT, 12-13
CTXRULE, 12-13
function-based, 6-5
options for XMLType, 3-3
Oracle Text, 1-27, 6-48, 12-1
XMLType, 6-4
index-organized tables, 3-19
inheritance
XML schema, restrictions in complexTypes, 9-16
in-place XML schema evolution, 10-15
inPlaceEvolve PL/SQL procedure, 10-1
insertChildXML SQL function, 4-23
insertChildXMLafter SQL function, 4-26
insertChildXMLbefore SQL function, 4-25
insertXML PL/SQL function, 14-1
insertXML() XMLType method, 15-11
insertXMLafter SQL function, 4-29
insertXMLbefore SQL function, 4-27
installing Oracle XML DB, 34-1
instance document
definition, 1-14
specifying root element namespace, 7-4
Internet Data Access Presentation (IDAP)
SOAP specification for AQ, 37-5
Internet Protocol Version 6
FTP, 28-13
HTTP(S), 28-16
IPv6
FTP, 28-13
HTTP(S), 28-16

is_hierarchy_enabled PL/SQL function, 29-13
isSchemaBased() XMLType method, 7-14
IsSchemaValid() XMLType method, 11-7
isSchemaValid() XMLType method, 7-14
isSchemaValidated() XMLType method, 7-14, 11-7

J

Java
  connections, thick and thin, 15-13
  DOM API for XMLType, 15-1
  Oracle XML DB applications, 32-1
  oracle.xml.parser.v2, 15-1
Java Content Repository API
  See JCR
JCR
  compliance levels supported, 31-10
  document view serialization
    definition, 31-5
  files and folders, exposure, 31-3
  getPath() Java method, 31-6
  hard link, 31-6
  logging, 31-9
  Oracle XML DB Repository access, 31-2
  overview, 31-1
  restrictions for Oracle XML DB Content Connector, 31-10
  weak link, 31-6
  XML schema, 31-11
JCR node types
  mapping from XML Schema data types, 31-14
  mapping from XML Schema global element declarations, 31-17
  nt:file, 31-2
  nt:folder, 31-2
  Oracle extensions, 31-5
jcr:content, 31-6
jcr:data property, 31-5
JDBC
  accessing XML documents, 15-2
  drivers, OCI and thin, 15-4
  loading large XML documents, 15-11
  manipulating XML documents, 15-4
JSR-170
  See JCR 1.0

L

large node handling, 13-12
lazy XML loading (lazy manifestation), 13-2
link
  document
    definition, 21-7
  folder
    definition, 21-7
  hard
    definition, 21-7
  repository
    definition, 21-7
  weak
    definition, 21-7
link name
  definition, 21-4
LinkIn event, repository, 30-4
linking, repository
  definition, 3-81
link-properties document
  definition, 3-81
LinkTo event, repository, 30-4
loading
  large documents with collections, 9-28
loading large XML documents using JDBC, 15-11
loading of XML data, lazy, 13-2
LOB locator, 24-6
LOBs
  mapping XML fragments to, 9-14
local XML schema
  definition, 7-14
  using fully qualified URL to specify, 9-13
Lock event, repository, 30-4

M

maintainDOM attribute, 7-16, 7-35, 7-39, 7-41
maintainOrder attribute, 7-40
manifestation, lazy, 13-2
mapping
  complexType any and anyAttributes declarations, 9-18
  complexType to SQL
    out-of-line storage, 9-4
  overriding using SQLType attribute, 7-48
  simpleContent to object types, 9-17
mapping XML Schema complexType data types to SQL, 7-52
mapping XML Schema data types to binary XML encoding types, 7-53
mapping XML Schema data types to SQL data types, 7-45
mapping XML Schema to SQL
  simpleType, 7-48
mapUnboundedStringToLob attribute, 7-41
matches XQuery function (Oracle), 5-12
maxOccurs attribute, 7-40
metadata
  definition, 29-1
  system-defined
    definition, 1-7
  user-defined
    definition, 1-7
methods
  XMLType
    createXML(), 13-2, 15-2
    getBLOBVal() (deprecated), 1
    getCLOB(), 15-12
    getCLOBVal() (deprecated), 1
    getNamespace() (deprecated), 1
    getNumberVal(), 4-2
    getObject(), 15-3
    getOPAQUE(), 15-3
    getRootElement() (deprecated), 1

getSchemaURL(), 7-14
getStringVal() (deprecated), l
insertXML(), 15-11
isSchemaBased(), 7-14
IsSchemaValid(), 11-7
isSchemaValid(), 7-14
isSchemaValidated(), 7-14, 11-7
schemaValidate(), 7-14
setObject(), 15-4
setSchemaValidated(), 7-14, 11-7
writeToStream(), 21-12
XML schema, 7-13
MIME
overriding with DBUri servlet, 20-27
mix:referenceable, 31-5
mixed XML content
definition, 1-16
modifyParameter PL/SQL procedure, 6-41

N

NamedNodeMap object (DOM), 13-9
namespace
prefixes
ocjr, 31-5
URL for XML schema, 7-4
XQuery, 5-10, 5-28
naming SQL objects, 7-34
navigational access to repository resources, 21-9
nested XML
generating using DBMS_XMLGEN, 18-33
generating with XMLElement, 18-7
nested XMLAgg functions and XSQL, 18-54
NESTED_TABLE_ID pseudocolumn, 3-28
newDOMDocument() function, 13-9
NodeList object (DOM), 13-9
nodes, large (DBMS_XMLDOM), 13-12
nodes, large (Java), 15-16
noNamespaceSchemaLocation attribute, 7-4, 19-8
non-schema-based view
definition, 19-1
nt:file JCR node type, 31-2
nt:folder, 31-2
nt:folder JCR node type, 31-2
numeric XML Schema data types
mapping to SQL, 7-49

O

object attributes
for collection (SQL), 7-52
of XMLFormat, 18-48
REF, 9-5, 9-20
sys_DburiGen SQL function
passing to, 20-23
SYS_XDBPD$, 7-16
XMLType, in AQ, 37-5
object identifier
definition, 27-7
OBJECT_ID column of XDB$ACL table, 27-7
object-based persistence of XML data
definition, 1-15
object-relational storage of XML data
definition, 1-15
occurrence indicator
definition, 5-4
OCI API for XML, 16-1
ocjr namespace prefix, 31-5
OCT
definition, 3-19
ODP.NET, 17-1
OID
See object identifier
ojcr:folder, 31-5
operator
See functions, SQL
optimizer hints
COST_XML_QUERY_REWRITE, 5-31
optimizing repository queries, 26-2
ora:contains XPath function (Oracle), 12-17
policy
definition, 12-18
ora:defaultTable XQuery pragma, 5-14, 5-37
ora:instanceof Oracle XPath function (deprecated), l
ora:instanceof-only Oracle XPath function (deprecated), xlviii, l
ora:invalid_path XQuery pragma, 5-14
ora:matches XQuery function (Oracle), 5-12
ora:replace XQuery function (Oracle), 5-12
ora:sqrt XQuery function (Oracle), 5-13
ora:tokenize XQuery function (Oracle), 5-13
ora:view_on_null XQuery pragma, 5-14
ora:xq_proc XQuery pragma, 5-15, 6-30
ora:xq_qry XQuery pragma, 5-15
Oracle ASM files
accessing, 21-12
using FTP, 28-11
Oracle ASM virtual folder, 21-6
Oracle Data Provider for .NET, 17-1
Oracle Enterprise Manager
administering Oracle XML DB, 34-4
Oracle extensions to JCR node types, 31-5
Oracle Internet Directory, 27-20
Oracle Net Services, 1-2
Oracle Text
contains SQL function and XMLType, 6-5
index, 1-27, 6-48, 12-1
searching for resources, 25-20
searching XML in CLOB instances, 1-27
Oracle XML DB
access models, 2-3
advanced queuing, 1-27
architecture, 1-2
features, 1-12
installation, 34-1
Java applications, 32-1
overview, 1-1
Repository
See repository
upgrading, 34-3
versioning, 24-1

when to use, 2-1
Oracle XML DB Content Connector, 31-1
how to use, 31-7
logging API, 31-9
overview, 31-2
restrictions, 31-10
sample code to upload file, 31-8
See also JCR
OracleRepository, 31-7
oracle.xdb.XMLType Java class, 15-1, 15-14
oracle.xml.parser.v2 Java package, 15-1
order index of XMLIndex
definition, 6-13
ordered collection
definition, 3-19
ordered collection table (OCT)
definition, 3-19
ordered collections in tables (OCTs)
default storage of varray, 7-52
out-of-line storage, 9-4
collections, 9-8
XPath rewrite, 9-7

P

parent ACL
definition, 27-9
partial update of XML data
definition, 4-12
partial validation of XML data
definition, 3-32
partitioning XMLType tables, 9-10
binary XML, 3-3
PASSING clause of XMLTable
FROM list order, 5-25
path component of a resource path name
definition, 21-3
path index of XMLIndex
definition, 6-13
path name
definition, 21-3
resolution, 21-6
path SQL function, 25-7
path table of XMLIndex, 6-14
PATH_VIEW, 25-1
path-based access to repository resources, 21-9
path-index trigger
definition, 27-19
PD (positional descriptor), 7-16
persistence models for XML data, 1-14
PL/SQL functions
See functions, PL/SQL
PL/SQL packages
DBMS_METADATA, 20-3
reference documentation, 1-5
DBMS_XDB, 26-1
reference documentation, 1-5
DBMS_XDB_ADMIN
reference documentation, 1-5
DBMS_XDB_VERSION, 24-1
reference documentation, 1-5
DBMS_XDBT, 1-5
DBMS_XDBZ, 27-19, 29-13
reference documentation, 1-5
DBMS_XEVENT
reference documentation, 1-5
DBMS_XMLDOM, 13-3
reference documentation, 1-5
DBMS_XMLGEN, 18-22
reference documentation, 1-5
DBMS_XMLINDEX
reference documentation, 1-5
DBMS_XMLPARSER, 13-19
reference documentation, 1-5
DBMS_XMLQUERY, 1-6
Oracle Java Virtual Machine dependency, B-2
DBMS_XMLSAVE, 14-1
Oracle Java Virtual Machine dependency, B-2
reference documentation, 1-6
DBMS_XMLSCHEMA
reference documentation, 1-6
See DBMS_XMLSCHEMA PL/SQL package
DBMS_XMLSTORE, 14-1
reference documentation, 1-6
DBMS_XSLPROCESSOR, 13-20
reference documentation, 1-6
for XMLType, 13-1
PL/SQL procedures
See procedures, PL/SQL
point-to-point
support in AQ, 37-1
policy for ora:contains XPath function (Oracle)
definition, 12-18
ports
configuring
FTP, 28-4
HTTP, 28-5
HTTPS, 28-6
positional descriptor (PD), 7-16
post-parse persistence of XML data
definition, 1-15
pragmas
XQuery
Oracle, 5-14
predefined ACLs, 27-7
preference
Oracle Text indexing
definition, 12-13
pretty-printing, 18-17
in book examples, xliv
not done by SQL/XML functions, 3-60
Web service output, 33-5
primitive XML Schema data types
mapping to SQL, 7-50
principal
definition, 27-2
private (local) XML schema, definition, 7-14
privilege
definition, 27-3
procedures

PL/SQL
closeContext, 14-2
copyEvolve, 10-1
disable_hierarchy, 27-19
enable_hierarchy, 27-19
freeDocument, 13-10
gatherRepositoryStats, 26-2
inPlaceEvolve, 10-1
modifyParameter, 6-41
processXSL, 13-22
purgeLDAPCache, 27-18
registerParameter, 6-41
registerSchema, 7-7
setKeyColumn, 14-2
setUpdateColumn, 14-2
processXSL PL/SQL procedure, 13-22
protocol server, 28-1
architecture, 28-2
configuration parameters, 28-3
event-based logging, 28-8
FTP, 28-8
configuration parameters, 28-4
HTTP, 28-14
configuration parameters, 28-5
WebDAV, 28-20
configuration parameters, 28-5
protocols, access to repository resources, 21-10
public (global) XML schema, definition, 7-14
publish/subscribe
support in AQ, 37-1
purchase-order XML document
used in full-text examples, 12-26
purchase-order XML schema, 3-14
annotated, 3-20, A-30
graphical representation, 3-17
revised, 10-2, A-33
purgeLDAPCache PL/SQL procedure, 27-18
purgeSchema PL/SQL procedure, 7-13

Q

qualified XML schema URLs, 9-13
query-based access to resources
using RESOURCE_VIEW and PATH_VIEW, 25-2
using SQL, 21-13
querying XMLType data
choices, 4-1
transient data, 4-7

R

recursive schema support, 9-24
REF object attribute, 9-5, 9-20
REGISTER_NT_AS_IOT option for XML schema registration, 3-19, 7-7
registered XML schemas, list of, 7-11
registering an XML schema, 7-7
debugging, 7-9
default tables, creating, 7-10
SQL object types, creating, 7-9
registering an XML schema for JCR, 31-11
registerParameter PL/SQL procedure, 6-41
registerSchema PL/SQL procedure, 7-7
renaming an XMLIndex index, 6-18
Render event, repository, 30-3
replace XQuery function (Oracle), 5-12
repository, 21-3
access by anonymous user, 28-6
access using JCR, 31-2
data storage, 21-5
event, 30-1
configuring, 30-8
predefined, 30-3
event handler
definition, 30-2
event listener
definition, 30-2
hierarchical index, 3-86
resource
See resource
use with XQuery, 5-17
repository link
definition, 21-7
repository objects, 31-7
RESID
definition, 24-2
resource
access, 21-3
using protocols, 28-7
definition, 1-7, 29-1
deleting, 21-7
nonempty container, 25-15
using DELETE, 25-14
managing with DBMS_XDB, 26-1
required privileges for operations, 27-4
searching for, using Oracle Text, 25-20
setting property in ACLs, 27-11
simultaneous operations, 25-18
updating, 25-15
resource configuration file
definition, 22-1
resource configuration list
definition, 22-2
resource content
definition, 21-4
resource document
definition, 3-73
resource ID
definition, 24-2
resource name
definition, 21-4
resource version
definition, 24-2
RESOURCE_VIEW
explained, 25-1
resource-view-cache-size configuration parameter, 25-20
retrieving large documents with collections, 9-28
rewrite
XPath (XPath), 8-1
XQuery, 5-30

role
authenticatedUser, DBuri servlet, 20-28
database
definition, 27-2
XDB_SET_INVOKER, 30-8
XDBADMIN, 7-15, 22-2, 27-7, 30-8
root folder, repository, 21-1
root XML Schema
definition, 7-9
rule-based XML query rewrite
definition, 5-31

S

scalar value
converting to XML document using sys_XMLGen, 18-48
schema evolution
See XML schema evolution
schema for schemas (XML Schema)
definition, 1-13
schema location hint
definition, 3-24
schemaValidate() XMLType method, 7-14
searching CLOB instances, 1-27
security
DBUri, 20-28
semi-structured XML data
definition, 1-16
servlets
accessing repository data, 21-14
APIs, 32-7
configuring, 32-3
session pooling, 32-7
writing, 32-7
in Java, 32-2
XML manipulation, 32-2
session pooling, 32-7
protocol server, 28-2
setKeyColumn PL/SQL procedure, 14-2
setObject() XMLType method, 15-4
setSchemaValidated() XMLType method, 7-14, 11-7
setUpdateColumn PL/SQL procedure, 14-2
simple XLink link
definition, 23-2
simpleContent
mapping to object types, 9-17
simpleType XML Schema data types
mapping to SQL, 7-48
SOAP, 33-1
access through Advanced Queuing, 1-2
IDAP, 37-5
SQL function
See functions, SQL
SQL functions
See functions, SQL
SQL object types
creating during XML schema registration, 7-9
SQL operator
See functions, SQL
SQL*Loader, 35-1
SQL*Plus
XQUERY command, 5-38
SQLCollSchema attribute, 7-40
SQLCollType attribute, 7-35, 7-40
SQLInline attribute, 7-40, 9-4, 9-5
SQLJ, 15-14
SQLName attribute, 7-35, 7-40
SQLSchema attribute, 7-40, 7-41
SQLType attribute, 7-35, 7-40, 7-41, 9-14
SQL/XML generation functions
definition, 1-20
SQL/XML publishing functions
definition, 1-20, 18-3
SQL/XML query and access functions
definition, 1-20
SQL/XML standard
generating XML data, 18-2
querying XML data
XMLExists and XMLCast, 4-2
XMLQuery and XMLTable, 5-5
sqrt XQuery function (Oracle), 5-13
srclang attribute, 7-40
standard metadata, 31-5
static type-checking
XQuery language, 5-37
storage
out of line, 9-4
collections, 9-8
XMLType, CREATE TABLE, 7-32
storage models for XMLType, 1-14
storeVarrayAsTable attribute, 7-35, 7-41
string XML Schema data types
mapping to SQL, 7-49
mapping to VARCHAR2 vs CLOB, 7-51
structured storage of XML data
definition, 1-15
structured XMLIndex component
definition, 6-9
style sheet for updating XML instance documents, 10-10
sys folder, repository, 21-2
sys_DburiGen SQL function, 20-22
inserting database references, 20-24
retrieving object URLs, 20-25
use with text node test, 20-23
SYS_NC_ARRAY_INDEX$ column, 3-28
SYS_XDBPD$ object attribute, 7-16
sys_XMLAgg SQL function, 18-54
sys_XMLGen SQL function, 18-46
converting XMLType instances, 18-51
object views, 18-52
XMLFormat object attributes, 18-48
XMLGenFormatType object, 18-48
SYSAUX tablespace, 34-1
system ACL
definition, 27-7
system ACLs, 27-7
system-defined metadata
definition, 1-7

T

tableProps attribute, 7-40
tables
  index-organized, 3-19
tablespace
  SYSAUX, 34-1
text value of a node, effective
  definition, 6-16
text-based persistence of XML data
  definition, 1-15
third-party XLink link
  definition, 23-2
tokenize XQuery function (Oracle), 5-13
translate attribute, 7-40
trigger, path-index
  definition, 27-19
type-checking, static and dynamic
  XQuery language, 5-37

U

UDT
  generating an element from, 18-9
UncheckOut event, repository, 30-4
under_path SQL function, 25-5
uniform access control mechanism
  definition, 27-19
unique constraint on parent element of an attribute, 9-3
UnLinkIn event, repository, 30-4
Unlock event, repository, 30-4
unresolved XLink and XInclude links, 23-9
unstructured storage of XML data
  definition, 1-15
unstructured XMLIndex component
  definition, 6-9
Update event, repository, 30-4
updateXML PL/SQL function, 14-1
updateXML SQL function, 4-14
  mapping NULL values, 4-18
updating repository resource, 25-15
updating XML data
  partial update
    definition, 4-12
  updating same node more than once, 4-20
  using SQL functions, 4-10
    optimization, 4-20
upgrading Oracle XML DB, 34-3
URIFACTORY PL/SQL package
  configuring to handle DBURI-ref, 20-29
  creating subtypes of URIType, 20-20
Uri-reference
  database and session, 20-15
  DBUri-ref, 20-12
  HTTP access for DBUri-ref, 20-26
  URIFACTORY PL/SQL package, 20-20
  URIType examples, 20-8
URIType
  examples, 20-8
user
  definition, 27-2
user XDB, 2-1, 3-68, 3-73, 25-20, 26-2, 27-7, 36-3
user-defined metadata, 31-6
  definition, 1-7

V

validating
  examples, 11-8
  IsSchemaValid() XMLType method, 11-7
  isSchemaValidated() XMLType method, 11-7
  setSchemaValidated() XMLType method, 11-7
  XMLIsValid SQL function, 11-7
    use as CHECK constraint, 3-33
validation of XML data, partial
  definition, 3-32
value index of XMLIndex
  definition, 6-13
varray in a LOB
  definition, 3-19
varray in a table
  definition, 3-19
VCR
  See version-controlled resource
version resource
  definition, 24-2
version series of a resource
  definition, 24-3
versionable resource
  definition, 24-2
VersionControl event, repository, 30-4
version-controlled resource
  definition, 24-2
versioning, 1-7, 24-1
views
  RESOURCE and PATH, 25-1

W

weak link
  definition, 21-7
  deletion, 23-9
  JCR, 31-6
Web service, 33-1
  pretty-printing output, 33-5
WebDAV protocol server, 28-20
WebFolder
  creating in Windows 2000, 28-22
well-formed XML document
  definition, 3-31
writeToStream() XMLType method, 21-12
WSDL
  Web service for accessing stored PL/SQL, 33-6
  Web service for database queries, 33-3

X

XDB database schema (user account), 2-1, 3-68, 3-73, 25-20, 26-2, 27-7, 36-3
xdb namespace, 27-5
XDB$ACL table, 27-7

xdb:columnProps attribute, 7-39
xdb:defaultSchema attribute, 7-9
xdb:defaultTable attribute, 7-35, 7-39
xdb:defaultTableSchema attribute, 7-39
xdb:maintainDOM attribute, 7-16, 7-35, 7-39, 7-41
xdb:maintainOrder attribute, 7-40
xdb:mapUnboundedStringToLob attribute, 7-41
xdb:maxOccurs attribute, 7-40
xdb:SQLCollSchema attribute, 7-40
xdb:SQLCollType attribute, 7-35, 7-40
xdb:SQLInline attribute, 7-40, 9-4, 9-5
xdb:SQLName attribute, 7-35, 7-40
xdb:SQLSchema attribute, 7-40, 7-41
xdb:SQLType attribute, 7-35, 7-40, 7-41, 9-14
xdb:srclang attribute, 7-40
xdb:storeVarrayAsTable attribute, 7-35, 7-41
xdb:tableProps attribute, 7-40
xdb:translate attribute, 7-40
XDB_SET_INVOKER role, 30-8
XDBADMIN role, 7-15, 22-2, 27-7, 30-8
XDBBinary element, 21-11
  definition, 21-4
xdbconfig.xml configuration file, 34-5
xdbcore parameters, 9-29
xdbcore-loadableunit-size configuration parameter, 9-29, 25-20
xdbcore-xobmem-bound configuration parameter, 9-29, 25-20
XDBUri, 20-3
  definition, 20-3, 20-10
XDBURIType
  definition, 20-3
  using constructor to expand compound documents (XInclude), 23-5
XInclude, 23-1
  definition, 23-1
  unresolved link, 23-9
XLink, 23-1
  complex link
    definition, 23-2
  definition, 23-1
  extended link
    definition, 23-2
  link types, 23-2
  simple link
    definition, 23-2
  third-party link
    definition, 23-2
  unresolved link, 23-9
XML attributes
  See attributes
XML diagnosability mode, 5-35
XML entities
  using a DTD with binary XML storage, 7-5
XML fragment
  definition, 3-46
  mapping to LOBs, 9-14
  SQL operations on, 3-46
XML instance document
  definition, 1-14
XML Object (XOB), 2-7
XML query rewrite
  definition, 5-30
    cost-based, 5-31
    rule-based, 5-31
XML Schema
  definition, xliv
XML schema
  annotations, 3-19, 7-34
    querying to obtain, 7-43
  circular dependencies, 9-18
  complexType declarations, 9-15, 9-19
  cyclical dependencies, 9-18
  definition, 1-13, 7-2
  deletion, 7-12
  evolution, 10-1
    backward-compatible definition, 10-15
  for XML schemas that can be registered, 7-9
  generating from object-relational type, 9-1
  inheritance in, complexType restrictions, 9-16
  local and global, 7-14
  managing and storing, 7-9
  mapping to SQL object types, 13-6
  registration with Oracle XML DB, 7-7
    for use with JCR, 31-11
  updating after registering, 10-1
  URLs, 9-13
  use with JCR, 31-11
  use with Oracle XML DB, 7-5
  W3C Recommendation, 3-13, 7-1
  XMLType methods, 7-13
XML Schema data types
  mapping to binary XML encoding types, 7-53
  mapping to JCR node types, 31-14
  mapping to SQL data types, 7-45
XML schema definition
  definition, 1-13
XML schema evolution
  copy-based, 10-2
  in-place, 10-15
XML schema-based tables and columns, creating, 7-27
XML schema-based view
  definition, 19-1
XML SQL Utility (XSU), generating XML, 18-57
XML_DB_EVENTS parameter, 30-8
XMLAgg SQL function, 18-12
XMLAttributes SQL function, 18-3
XMLCast SQL function, 4-4
XMLCDATA SQL function, 18-21
XMLColAttVal SQL function, 18-20
XMLComment SQL function, 18-16
XMLConcat SQL function, 18-11
XMLElement SQL function, 18-3
XMLExists SQL function, 4-3
XMLEXTRA object column, 9-22
XMLForest SQL function, 18-10
XMLFormat
  XMLAgg, 18-13

XMLFormat object type
sys_XMLGen
XMLFormatType object, 18-48
XMLGenFormatType object, 18-48
XMLIndex
creating index, 6-18
dropping index, 6-18
order index
definition, 6-13
path index
definition, 6-13
path table, 6-14
renaming index, 6-18
structured component
definition, 6-9
unstructured component
definition, 6-9
value index
definition, 6-13
XMLIsValid SQL function, 3-33, 11-7
XMLOptimizationCheck SQL*Plus setting, 5-35
XMLOptimizationCheck SQL*Plus system variable, 0-xlviii, 5-35
XMLParse SQL function, 18-18
XMLPI SQL function, 18-15
XMLQuery SQL function, 5-5, 5-6
XMLRoot SQL function, 18-19
XMLSequence Oracle SQL function (deprecated), 1
XMLSerialize SQL function, 18-17
XMLTable SQL function, 5-5, 5-7
breaking up an XML fragment, 3-46
breaking up multiple levels of XML data, 3-49
PASSING clause and FROM list order, 5-25
XMLtransform SQL function, 11-2
XMLType
as abstract data type, 1-14
benefits, 3-2
constructors, 3-5
contains SQL function, 6-5
CREATE TABLE statement, 7-32
DBMS_XMLDOM PL/SQL API, 13-3
DBMS_XMLPARSER PL/SQL API, 13-19
DBMS_XSLPROCESSOR PL/SQL API, 13-20
definition, 1-12
extracting data, 4-7
indexing columns, 6-4
instances, PL/SQL APIs, 13-1
loading data, 35-1
loading with SQL*Loader, 35-1
methods
createXML(), 13-2, 15-2
getCLOB(), 15-12
getNumberVal(), 4-2
getObject(), 15-3
getOPAQUE(), 15-3
getSchemaURL(), 7-14
insertXML(), 15-11
isSchemaBased(), 7-14
IsSchemaValid(), 11-7
isSchemaValid(), 7-14
isSchemaValidated(), 7-14, 11-7
schemaValidate(), 7-14
setObject(), 15-4
setSchemaValidated(), 7-14, 11-7
writeToStream(), 21-12
XML schema, 7-13
PL/SQL packages, 13-1
querying, 4-1
querying transient data, 4-7
querying XMLType columns, 4-7
queue payloads, 37-5
storage models, 1-14
table, querying with JDBC, 15-2
tables, views, columns, 7-27
views, access with PL/SQL DOM APIs, 13-7
XOB, 2-7
XPath language
functions
ora:contains (Oracle), 5-12, 12-17
syntax, 4-2
See also XQuery language
XPath rewrite, 8-1
definition, 5-30
indexes on singleton elements and attributes, 6-6
out-of-line storage, 9-7
XQuery
extension expressions, 5-14
pragmas
Oracle, 5-14
XQUERY command, SQL*Plus, 5-38
XQuery functions and operators
support, 5-43
XQuery language, 5-1
expressions, 5-3
FLWOR, 5-4
rewrite, 5-30
functions
ora:contains (Oracle), 5-12, 12-17
ora:matches (Oracle), 5-12
ora:replace (Oracle), 5-12
ora:sqrt (Oracle), 5-13
ora:tokenize (Oracle), 5-13
item
definition, 5-2
namespaces, 5-10, 5-28
optimization, 5-30
optimization over relational data, 5-31
Oracle extension functions, 5-11
Oracle XML DB support, 5-42
performance, 5-30
predefined namespaces and prefixes, 5-10
referential transparency
definition, 5-2
sequence
definition, 5-2
SQL*Plus XQUERY command, 5-38
tuning, 5-30
type-checking, static and dynamic, 5-37
unordered mode
definition, 5-2

use with Oracle XML DB Repository, 5-17
use with XMLType relational data, 5-23
optimization, 5-32
XMLQuery and XMLTable SQL functions, 5-5
examples, 5-15
XSD
definition, 1-13
xsi:noNamespaceSchemaLocation attribute, 7-4, 19-8
XSL style sheet
definition, 13-20
XSLT
style sheets
for updating XML instance documents, 10-10
use with DBUri servlet, 3-88
use with Oracle XML DB, 3-64
use with package DBMS_
XSLPROCESSOR, 13-22
XSQL Pages Publishing Framework
generating XML, 18-2, 18-54
XSU, generating XML, 18-57